Tarea 4 - ISIS 1105

Santiago Muñoz Martínez (202221167)

April 11, 2025

1 Problema 1 - Transformación de Estructuras

1.1 Matriz de Adyacencias a Lista de Adyacencias

Se propone el siguiente código en GCL para pasar un grafo bajo una matriz de adyacencias a uno bajo una lista de adyacencias, donde las listas albergan tuplas y, en cada tupla, el primer elemento denota el nodo conectado y el segundo elemento denota el peso de la arista.

```
fun matToList(graph: matrix[0, n)×[0, n) of int) ret graphList: array [0, n) of List<tuple>
    var graphList: array[0,n) of List<tuple>
    for (int i := 0; i < n; i++) →
        var adjacentList: LinkedList<tuple>;
        graphList[i] := adjacentList;
    rof
    for (int i := 0; i < n; i++) →
        for (int j := i+1; j < n; j++) →
            var weight: int;
            weight := graph[i][j];
            if weight > 0 →
                var edge1: tuple<int, int>;
                edge1[0] := j;
                edge1[1] := weight;
                var edge2: tuple<int, int>;
                edge2[0] := i;
                edge2[1] := weight;
                graphList[i].add(edge1);
                graphList[j].add(edge2);
            fi
        rof
    rof

ret graphList
```

El algoritmo funciona recorriendo la matriz que representa el grafo y generando las listas de adyacencia correspondientes a cada uno de las filas de dicha matriz. Ya que trabajamos con un grafo no dirigido, la matriz es simétrica (con respecto a su diagonal principal), por lo que solo se debe revisar una de estas dos mitades; en este caso revisamos la mitad superior. Además, dado que no hay autociclos, nunca revisamos la diagonal principal. Por ejemplo, para la primera fila (i = 0) empezamos a revisar desde la segunda columna (j = i+1 = 1), para la segunda fila empezamos desde la tercera columna y así sucesivamente.

Con la implementación clara, vemos que el grueso de las operaciones corresponde a encontrar un peso distinto de 0 en la matriz, generar las tuplas correspondientes a la arista y agregarlas a las listas de adyacencia. En esto estamos revisando $n^2/2 - n$ casillas de la matrix (la mitad del total menos la diagonal principal) y,

dado que para todas revisamos si el peso es distinto de 0, por lo menos se hacen $n^2/2 - n$ operaciones en el doble ciclo. Además, inicialmente hicimos n creaciones de listas de adyacencia para inicializar la estructura. Con esto, tendríamos que la complejidad temporal está dada por

$$T(n) = O(n^2/2 - n + n) = O(n^2/2)$$

lo cual por notación asintótica se aproxima a

$$T(n) = O(n^2)$$

Note que como debemos revisar todas las casillas, la complejidad no depende del número de aristas que tenga el grafo dado que no podemos solo pasar las que sí tienen peso, pues no sabemos cuáles son exactamente.

1.2 Lista de Adyacencias a Matriz de Adyacencias

Se propone el siguiente código en GCL para pasar un grafo bajo una lista de adyacencias a uno bajo una matriz de adyacencias, donde las listas albergan tuplas y, en cada tupla, el primer elemento denota el nodo conectado y el segundo elemento denota el peso de la arista.

```
fun listToMatrix(graph: array [0, n) of List<tuple>) ret graphMatrix: matrix[0, n)×[0, n) of int
    var graphMatrix: matrix[0, n)×[0, n) of int>
    for (int i := 0; i < n; i++) →
            for (int j := 0; j < n; j++) →
                    graphMatrix[i][j] := 0;
            rof
    rof
    for (int i := 0; i < n; i++) →
            var adjacentList: List<tuple>
            adjacentList := graph[i];
            for (tuple edge : adjacentList) →
                    var weight, destination: int;
                    destination := edge[0];
                    weight := edge[1];
                    if graphMatrix[i][destination] = 0 → graphMatrix[i][destination] := weight;
                    fi
            rof
    rof

ret graphMatrix
```

El algoritmo recorre el arreglo de listas de adyacencia y, para cada lista, va añadiendo el peso de cada arista a la casilla correspondiente en la matriz utilizada para representar el grafo, declarada con anterioridad. Además, dado que trabajamos con un grafo no dirigido, con toda seguridad encontraremos cada arista dos veces, por lo que el algoritmo revisa que una casilla sea 0 antes de cambiarla; si no es 0, ya fue cambiada y no hace nada.

En cuanto a la complejidad temporal, vemos que tenemos un doble ciclo anidado, donde el ciclo exterior se ejecuta n veces, mientras que el interior se ejecuta tantas veces como aristas tenga el nodo actual del ciclo exterior. Efectivamente, estamos visitando todas las aristas del grafo pero, como se trata de uno no dirigido, cada arista la visitamos dos veces. Dado que por cada arista realizamos por lo menos tres operaciones, esto es lo que termina cargando la mayor complejidad del doble ciclo. Además, inicializar la matriz requiere n^2 operaciones. Con esto, podemos resumir la complejidad asintótica del algoritmo como

$$T(n, m) = O(n^2 + 2m) = O(n^2 + m)$$

En caso que el grafo sea denso, m se acerca a su máximo de $n(n-1)$, por lo que la complejidad temporal se podría aproximar a

$$T(n) \approx O(n^2)$$

2 Problema 2 - Implementación Algoritmos MST

2.1 Algoritmo de Prim

Este algoritmo fue implementado en Java y acá se plasma el GCL correspondiente. Sigue el funcionamiento general de Prim, en el cual se empieza por un nodo cualquiera y se va añadiendo la arista de menor peso a un nodo no conectado ya al MST. Siguiendo el enunciado, el grafo a usar está implementado como una matriz de adyacencias de tamaño $n \times n$.

```
fun algoritmoPrim(grafo: matrix[0, n)×[0, n) of int, nodoInicial: int) ret aristasIncluidas: List<List<int>>
    var nodosVisitados: Set<int> := new HashSet();
    var aristasIncluidas: List<List<int>> := new ArrayList();
    nodosVisitados.add(nodoInicial);

    var distancia: matrix[0, n)×[0, 2) of int;
    for (int i := 0; i < n; i++) →
        if grafo[nodoInicial][i] ≠ 0 →
            distancia[i][0] := nodoInicial;
            distancia[i][1] := grafo[nodoInicial][i];
        [ ] grafo[nodoInicial][i] = 0 →
            distancia[i][0] := -1;
            distancia[i][1] := INF;
        fi
    rof

    do nodosVisitados.size() < n →
        var nodoOrigenMenorPeso: int;
        var nodoDestinoMenorPeso: int;
        nodoOrigenMenorPeso := -1;
        nodoDestinoMenorPeso := -1;
        var menorPeso: int;
        menorPeso := INF;

        for (int i := 0; i < n; i++) →
            if distancia[i][1] < menorPeso and i ∉ nodosVisitados →
                nodoOrigenMenorPeso := distancia[i][0];
                nodoDestinoMenorPeso := i;
                menorPeso := distancia[i][1];
            fi
        rof

        var arista: List<int>;
        arista := new ArrayList();
        arista.add(nodoOrigenMenorPeso);
        arista.add(nodoDestinoMenorPeso);
        arista.add(menorPeso);
        aristasIncluidas.add(arista);

        nodosVisitados.add(nodoDestinoMenorPeso);
        actualizarDistancia(grafo, nodoDestinoMenorPeso, distancia);
    od

ret aristasIncluidas
```

```
fun actualizarDistancia(grafo: matrix[0,n)×[0,n) of int, nodoAgregado: int, distancia: matrix[0,n)×[0,2) of int) ret null
    for (int i := 0; i < distancia.length; i++) →
        var nuevaDistancia, distanciaActual: int;
        nuevaDistancia := grafo[i][nodoAgregado];
        distanciaActual := distancia[i][1];
        if nuevaDistancia ≠ 0 and nuevaDistancia < distanciaActual →
            distancia[i][0] := nodoAgregado;
            distancia[i][1] := nuevaDistancia;
        fi
    rof
```

i) **Explicación implementación**

Como se mencionó anteriormente, la implementación sigue el funcionamiento canónico del algoritmo de Prim: añadir nodos no visitados uno a uno al árbol mediante la arista de menor peso que los conecte al MST. Inicialmente desarrollé un código que, para encontrar dicha arista de menor peso, recorría todas las aristas de todos los nodos ya visitados y guardaba la de menor peso. Aunque esto es correcto, es bastante ineficiente, pues el algoritmo recorre demasiadas aristas, unas casi n veces, sin que haya una verdadera necesidad de hacer esto.

Para optimizar el algoritmo me basé en las recomendaciones de esta publicación en Stack Overflow, la cual detalla como conseguir que Prim tenga una complejidad de $O(n^2)$ bajo una matriz de adyacencias, donde n es el número de vertices del grafo. La mejora sobre el acercamiento ingenuo que había implementado originalmente radica en la inclusión de un arreglo llamado **distancia**, de tamaño n, el cual guarda para cada nodo del grafo la arista de mínima distancia que conecta a dicho nodo con el MST. Si no hay arista que conecte al nodo con el MST, el peso se puede modelar como un entero muy grande. Originalmente, el arreglo de **distancia** únicamente guarda el peso de la arista. Sin embargo, dado que necesitaba poder reconstruir el árbol, modifiqué la estructura para que fuera una matriz $n \times 2$, donde cada fila corresponde a un nodo, la primera columna corresponde al nodo en el MST con el cual el nodo de la fila se conecta con mínimo peso y la segunda columna corresponde al peso de dicha arista.

Con esto claro, pasamos a un resumen del algoritmo. Este inicia generando un conjunto de nodosVisitados, que nos permitirá saber cuando parar de añadir aristas al MST, así como una lista de aristas, representadas como una lista en si misma, que corresponderán al retorno del algoritmo. Como parte de la inclusión de **distancia** al algoritmo, esta se debe inicializar con el nodo de inicio, que llega por parametro, como el único en el MST. Por ende, se recorre el arreglo y en cada nodo se revisa si este tiene una conexión directa con el nodo de inicio. Si este es el caso, la primera columna se actualiza como el nodo de inicio y la segunda con el peso de dicha arista. De lo contrario, el peso se inicializa en **INF**, el cual corresponde al máximo entero permitido por Java, y el nodo conectado se inicializa en -1.

Luego, empieza el funcionamiento de Prim, generando un ciclo while que se repite hasta que se hayan visitado todos los nodos. En cada iteración, se busca la arista de menor peso, para lo cual utilizamos la estructura de **distancia**. En esta iteramos sobre las filas, que corresponden a nodos y, para aquellas que no estén ya en el conjunto de nodos visitados, revisamos si su peso es menor al peso mínimo actual. En este caso, se actualiza el nodo de origen (el que ya está en el MST, primera columna de la estructura), el nodo de destino (el que se agrega al MST, la fila de la estructura) y el peso.

Al terminar el ciclo de búsqueda de la arista, se crea la lista que la va a guardar y se añaden los elementos en el orden correspondiente. Luego, se añade la arista a la lista de aristas incluidas y el nodo de desitno al conjunto de nodos visitados. Por último pero, crucialmente, se actualiza **distancia**. Esto resulta sencillo pues, dado que añadimos un nodo a la vez al MST, basta con revisar para cada nodo que todavía no haya sido visitado si tiene una conexión que el nodo recién agregado y, si la tiene,

si esta tiene un menor peso que su menor distancia actual al MST, guardada en **distancia**. De esta operación se encarga la función **actualizarDistancia**.

ii) **Complejidad Temporal**

Basado en la explicación anterior vemos que la principal complejidad temporal del algoritmo radica en los dos ciclos anidados, el primero para añadir uno a uno los nodos al MST y, en cada paso, el segundo para encontrar la arista de menor peso entre el MST y un nodo no en este. Como se explicó, encontrar la arista de menor peso usando la estructura **distancia** tiene una complejidad de $O(n)$ (recordar que n es el número de nodos), pues se hace un recorrido sobre las n filas de la estructura. Ahora bien, esta acción se repite $n-1$ veces, dado que este es el número de nodos que se deben añadir al conjunto de nodos visitados antes de que el algoritmo pare. Por ende, el algoritmo realizaría en el orden de $n^2 - n$ operaciones para encontrar el MST, lo cual en notación Big Oh asintótica se traduce en una complejidad

$$T(n) = O(n^2)$$

2.2 Algoritmo de Kruskal

Este algoritmo fue implementado en Java y acá se plasma el GCL correspondiente. Sigue el funcionamiento general de Kruskal, en el cual se ordenan las aristas del grafo en orden creciente de peso y se va escogiendo la de menor peso que no genere un ciclo. Siguiendo el enunciado, el grafo a usar está implementado usando listas de adyacencias. Para esto, se generó una clase auxiliar llamada **Arista**, la cual guarda los dos nodos y el peso de una arista. Por ende, la estructura del grafo es un HashMap donde cada llave es un nodo (cada nodo es un número entre 0 y $n-1$, con n siendo el número total de nodos) y para cada uno su valor es una lista de objetos **Arista**.

```
fun algoritmoKruskal(grafo: Map<int, List<Arista>>) ret aristasIncluidas: List<Arista>
    var aristasOrdenadas: List<Arista>;
    aristasOrdenadas := ordenarAristasPeso(grafo);
    var n: int;
    n := grafo.size();
    var aristasIncluidas: List<Arista>;
    var nodoRaiz, numHijos: array[0,n) of int;
    for (int i := 0; i < n; i++) →
            numHijos[i] := 0;
            nodoRaiz[i] := i;
    rof
    var indArista: int;
    indArista := 0;
    do aristasIncluidas.size() < n - 1 →
            var arista: Arista;
            arista := aristasOrdenadas.get(indArista);
            var node1, node2, raiz1, raiz2: int;
            node1 := arista.nodoOrigen;
            node2 := arista.nodoDestino;
            raiz1 := find(nodoRaiz, node1);
            raiz2 := find(nodoRaiz, node2);
            if raiz1 ≠ raiz2 →
                    union(nodoRaiz, numHijos, raiz1, raiz2);
                    aristasIncluidas.add(arista);
            fi
            indArista := indArista + 1;
    od
ret aristasIncluidas
```

```
fun find(parent: array[0,n) of int, nodoEncontrar: int) ret nodoRaiz: int
    var raizActual: int;
    raizActual := parent[nodoEncontrar];
    do raizActual ≠ nodoEncontrar →
            nodoEncontrar := raizActual;
            raizActual := parent[nodoEncontrar];
    od
    nodoRaiz := nodoEncontrar;
ret nodoRaiz

fun union(parent: array[0,n) of int, rank: array[0,n) of int, nodo1: int, nodo2: int) ret void
    var hijos1, hijos2: int;
    hijos1 := rank[nodo1];
    hijos2 := rank[nodo2];
    if hijos1 ≥ hijos2 →
            parent[nodo2] := nodo1;
            rank[nodo1] := hijos1 + 1;
    else →
            parent[nodo1] := nodo2;
            rank[nodo2] := hijos2 + 1;
    fi

fun ordenarAristasPeso(grafo: Map<int, List<Arista>>) ret listaCompleta: List<Arista>
    var aristasUnicas: Set<Arista>;
    aristasUnicas := {};
    for (listaAdyacencia: List<Arista> : grafo.values()) →
            aristasUnicas.addAll(listaAdyacencia);
    rof
    var listaCompleta: List<Arista>;
    listaCompleta := List(aristasUnicas);
ret listaCompleta

class Arista implements Comparable<Arista>
    var nodoOrigen, nodoDestino, peso: int;

        fun Arista(nodoOrigen: int, nodoDestino: int, peso: int)
            this.nodoOrigen := nodoOrigen;
            this.nodoDestino := nodoDestino;
            this.peso := peso;

        fun compareTo(otraArista: Arista) ret comparacion: int
            comparacion := Integer.compare(this.peso, otraArista.peso);
            ret comparacion
```

i) **Explicación implementación**

Para implementar Kruskal me basé en las explicaciones de los siguientes videos sobre el uso del algoritmo **Union-Find** para verificar de manera eficiente que una arista no genere un ciclo en el MST que va generando Kruskal, principal problema a resolver al implementar el algoritmo:

(a) Kruskal algorithm implementation - Techdose

(b) Union Find Kruskal's Algorithm - WilliamFiset

(c) Union Find in 5 minutes — Data Structures & Algorithms - Potato Coders

En esencia, al usar **Union-Find** lo que hacemos es asignar los nodos a conjuntos y, si queremos incluir una arista revisamos los conjuntos a los que pertenecen los nodos. Si pertenecen al mismo conjunto no se añade la arista, pues generaría un ciclo. De lo contrario, se añade la arista y se combinan los conjuntos. Crucialmente, los nodos del conjunto más pequeño se añaden a los del más grande.

Una forma eficiente de implementar **Union-Find** es mediante dos arreglos de tamaño n (número de nodos), que en mi código llamo **nodoRaiz** y **numHijos**, los cuales nos permiten organizar fácilmente los nodos en conjuntos mediante nodos representantes. Utilizar nodos representantes nos ahorra tener que guardar y combinar conjuntos completos y, en cambio, tener una representación reducida del MST a medida que se va generando. Por un lado, **nodoRaiz** guarda para cada nodo su nodo padre o raíz, que puede ser el mismo nodo (así inicia) u otro nodo. Con esto debemos introducir el concepto de nodo raíz absoluto. Esto pues, podemos tener un nodo cuyo nodo raíz tenga una raíz diferente a él mismo. Un nodo raíz absoluto surge de iterar sobre las raíces hasta llegar a un nodo cuya raíz es el mismo. De esta manera, nuestros nodos representantes para los conjuntos serán estos nodos raíz absolutos. Por otro lado, **numHijos** guarda el número de hijos directos con los que cuenta un nodo.

Para mostrar cómo funciona esta estructura y algoritmo tenemos el siguiente ejemplo. Suponga un grafo con $n = 6$. **numHijos** se inicializa en 0, pues ningún nodo tiene hijos, mientras que cada posición de **nodoRaiz** se inicializa en ella misma, indicando que cada nodo es su propio nodo raíz/representante y que todos pertenecen a conjuntos distintos. Ahora suponga que la arista de menor peso del grafo conecta a 1 y 5. Primero, revisamos que no tengan el mismo nodo raíz absoluto, para lo cual utilizamos la operación **find**, definida en el GCL anterior. Dado que es la primera arista, 1 y 5 son sus propios representantes, por lo que se puede incluir la conexión. Al incluir la arista, ahora 1 y 5 deberían hacer parte del mismo grupo. Aquí utilizamos la estructura **numHijos** para saber los nodos hijos tanto de 1 y de 5. De manera general, haremos que el nodo con menor número de hijos termine apuntando al nodo con mayor número de hijos. Dado que hay un empate, no importa como lo asignemos, así que podemos decir que 5 apunta a 1 y nos aseguramos de sumar 1 al número de hijos del nodo 1. Esto es de lo que se encarga la operación **union**, definida en el GCL. Tras estas operaciones los arreglos quedan de la siguiente manera

$$\text{nodoRaiz} = [\ 0, 1, 2, 3, 4, 1\ ]$$

$$\text{numHijos} = [\ 0, 1, 0, 0, 0, 0\ ]$$

Ahora, el nodo 5 apunta a 1 como su raíz. Suponga ahora que queremos añadir una arista entre 3 y 4. Repitiendo el proceso anterior y escogiendo que 4 apunte a 3 tenemos que

$$\text{nodoRaiz} = [\ 0, 1, 2, 3, 3, 1\ ]$$

$$\text{numHijos} = [\ 0, 1, 0, 1, 0, 0\ ]$$

Ahora añadimos una arista entre 0 y 5. Al ejecutar el algoritmo nos damos cuenta que la raíz absoluta de 5 es 1, mientras que la de 0 es el mismo, por lo que se pueden unir. En este caso, se puede tomar la decisión de si asociar 0 a 1 o a 5, lo cual cambiará la complejidad. Crucialmente, siempre asociar a la raíz absoluta, en este caso 1, reduce al mínimo el número de pasos entre un nodo y su raíz absoluta, optimizando la operación de **find**, por lo que ese fue el acercamiento que tomé. Esto se evidencia en la función **algoritmoKruskal** que explicaré más adelante. Con esto los arreglos quedan de la siguiente manera:

$$\text{nodoRaiz} = [\ 1, 1, 2, 3, 3, 1\ ]$$

$$\text{numHijos} = [\ 0, 2, 0, 1, 0, 0\ ]$$

Ahora, como último ejemplo, suponga que quisiéramos añadir una arista entre 4 y 1. La raíz absoluta de 4 es 3, con un hijo, mientras que la de 1 es el mismo con 2 hijos; por ende, podemos agregar la arista y 3 apuntará a 1 como su raíz tal que los arreglos queden de la siguiente forma

$$\text{nodoRaiz} = [\ 1, 1, 2, 1, 3, 1\ ]$$

$$\text{numHijos} = [\ 0, 2, 0, 2, 0, 0\ ]$$

Este es el funcionamiento del algoritmo **Union-Find** y es muy útil para llevar registro de los distintos grupos a medida que Kruskal va generando el MST con el objetivo de no generar ciclos.

Antes de pasar a la implementación de Kruskal, explicaré la implementación de **find** y **union**. En el caso de **find**, esta función recibe el arreglo de nodos raíces y el nodo para el cual se quiere encontrar su raíz absoluta **nodoEncontrar**. Con esto, se accede a la raíz de **nodoEncontrar** y, mientras esta sea diferente a **nodoEncontrar** reemplazamos **nodoEncontrar** por su raíz y actualizamos la raíz a la del nuevo **nodoEncontrar**. Por otro lado, **union** recibe los dos arreglos y los dos nodos que se quieren unir. La función primero accede al número de hijos de cada nodo y luego los compara. El nodo que tenga mayor número de hijos se actualiza como la raíz del otro y se le suma un hijo a su entrada en el arreglo correspondiente. En caso de empate, se decidió arbitraramiente que se debe asociar el segundo nodo al primero, como si este último hubiera tenido más hijos; esto no afecta el desarrollo del algoritmo en ejecución.

Habiendo entendido el rol de **Union-Find** en la implementación, pasamos a la explicación de esta. El algoritmo empieza generando la lista de aristas ordenadas por peso de manera creciente. Para esto, se implementó la función auxiliar **ordenarAristasPeso**, la cual recibe la estructura del grafo. Para facilitar el ordenamiento, en la construcción del grafo si, por ejemplo, recibimos una arista entre los nodos 1 y 2, generamos un único objeto **Arista**, el cual agregamos a las listas de adyacencia de tanto 1 como 2. De esta manera, la función genera un conjunto de objetos **Arista**. Dado que en un conjunto no hay elementos repetidos, la función puede recorrer todas las listas de adyacencia y agregar indiscriminadamente todos los elementos de cada una al conjunto. Esto pues, si una arista ya se encontraba en el conjunto y se vuelve a añadir, lo cual con toda seguridad pasará para todas las aristas pues el grafo es no dirigido, simplemente no se añade, pues es un conjunto y no admite duplicados. Por último, la función genera una lista, implementada como ArrayList, a partir de dicho conjunto y lo ordena utilizando el método nativo de List.sort() en Java[1].

Con las aristas ordenadas, el algoritmo genera todas las estructuras que necesita: accede al tamaño del grafo; genera una lista de aristas incluidas en el MST para el retorno; genera e inicializa correctamente los arreglos para la implementación de **Union-Find**. Con esto listo, empieza a recorrer la lista de aristas hasta que se hayan incluido n-1 en la lista de aristas de retorno. En cada paso, accede a los dos nodos de la arista y, para cada uno, ejecuta la operación **find**. En caso de que las raíces de los nodos sean diferentes, invoca la operación **union** entre las raíces de los nodos[2] y añade la arista a la lista de retorno.

ii) **Complejidad Temporal**

Bajo esta implementación, la complejidad temporal se encuentra medida por dos aspectos: el ordenamiento de la lista de aristas y la ejecución de la operación **find** en cada paso del ciclo de adición de aristas. En el ordenamiento, se utiliza el método nativo de List.sort() del Java, el cual trabaja sobre una modificación de TimSort, que a su vez es un algoritmo híbrido entre **mergeSort** e **insertionSort** y que tiene una complejidad temporal para n elementos de

$$T(n) = n \log n$$

En este caso, tendríamos una complejidad

$$T(m) = m \log m$$

[1]Nota. Para que esto funcione, se hizo un *Override* sobre el método **compareTo** para la clase **Arista**, de tal forma que Java pudiera comparar aristas mediante su atributo de peso. Esto se puede ver en la definción de la clase en el GCL.

[2]Aquí es donde tomamos la decisión descrita anteriormente sobre cómo actualizar los nodos raíces en la unión de dos conjuntos. Esto es lo que nos permite acortar al máximo los saltos entre nodo y su raíz absoluta a medida que el árbol va creciendo, manteniendo a la operación de **find** en tiempo casi constante.

donde m es el número de aristas del grafo.

Por otro lado las operaciones del algoritmo **Union-Find** se ejecutan cada una, a lo mucho, en el orden de $n-1$ veces. Ahora bien, por la explicación anterior sabemos que **union** trabaja en tiempo constante, pues únicamente hace accesos y modificaciones a arreglos y no recorridos. En cambio, **find** no necesariamente es constante, pues sí genera un recorrido sobre el arreglo y, al final, la complejidad estará dada por la cantidad de saltos promedio entre un nodo y su nodo raíz absoluto. Sin embargo, las decisiones de implementación aseguran que este tiempo de ejecución sea mínimo y, en general, se pueda tomar casi como constante, como subraya esta publicación en Stack Overflow.

Para visualizar esto miremos como crecería la distancia entre un nodo y su nodo raíz absoluto. Supongamos que inicialmente unimos dos nodos 1 y 2, y que 2 apunta a 1. Luego de hacer esto, cualquier otro nodo que queramos conectar con 1 o 2 encuentra el nodo raíz 1 en una única operación, pues tanto 1 como 2 apuntan a 1 en el arreglo, y lo reemplazan por su actual nodo raíz. Sucesivamente, cualquier nodo que queramos añadir a un nodo que ya fue "apadrinado" por 1 encuentra el nodo raíz 1 en una única iteración, pues estos hijos ya van a haber estado apuntando a 1. La única forma en la que los hijos de 1 van a tener que hacer más de una iteración para encontrar su nodo raíz es si el nodo 1 mismo es unido a otro nodo, por ejemplo 3, con más hijos. En ese caso, 1 apunta a 3 y todos los hijos de 1 siguen apuntando a 1, por lo que tendrían que efectuar dos saltos en el arreglo para encontrar su nodo raíz absoluto. Aún con esto, vemos que el aumento es mínimo y que, a medida que avance el algoritmo estos eventos se hacen menos probables, pues es más probable que un nuevo nodo se añada a un nodo representante ya existente a que genere su propio subconjunto que eventualmente termine siendo más grande que otros. Por ende, la complejidad temporal de **find**, si es que no es constante bajo la implementación actual, por lo menos es muy pequeña a medida que crece el número de nodos.

Con esto, tendríamos que la complejidad del algoritmo estaría descrita por la ecuación[3] (obviamos **union** porque seguramente se ve dominada, por lo menos, por el ordenamiento de la lista, al solo tener una complejidad total en el orden de $O(n-1)$)

$$T(n,m) = m\log m + \alpha(n)\cdot(n-1)$$

donde α representa la complejidad de **find**. Por nuestro análisis, sabemos que esto es muy bajo, casi comparable con algo constante, por lo que podemos reescribir la complejidad como

$$T(n,m) = m\log m + (n-1)$$

por complejidad asintótica, tendríamos que

$$T(m) = m\log m$$

lo cual indica que el ordenamiento de las aristas corresponde a la operación más costosa temporalmente del algoritmo.

2.3 Generación de Casos de Prueba y Salida Algoritmos

Dado que ambos algoritmos aceptan los mismos grafos con las mismas características, para la generación de casos de prueba para el algoritmo se desarrolló un único programa que construye N grafos (establecido por defecto como 100) no dirigidos, conectados y con un único MST. Dicho programa es semialeatorio en cuanto al número de nodos, aristas[4], pesos de las aristas y combinación de conexiones. El programa está implementado en Java y corresponde a **GeneracionCasosPruebaP2.java**. Este cuenta con variables locales que permiten cambiar los parámetros de generación: número de casos, mínimo y máximo número de nodos, máximo peso de las aristas, y máxima cantidad de aristas adicionales.

[3] Tomo notación prestada de la publicación referenciada anteriormente para definir el tiempo de ejecución de **find**

[4] Se genera aleatoriamente el número de aristas adicionales que tiene cada grafo, teniendo en cuenta el máximo total que puede tener un grafo no dirigido, y las aristas que ya se usaron para que sea conectado

Los grafos generados por el programa se escriben a un archivo de texto .IN, donde la primera línea tiene el número de casos de prueba y cada línea subsecuente es un grafo. Cada línea se encuentra estructurada de la siguiente manera

$$\text{n} \quad \text{initialNode} \quad \text{node1, node2, weight} \quad \text{node3, node4, weight} \quad ...$$

donde n denota el número de nodos del grafo, initialNode el nodo por el que debe empezar el algoritmo (solo se usa en Prim) y cada arista se encuentra caracterizada por dos nodos y un peso, separados por comas entre sí. El peso de las aristas siempre es el tercer elemento de cada entrada. Cada arista se separa de las otras y de n por un espacio. Por último, cada nodo es un número entre 0 y n - 1, incluyendo ambos límites. Este es el formato que lee el archivo del algoritmo.

Usando este archivo para una ejecución del algoritmo a través de entrada y salida estándar, el retorno de este consiste de un archivo .OUT donde cada línea es la respuesta a la línea correspondiente en el archivo de entrada. Cada línea empieza por el mínimo peso del MST generado por el algoritmo, seguido de las aristas que componen dicho árbol estructuradas en el mismo formato que en el archivo de entrada.

Por último, el algoritmo de generación de casos de prueba también genera un archivo .txt, actualmente llamado respuestaEsperadaMST.txt, con lo que se espera que devuelvan los algoritmos al ejecutarse. Tanto el .IN como el .txt se generan al tiempo al ejecutar el programa.

3 Problema 3 - Grafos Bipartitos

Para detectar un grafo conexo[5] bipartito implementé en un algoritmo en Java, con el correspondiente GCL a continuación. Este algoritmo se basa en BFS para hacer una búsqueda por niveles, en la cual se van añadiendo los nodos a uno de dos conjuntos dependiendo del conjunto al que se añadió el nodo por el cual fueron descubiertos. El grafo se implementa como un **HashMap** cuyas llaves son los nodos (0 a $n-1$) y sus valores son listas de objetos **Arista**[6].

```
fun esBipartito(grafo: Map<int, List<Arista>>) ret retorno: List<Set<int>>
    var nodosRojos, nodosAzules, descubiertos, procesados: Set<int>;
    var cola: Queue<int>;
    var nodoOrigen: int;
    nodoOrigen := 0;
    var dobleColor: bool;
    dobleColor := false;
    cola.offer(nodoOrigen);
    descubiertos.add(nodoOrigen);
    nodosRojos.add(nodoOrigen);
    do (cola.size() > 0 and dobleColor = false) →
        var nodoActual, colorOrigen: int;
        nodoActual := cola.poll();
        if nodosRojos.contains(nodoActual) →
            colorOrigen := 1;
        else →
            colorOrigen := 0;
        fi
        procesados.add(nodoActual);
        var listaAdyacencia: List<Arista>;
        listaAdyacencia := grafo.get(nodoActual);
        for (arista: listaAdyacencia) →
```

[5] Tras aclararlo con un monitor, se estableció que los grafos a evaluar debían ser, por lo menos, conectados

[6] La implementación de esta clase es reutilizada de la generada para Kruskal. Únicamente cambia que en vez de que cada arista guarde ambos nodos, solo guarda el nodo de destino, siendo el nodo de origen la llave en el HashMap que lleva a la lista de la arista.

```
                var nodoDestino: int;
                nodoDestino := arista.getNodoConectado();
                if colorOrigen = 1 →
                        if nodosRojos.contains(nodoDestino) →
                                dobleColor := true;
                        [ ] not nodosRojos.contains(nodoDestino) →
                                if not descubiertos.contains(nodoDestino) →
                                        cola.add(nodoDestino);
                                        descubiertos.add(nodoDestino);
                                        nodosAzules.add(nodoDestino);
                                fi
                        fi
                else →
                        if nodosAzules.contains(nodoDestino) →
                                dobleColor := true;
                        [ ] not nodosAzules.contains(nodoDestino) →
                                if not descubiertos.contains(nodoDestino) →
                                        cola.add(nodoDestino);
                                        descubiertos.add(nodoDestino);
                                        nodosRojos.add(nodoDestino);
                                fi
                        fi
                fi
        rof
    od
    var retorno: List<Set<int>>;
    if not dobleColor →
            retorno.add(nodosAzules);
            retorno.add(nodosRojos);
    fi
ret retorno
```

1. **Explicación implementación**

   Como se mencionó anteriormente, el código es realmente una extensión de BFS que va revisando que los nodos se puedan dividir adecuadamente en los dos conjuntos disjuntos que definen a un grafo bipartito. Para esto, se crean cuatro conjuntos, dos naturales a BFS, **procesados** y **descubiertos**, y dos adicionales, **nodosRojos** y **nodosAzules**. Como sus nombres lo indican, estos últimos dos guardarán los nodos "coloreados" con el color respectivo[7]. Dado que trabajamos con grafos conectados, podemos empezar desde cualquier nodo nuestro BFS; arbitrariamente empezamos en 0. Además, generamos un centinela que nos permitirá terminar el ciclo tempranamente si detectamos que hay un nodo que fue coloreado con colores distintos y que servirá para saber el resultado final para el retorno. Por último, generamos la cola para almacenar los nodos descubiertos pero no procesados, otra estructura canónica de BFS.

   Una vez tenemos todo listo, añadimos el nodo de inicio a la cola y al conjunto de nodos rojos (no importa si es a este o al de azules realmente) y empezamos a aplicar BFS. En cada paso, asignamos **nodoActual** como el resultado de hacer **pop** sobre la cola y, utilizando operaciones de conjuntos determinamos su color: 1 para rojos, 0 para azules. Añadimos el nodo actual a **procesados** y accedemos a su lista de adyacencia. Luego, recorremos los objetos **Arista** de dicha lista de adyacencia y para cada arista revisamos que el nodo asociado a la arista sea de un color distinto del nodo actual: Si el **colorOrigen** es 1, el nodo asociado debería ser azul y, por ende, revisamos si está en los nodos

[7] Recordar que un grafo bipartito es aquel que se puede colorear con 2 colores de manera que ningún par de nodos conectados entre sí son del mismo color

rojos. En caso de estar, esto implica que el grafo no es bipartito, por lo que actualizamos nuestro centinela. Si no está, puede ser porque está en los azules o no ha sido coloreado; en ambos casos sigue siendo bipartito y procedemos a colorear el nodo, añadiéndolo al conjunto de nodos azules. Además, mantenemos las operaciones normales de BFS, añadiendo el nodo asociado a la arista a **descubiertos** y a la cola. El proceso inverso y análogo se realiza si el color de origen es 0 (azul).

Esto se repite hasta que no haya elementos en la cola (se visitaron todos los nodos y es bipartito) o cuando el centinela se active y nos indique que detectó un nodo con dos colores distintos. Luego, generamos una variable de retorno, que será una lista de conjuntos. Si el centinela no está activo (su valor es **false**) añadimos a esta lista ambos conjuntos de nodos. En otro caso, no agregamos nada. Cada caso es manejado apropiadamente por la función de ejecución

2. **Complejidad Temporal**

   Dado que se trata de una implementación de BFS normal y cuyos únicos cambios implican operaciones de conjuntos que, al usar **HashSet** se pueden hacer en $O(1)$, la complejidad del algoritmo es la misma con respecto a un BFS tradicional. Incluso, en caso de no ser bipartito, el algoritmo para antes de recorrer y procesar todos los nodos, ahorrando tiempo. Aún así, el peor caso sigue siendo recorrer todos los nodos, lo cual ocurre si el grafo es, en efecto, bipartito. De esta manera, la complejidad temporal del algoritmo se ve caracterizada, de manera asintótica, por

$$T(n, m) = O(n + m)$$

   lo cual corresponde a la complejidad de un algoritmo de BFS tradicional.

3.1 Generación de Casos de Prueba y Salida Algoritmo

Para la generación de casos de prueba para el algoritmo se desarrolló un programa que construye, de N casos de prueba totales, la mitad con grafos bipartitos y la otra mitad con grafos no bipartitos. Dicho programa es semialeatorio en cuanto al número de nodos, aristas[8], pesos de las aristas y combinación de conexiones. El programa está implementado en Java y corresponde a **GeneraciónPruebasBipartitos.java**. Este cuenta con variables locales que permiten cambiar los parámetros de generación: número de casos, mínimo y máximo número de nodos, máximo peso de las aristas, y máxima cantidad de aristas adicionales.

Los grafos generados por el programa se escriben a un archivo de texto .IN, donde la primera línea tiene el número de casos de prueba y cada línea subsecuente es un grafo. Cada línea se encuentra estructurada de la siguiente manera

$$\text{n} \quad \text{node1}, \text{node2}, \text{weight} \quad \text{node3}, \text{node4}, \text{weight} \quad ...$$

donde n denota el número de nodos del grafo y cada arista se encuentra caracterizada por dos nodos y un peso, separados por comas entre sí. El peso de las aristas siempre es el tercer elemento de cada entrada. Cada arista se separa de las otras y de n por un espacio. Por último, cada nodo es un número entre 0 y n-1, incluyendo ambos límites. Este es el formato que lee el archivo del algoritmo.

Usando este archivo para una ejecución del algoritmo a través de entrada y salida estándar, el retorno de este consiste de un archivo .OUT donde cada línea es la respuesta a la línea correspondiente en el archivo de entrada. Si el grafo es bipartito, se imprime True seguido de los conjuntos de nodos disjuntos, cada uno encapsulado por corchetes y con los nodos separados por comas. Si el grafo no es bipartito, simplemente imprime False.

Por último, el algoritmo de generación de casos de prueba también genera un archivo .txt, actualmente llamado respuestaEsperadaBipartitos.txt, con lo que se espera que devuelva el algoritmo al ejecutarse. Tanto el .IN como el .txt se generan al tiempo al ejecutar el programa.

[8]Se genera aleatoriamente el número de aristas adicionales que tiene cada grafo, teniendo en cuenta el máximo total que puede tener un grafo bipartito y uno no bipartito, y las aristas que ya se usaron para que cualquiera de los dos sea conectado

4 Referencias

A continuación se encuentran todas las referencias utilizadas como apoyo en el desarrollo de esta tarea:

1. https://www.baeldung.com/java-comparator-comparable
2. https://stackoverflow.com/questions/13132548/time-complexity-of-prims-mst-algorithm
3. https://stackoverflow.com/questions/32040718/does-using-union-find-in-kruskals-algorithm-actually-affect-worst-case-runtime
4. https://stackoverflow.com/questions/13132548/time-complexity-of-prims-mst-algorithm
5. https://www.youtube.com/watch?v=JZBQLXgSGfs
6. https://www.youtube.com/watch?v=Ub-fJ-KoBQM&t=612s
7. https://www.youtube.com/watch?v=ayW5B2W9hfo