Algoritmo de Kruskal

Alberto Calvo Madurga

October 2018

Planteamiento del problema

Nos situamos en el contexto de buscar una solución para la conexión de diferentes puntos tales como ciudades, hospitales,estaciones... Querríamos poder conocer la forma de conectar estos puntos con el mínimo recorrido posible. Esto es deseable por razones obvias como reducción de tiempos de recorrido así como de costes en los tramos.

Para la resolución de un problema como el que justo acabamos de escribir contaremos con una notación G={V,A}, es decir convertiremos nuestros puntos en Nodos que están unidos por Aristas formando un Grafo. Estos grafos pueden ser de distintas maneras. En un caso real podríamos suponer que de dos ciudades (dos nodos) que estén conectadas, van a tener un medio de comunicación en ambos sentidos, que extrapolado a la notación de grafos diremos que es un grafo **no dirigido**. Además también entendemos que si queremos optimizar las conexiones entre estas ciudades, entendemos que todas ellas van a estar comunicadas de una manera u otra. A esto se le llama un grafo **conexo**

Estas dos características definen el tipo de grafo sobre el que vamos a trabajar.

Árbol de Expansión

Teniendo un grafo conexo y no dirigido G, un **árbol de expansión** está compuesto por todos los vértices (nodos) y las aristas necesarias para que estén comunicados. Estos árboles cuentan con la restricción de que no puedan existir ciclos y que a menos exista una ruta entre cada par de vértices (aunque no sea directa).

Árbol de Expansión Mínima

Más especificamente tenemos los **árboles de expansión mínima** que son aquellos cuya suma de los pesos de las aristas es mínima. Existen diversos algoritmos que resuelven este problema (Minimum Spanning Tree, MST). Destacan el algoritmo de Prim y de Kruskal. En nuestro caso nos encargaremos de Kruskal

Idea del algoritmo

El algoritmo de Kruskal tiene una idea básica que es la búsqueda de un subconjunto de aristas que conecten todos los vértices de un grafo donde el valor de la suma de todas las aristas sea la mínima.

Este algoritmo es un ejemplo clásico de algoritmo voraz (greedy). Estos algoritmos tienen una estrategia de búsqueda en la que se elige la opción óptima en el instante actual, tomando decisiones con la información del momento sin tener en cuenta efectos futuros.Son algoritmos fáciles de implementar y eficientes, no obstante estos esquemas no garantizan la solución óptima.

Algoritmo de Kruskal

La forma de trabajar del algoritmo de Kruskal es bastante sencilla de entender. Primero ordenamos las aristas de grafo por su peso de menor a mayor. En cada iteración del bucle, se selecciona la arista de menor valor y se unirá siempre y cuando no se forme un ciclo, es decir, siempre y cuando no una dos vértices que ya tuvieran una ruta para comunicarse. En resumen y antes de pasar a detallar la implementación del algoritmo

en si, tenemos las características generales del problema:

- **Problema**: minimizar la suma de los pesos de las aristas que forman un arbol de expansión para un grafo dado.
- **Candidatos**: lista de aristas disponibles que conectan los puntos del grafo.
- **Solución**: conjunto de aristas que conectan todos los puntos y no forman ningún ciclo.
- **Func.selección**: selección de la arista con peso menor que no haya sido seleccionada previamente.
- **Func.factible**: la arista que se añade a la solución no forma ningún ciclo.
- **Func.objetivo**: lista de aristas que forman el MST

Construcciones fundamentales

Clase Grafo

Para comenzar vamos a crear una clase que represente un grafo, cuyos atributos serán el número de nodos, las aristas que los conectan y dos almacenes que guardan información sobre los padres de cada nodo y su rango (para un uso que posteriormente concretaremos).

```python
#Clase para representar un grafo
class Grafo:

    def __init__(self,nodos):
        self.V= nodos #Num de nodos
        self.aris = []

        self.padre =[] #Padre de cada nodo
        self.rango =[]   #Rango de cada nodo
```

Disjoint-Set :Union-Find

Union Find es una estructura de datos que modela una colección de conjuntos disjuntos (disjoint-sets) y esta basado en 2 operaciones:

- ***buscar(x)***: determina a que conjunto pertenece x. La función principal en nuestro caso es determinar si dos vertices pertenecen al mismo conjunto, para poder saber si al incluirlo en la solución se formaría un ciclo o no.
- ***fusionar(x,y)***: une el conjunto al que pertenece x con el conjunto al que pertenece y.

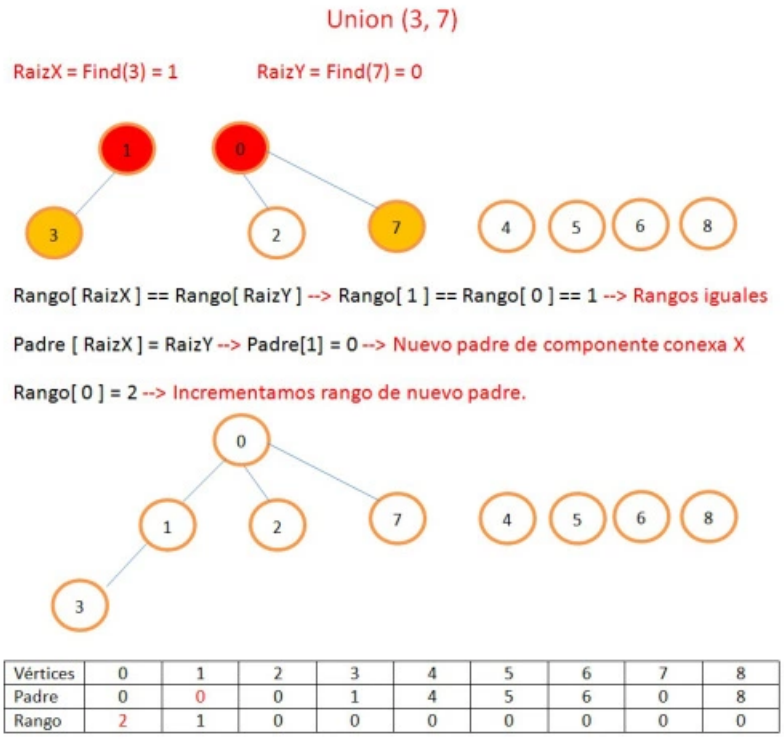
También en este caso he definido un método *iniciarConjuntos()* que nos inicializa los valores para los almacenes *padre* y *rango*.

```
def iniciarConjuntos(self):
    for nodo in range(self.V):
        self.padre.append(nodo)
        self.rango.append(0)
```

Mejora por unión de rango

La idea de esta mejora es unir el árbol de menor rango a la raíz del árbol con el mayor rango, los rangos los almacenamos en el array que lleva dicho nombre y sufrirán modificación al momento de la unión.

Es intuitivo ver como este arreglo soluciona posibles problemas en los que exista una rama del árbol en la que para buscar un nodo concreto tengamos que explorar una cantidad demasiado grande en proporción con el total de nodos.

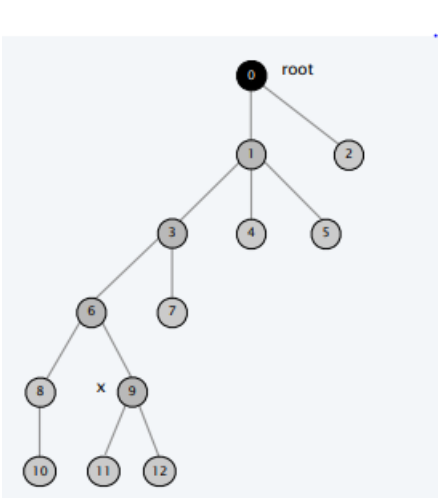


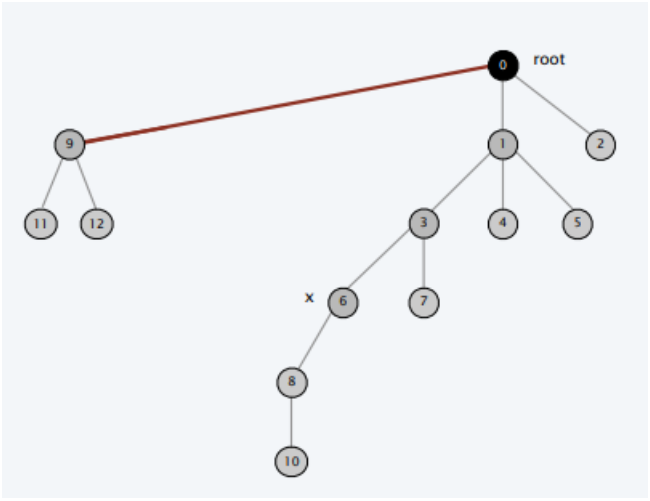
Vértices	0	1	2	3	4	5	6	7	8
Padre	0	0	0	1	4	5	6	0	8
Rango	2	1	0	0	0	0	0	0	0

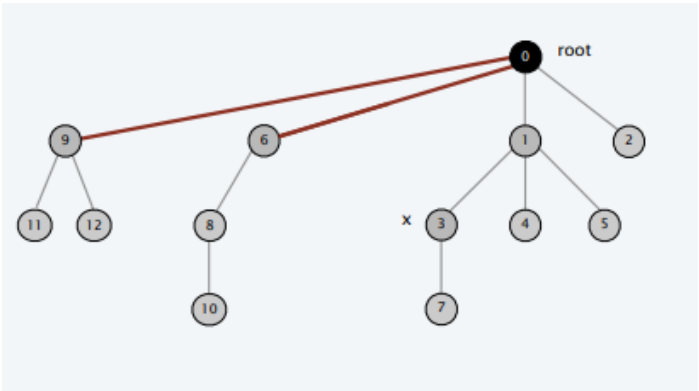
Mejora por compresión de caminos

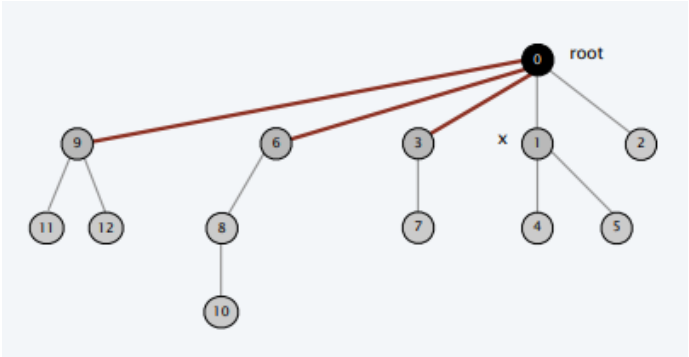
La idea de esta mejora es que cada nodo que visitemos en el camino al nodo raíz puede ser conectado directamente hacia la raíz, es decir al terminar de usar el método Find todos los nodos que visite tendrán como padre la raíz directamente. Esto va a reducir sustancialmente los tiempos de búsqueda del nodo.

Nota: La compresión de caminos no actualiza los rangos

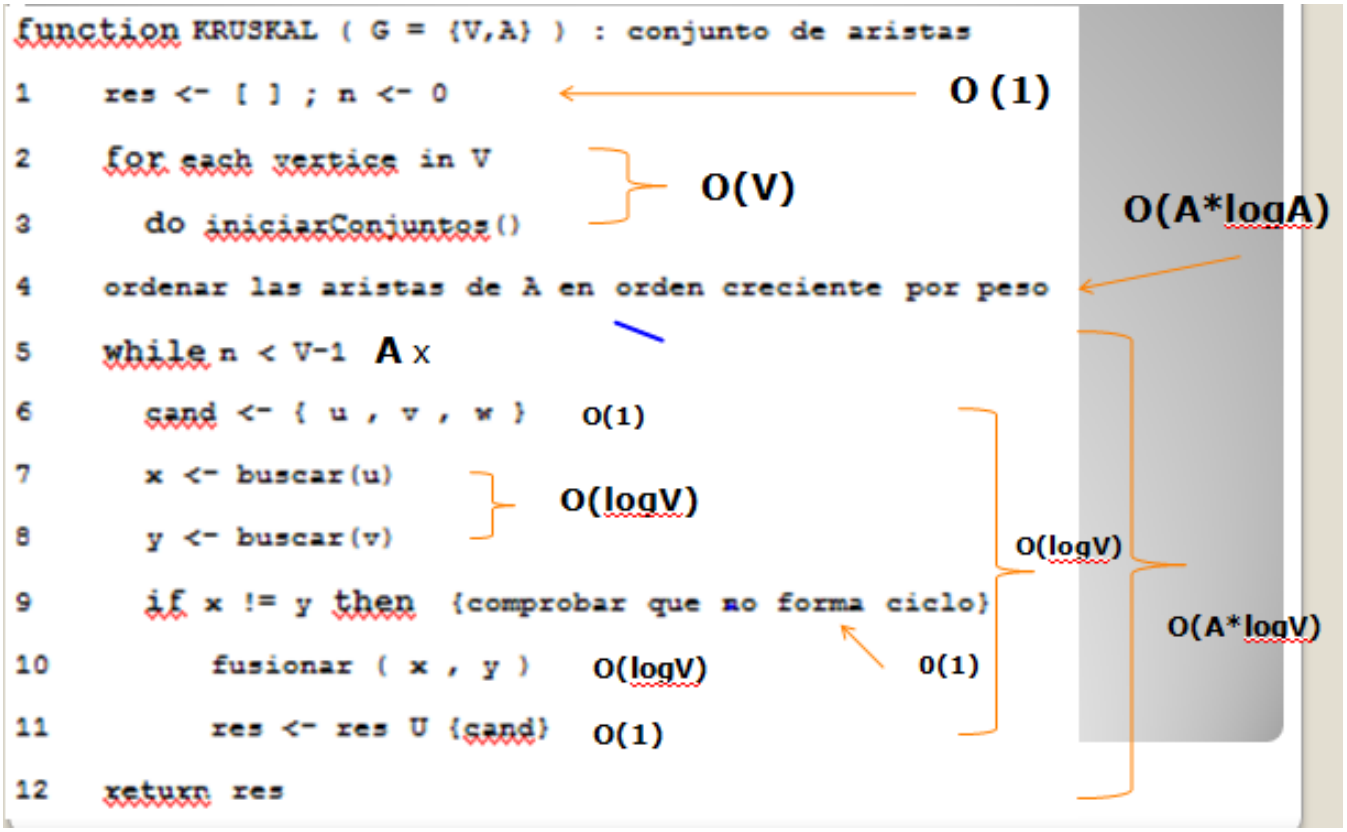








Pseudocódigo



Coste

El tiempo de ejecución del algoritmo depende de la implementación del "disjoint-set" . En este caso hemos implementado la unión por rango y la compresión de caminos, pues es la implementación más rápida asintóticamente conocida.

La inicialización de la línea 1 tiene O(1) y el tiempo de ordenación de los vértices es O(A*log(A)).En las líneas 2-3 tenemos la inicialización de los conjuntos que toma un tiempo de O(V).

El bucle entre las líneas 5 y 11 que es donde se desarrolllas operaciones Union-Find,que toman como mucho O(logV) y este bucle se repite A veces en el peor caso de tener que comprobar todas las aristas,que simplificando la la notación es de órden O(A).Combinando estos dos tenemos O(A*logV) para estas líneas.Hasta ahora tenemos **O(A*logA+A*logV)**

Sabemos que A es como mucho O(V ^2), por lo que O(LogV) y O(LogA) son lo mismo. Esto combinándolo con que A es estríctamente mayor que V-1, y estrictamente mayor que V pues un grafo de n nodos y n-1 aristas no requeriría el uso de este algoritmo. Con todo esto podemos deducir que el total es **O(A*logA)**

Cabe destacar que para un grafo denso, O(A*logV) puede ser peor que O(V^2), por tanto el algoritmo de Kruskal se recomienda usar para grafos dispersos (donde el número de aristas no está cercano al número máximo de aristas posibles).

Algoritmo detallado

Los primeros pasos del algoritmo son las inicializaciones como hemos comentado anteriormente. Un conjunto vacío para cada nodo, así como asignar rango 0 a cada uno y el padre a uno mismo (ver *iniciarConjuntos()*).

Después de esto podríamos reducir el algoritmo a tres fases diferenciadas:

- **Ordenación de las aristas**: Queremos hacerlo de manera creciente y lo podemos hacer con la función predefinida *sorted* que en este caso le indicamos mediante una pequeña función que lo ordene por el tercer elemento de la lista.

```
#Ordenamos las aristas de menor peso a mayor
g.aris =  sorted(g.aris,key=getKey)
```

```
#Funcion que te selecciona el elemento deseado de una lista
def getKey(item):
    return item[2]
```

- **Selección de la arista siguiente a comprobar**: Mantendremos un índice i para ir recorriendo la lista ordenada de las aristas, sea incluida o no (por formar ciclo), el índica aumenta una unidad y no se vuelve a comprobar dicha arista. Es como si la estuviéramos eliminando de la lista.

  Lo que hacemos es sacar la arista para poder comprobar los padres de los dos vértices. Una vez los tenemos, si coinciden es que ya están unidos, es decir que formaríamos un bucle de incluir dicha arista.

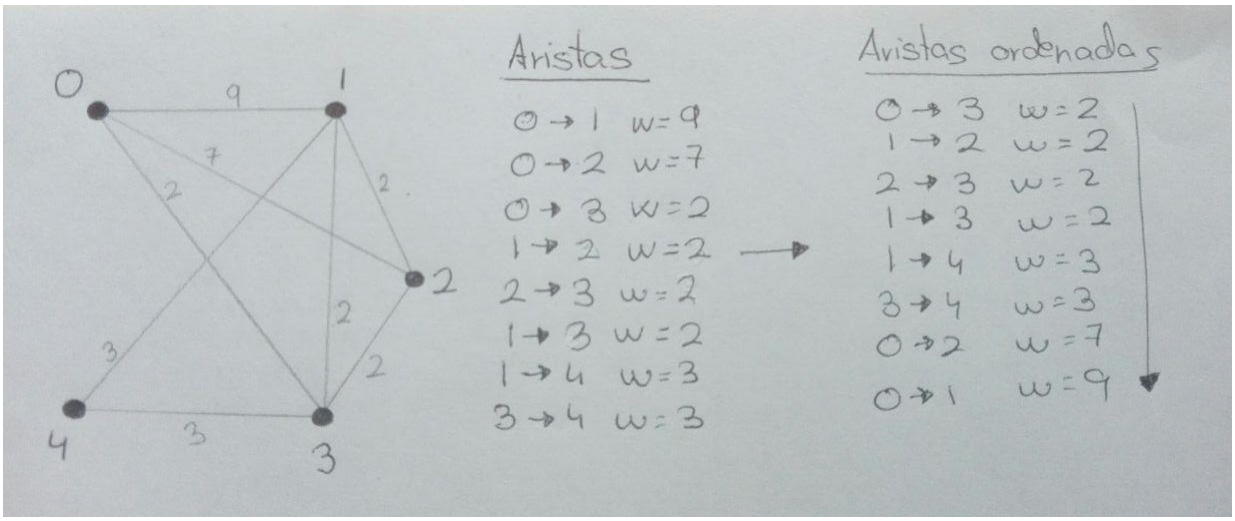
  Si esto no sucede, aumentamos el número de vértices conectados, añadimos la arista al almacen del resultado (con la función predefinida *append()* y llamamos a la función fusionar que conectará de la manera adecuada los vértices.(teniendo en cuenta la mejora que habíamos incluido)

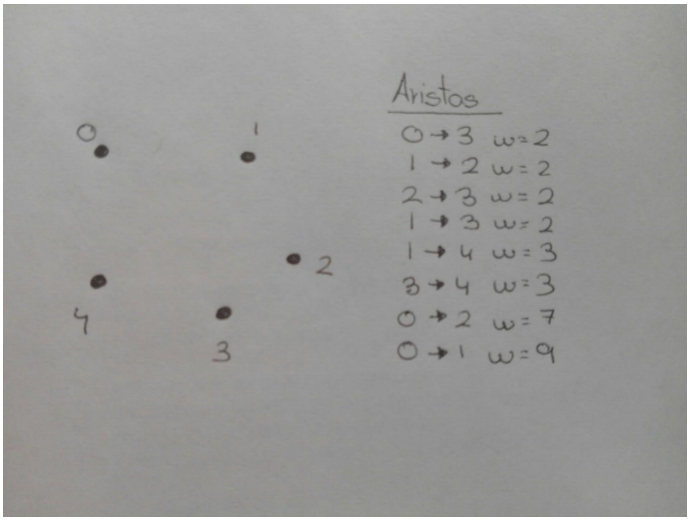
```
#Hasta que no estan los N-1 nodos conectados seguimos
while n < g.V -1 :
    #Como ya estan ordenados, seleccionamos cada vez el
    #camino de menor peso
    u,v,w =  g.aris[i]
    i = i + 1

    x = g.buscar(u)
    y = g.buscar(v)
```

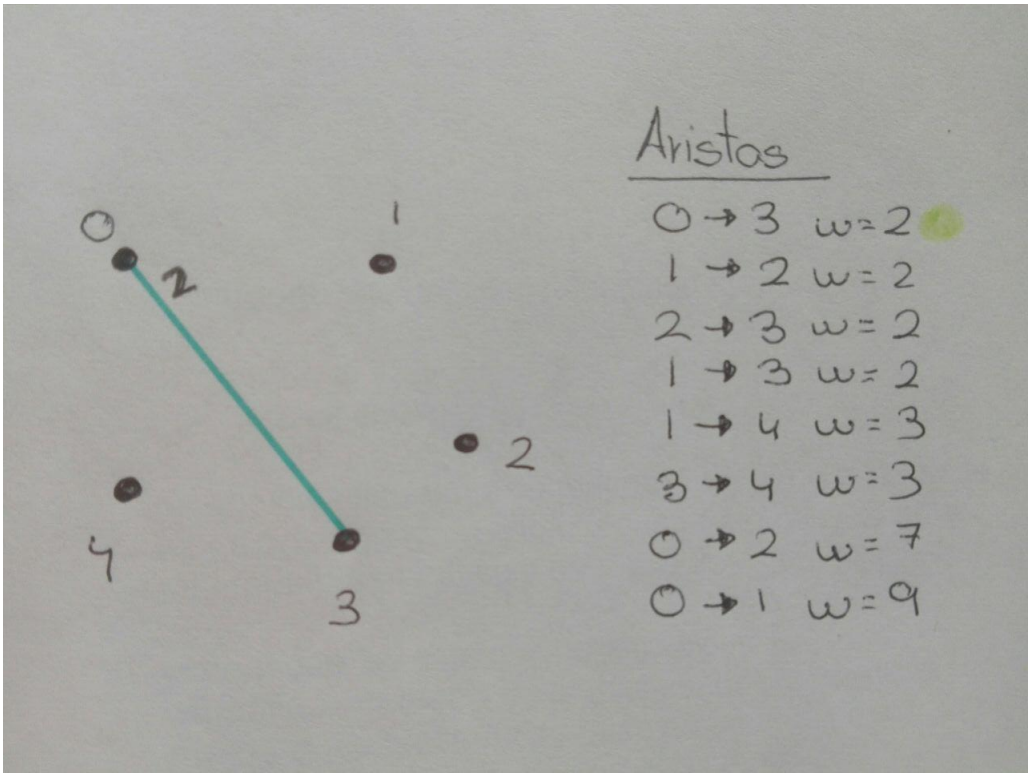
```python
# Comprobación de ciclo
if x != y:
    #Aumentamos el número de vértices conectados
    n = n + 1
    #Metemos la arista buena a la solucion
    res.append([u,v,w])
    g.fusionar(x, y)
else:
    #Hay ciclo,No incluimos la arista
```

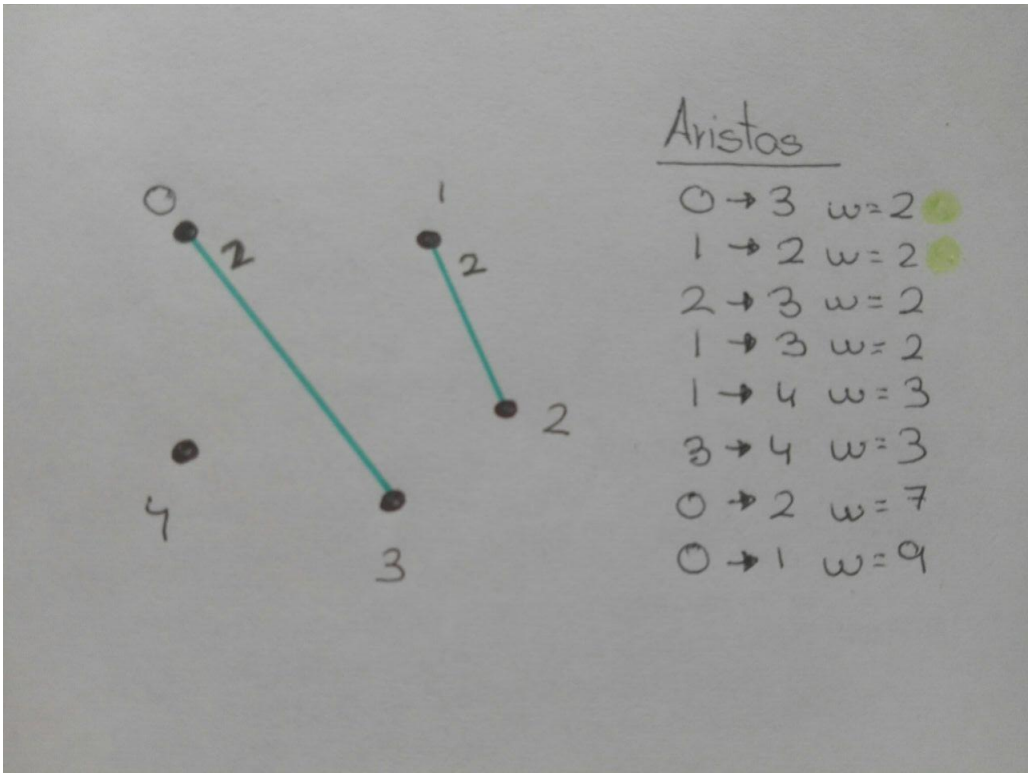
- **Repetición del paso anterior**: Como se ve en las fotos anteriores, el proceso se repite hasta que los N-1 vértices del grafo estén en el árbol de expansión.

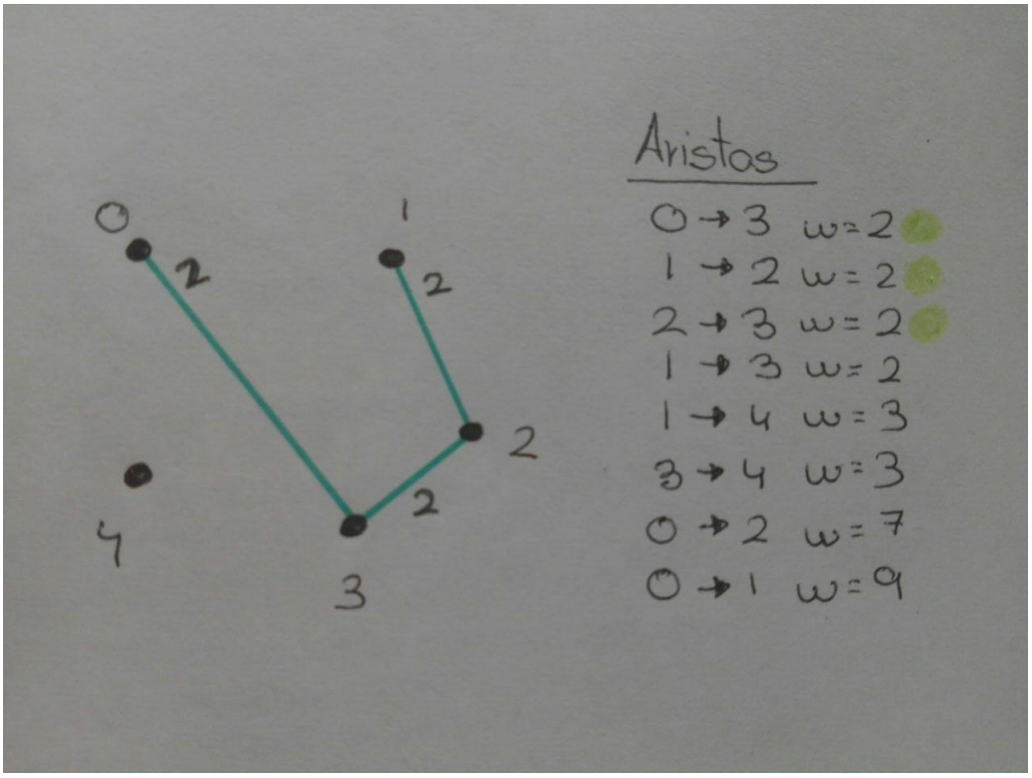
Ejemplo real

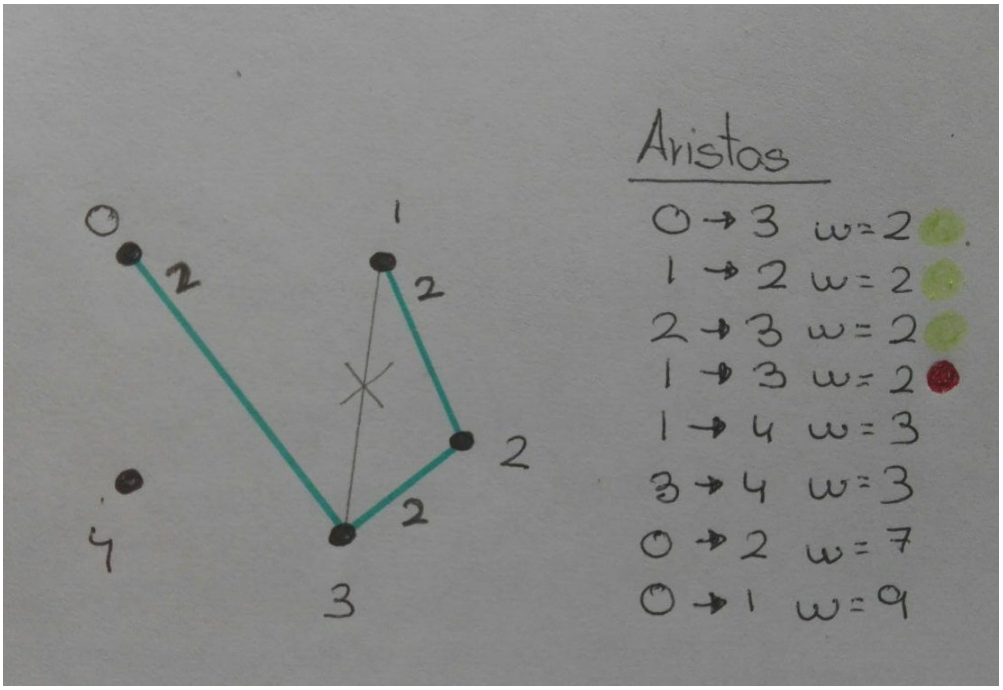


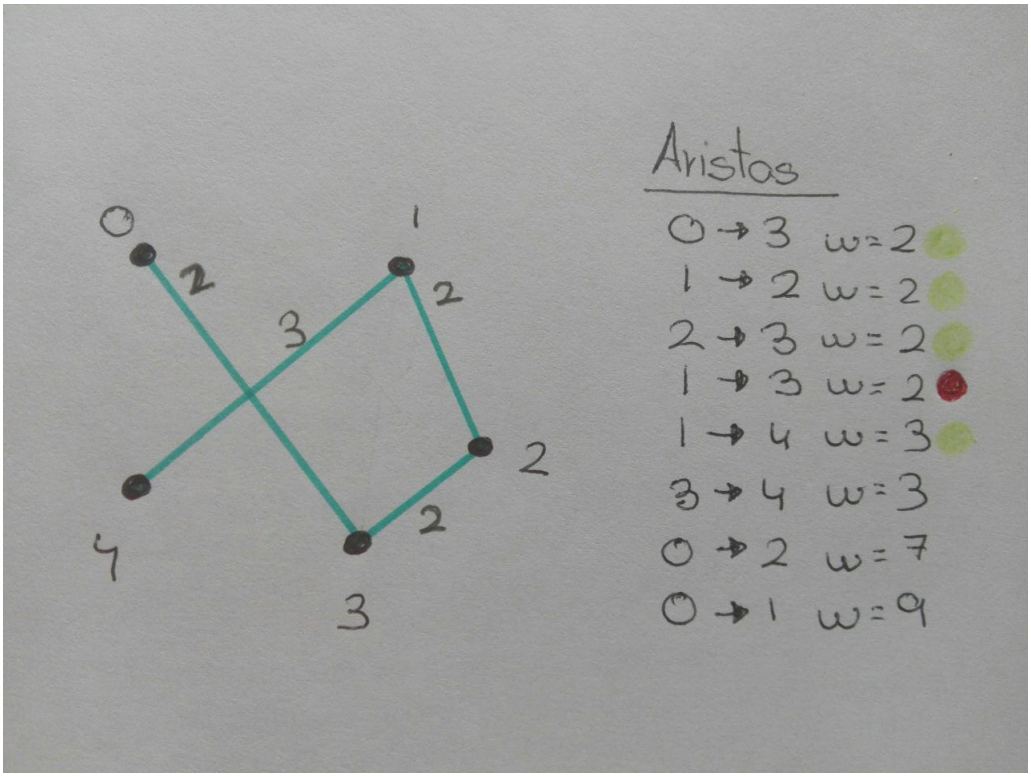



Aristas
0 → 3 w=2
1 → 2 w=2
2 → 3 w=2
1 → 3 w=2
1 → 4 w=3
3 → 4 w=3
0 → 2 w=7
0 → 1 w=9
0
1
2
3
4
2


Aristas
0 → 3 w=2
1 → 2 w=2
2 → 3 w=2
1 → 3 w=2
1 → 4 w=3
3 → 4 w=3
0 → 2 w=7
0 → 1 w=9
0
1
2
3
4
2


Aristas
0 → 3 w=2
1 → 2 w=2
2 → 3 w=2
1 → 3 w=2
1 → 4 w=3
3 → 4 w=3
0 → 2 w=7
0 → 1 w=9
0
1
2
3
4
2
2
2
2


Aristas
0 → 3 w=2
1 → 2 w=2
2 → 3 w=2
1 → 3 w=2
1 → 4 w=3
3 → 4 w=3
0 → 2 w=7
0 → 1 w=9
0
1
2
3
4
2
2
2
2



References

[1] "Introduction to Algorithms, Second Edition" - Thomas H.Cormen

[2] "Fundamentos de Algoritmia" - G. Brassard y P. Bratley

[3] https://jariasf.wordpress.com/2012/04/19/arbol-de-expansion-minima-algoritmo-de-kruskal

[4] https://es.wikipedia.org/wiki/Algoritmo_de_Kruskal

[5] https://jariasf.wordpress.com/2012/04/02/disjoint-set-union-find/