1. descripción de las estructuras de datos implementadas.

Arista: Utilizada para guardar las aristas del grafo para luego ordenarlas

Lista: Utilizada en la implementación de listas de union find en ella se almacena el representante de un conjunto, además de la cantidad de vértices. Tiene dos punteros a la lista que contiene los vértices que son parte del conjunto , uno que apunta al inicio de la lista y otro a la cola de la lista.

Nodo:Representa a un nodo siendo la componente elemental de la lista que contiene el conjunto de vértices y este tiene un puntero a su representante y otro al siguiente nodo en la lista .

2. Pseudocódigo de todas las operaciones implementadas.

2.1. Implementación con listas

2.1.1. MakeSet()

```
lista *set ← new lista();
set− >representante ← x;
*repre ← new nodo();
repre− >vertice ← x;
repre− >next ← NULL;
repre− >representadopor ← set;
set− >siguiente ← repre;
set− >cola ← repre;
set− >cantidadvertices ← 1;
conjuntos.insert(makepair ( x , set));
```

2.1.2. Find()

```
return conjuntos.find(X)− >second;
```

2.1.3. Union()

```
if Conjunto1− >cantidadvertices > Conjunto2− >cantidadvertices then
    iterador ← Conjunto2− >siguiente;
    delete(iterador− >representadopor);
    while iterador do
        conjuntos.erase(iterador− >vertice);
        conjuntos.insert(makepair(iterador− >vertice , Conjunto1));
         Conjunto1-¿cola ← iterador;
        iterador− >representadopor ← Conjunto1;
        Conjunto1− >cantidadvertices++;
        iterador ← iterador− >next;
    end while
else
    iterador ← Conjunto1− >siguiente;
    delete(iterador− >representadopor);
    Conjunto2− >cola− >next ← iterador;
    while iterador do
        conjuntos.erase(iterador-¿vertice);
        conjuntos.insert(makepair(iterador− >vertice , Conjunto2));
        iterador− >representadopor ← Conjunto2;
        Conjunto2− >cola ← iterador;
        Conjunto2− >cantidadvertices++;
        iterador ← iterador− >next;
    end while
end if
```

2.2. Implementación con arboles

2.2.1. MakeSet

Padres[x] ← x;
Rank[x] ← 0;

2.2.2. Find

if x ! = Padres[x] **then**
    Padres[x] ← this− >Find(Padres[x]);
end if
return Padres[x];

2.2.3. Link

if Rank[x] > Rank[y] **then**
    Padres[y] ← x;
elsePadres[x] ← y;
    if (Rank[x] == Rank[y] **then**
        Rank[y] ← (Rank[x])+1;
    end if
end if

2.2.4. Union

link(this− >Find (X) ,this− >Find(Y));

3. Análisis de complejidad y amortizado

3.1. Implementación con listas

- Complejidad MakeSet = O(1)
- Complejidad Find = O(1)
- Complejidad Union = $O(n^2)$
- Amortizado Union = $O(n)$
- Amortizado Union con heurística = O(logn)
- Amortizado MakeSet con heurística = O(1)
- Amortizado Find con heurística = O(1)

En cuanto al análisis amortizado de union es lineal dado que se llaman a funciones n veces, por lo tanto al unir el algoritmo toma tiempo O(n^2), por ejemplo:
crear(x_1) = 1
crear(x_q) = 1
unir(x_1,x_2) = 1
unir(x_2,x_3) = 2 ...
unir($x_q - 1$,x_q) = q-1.

Es una secuencia de m = n+q-1 operaciones requiriendo un tiempo de O(n + q^2) = $\theta(m^2)$ que en tiempo amortizado es $\Theta(m)$.

Heurística de union: En el caso que tomemos una heurística de optar por añadir siempre la lista más corta al final de la más larga, el análisis se reduce a O(m + nlog n).

3.2. Implementación con arboles

Complejidad MakeSet = O(m α(n))
Complejidad Find = O(m α(n))
Complejidad Union = O(m α(n))

Amortizado MakeSet = O(1)
Amortizado Find = O(α(n))
Amortizado Union = O(α(n))
Amortizado Find = O(α(n))

4. Demuestre que incluir aristas de menor a mayor peso es una opción greedy que permite encontrar un MST mínimo.

El algoritmo greedy es utilizado para maximizar como para minimizar dependiendo el objetivo, para este caso greedy lo que hará es a medida que se incluyan las aristas va a tomar aristas con el menor costo generando el mst, en efecto,

Por reducción al absurdo: Sea k1 la arista de menor peso en A.
Supongamos un MST T' que no incluye a k1.
Consideremos T'∪ k1 con peso (T'∪ k1) = peso(T') + peso(k1).
En T'∪ k1 aparece un ciclo, pero si eliminamos cualquier arista del ciclo (x), distinta de k1, obtenemos un árbol T*=T'+ k1 - x con peso
(T*) = peso(T') + peso(k1) – peso(x).
Por tanto, como peso(k1) < peso(x), deducimos que peso(T*) < peso(T'). Contradicción.

5. Evaluación experimental considerando diversos grafos de entrada

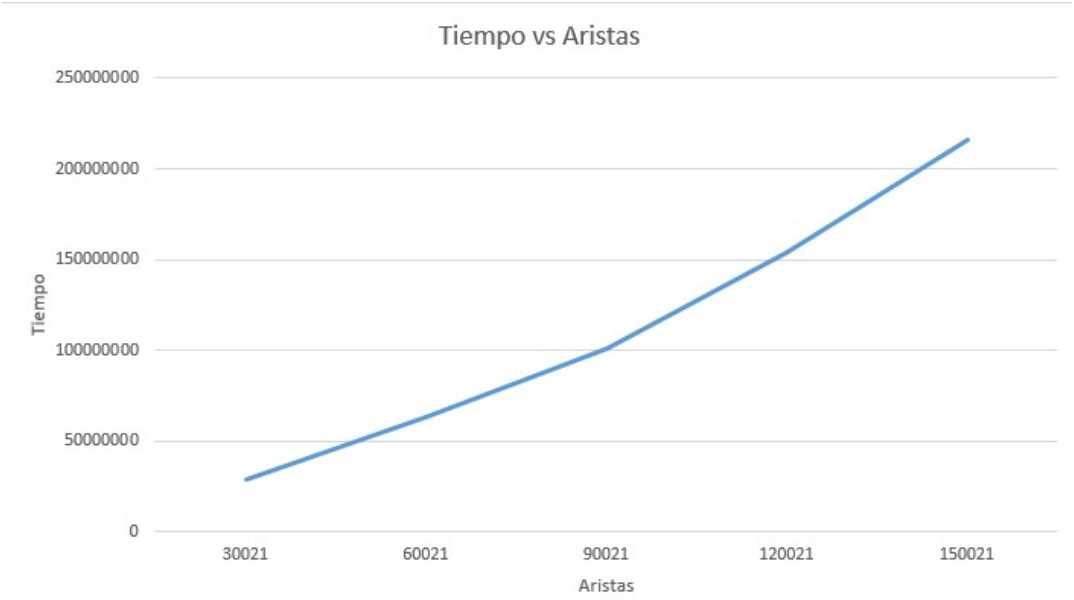

Figura 1: Implementación con lista

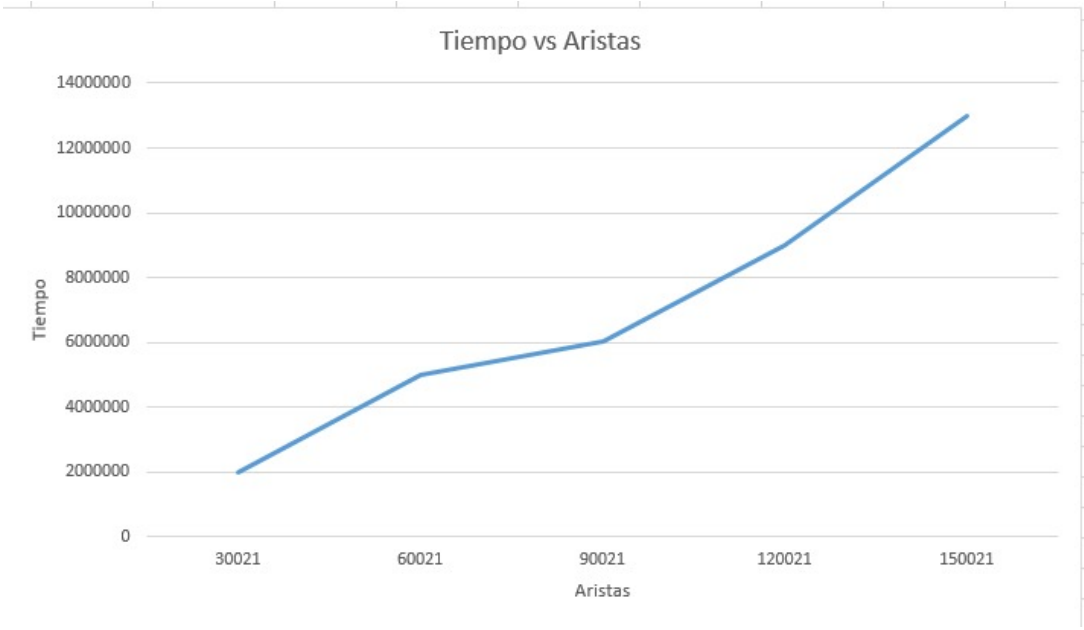

Figura 2: Implementación con Árbol

6. Discusión de los resultados obtenidos

Efectivamente los resultados obtenidos tiende a concordar con el análisis teórico realizado anteriormente. Esta implementación fue realizada considerando la cantidad de aristas.