UNIVERSIDAD DE
COSTA RICA

Universidad de Costa Rica

IE-0217: Estructuras de datos y algoritmos

Algoritmo de Kruskal

Estudiante:
Maria Jose Hernandez
Escuela de Ingeniería Eléctrica

Grupo:
01
II-2020

Índice

1. Introducción

Nos referimos a la programación como al arte de comunicarse con la computadora mediante instrucciones que siguen una sintaxis en particular dependiendo del lenguaje que se utilice [6]. Independientemente del lenguaje, el computador es capaz de procesar dos clasificaciones de datos, los simples y los estructurados [2]. Los tipos de datos tienen como propósito clasificar los objetos de un programa y determinar sus posibles valores [5]. Una estructura de datos es una organización particular para almacenar variables de otros tipos [8]. Las estructuras de datos son el objeto principal de estudio del presente curso. Como proyecto final, se desarrollará a lo largo de las siguientes secciones un análisis detallado del algoritmo de Kruskal. A continuación, se introducirán algunos conceptos esenciales para comprender este algoritmo.

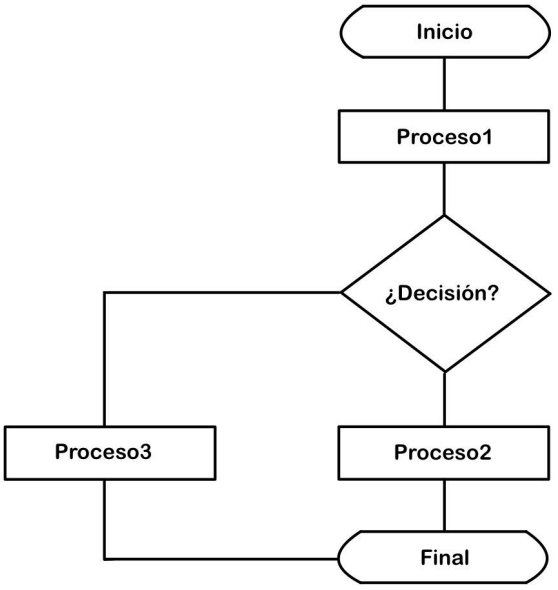
1.1. ¿Qué es un algoritmo?

Es posible pensar en los algoritmos como métodos para la resolución de un problema dado [11]. A partir de esta afirmación se entiende un algoritmo como una secuencia de pasos o instrucciones que permiten dar con la solución del problema.

Un algoritmo debe contar con las siguientes cinco características [7]:

- **Finitud**: Tener siempre un número finito de pasos
- **Definibilidad**: Cada paso debe definirse de un modo preciso
- **Entrada**: Debe tener cantidades dadas inicialmente
- **Salida**: Cantidades que tienen una relación específica con las entradas
- **Efectividad**: Tener operaciones lo más básicas posibles

En la siguiente imagen se puede observar un ejemplo de como luciría un diagrama de flujo para un algoritmo de tres instrucciones o tres procesos.

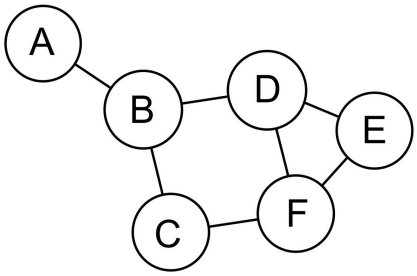


1.2. ¿Qué es un grafo?

Al inicio de esta sección se mencionó cómo los programas podían utilizar tanto datos simples como estructuras de datos. Se introducirá ahora una estructura de dato llamado grafo.

Un grafo es en esencia, un conjunto de vértices (o nodos) unidos mediante aristas (o enlaces) [12].

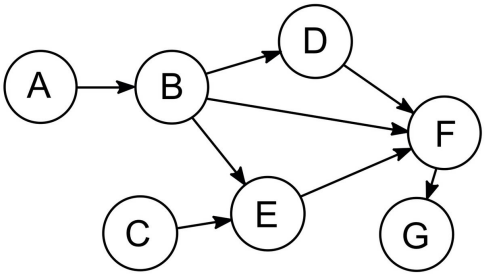
En la imagen siguiente se encuentra una repesentación visual de un grafo.



En este caso los círculos indicados con letras de A, ..., F son los nodos del grafo y las líneas que unen esos nodos son los enlaces del grafo.

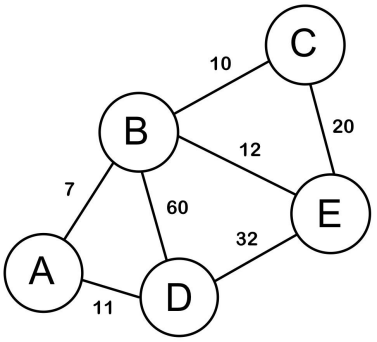
Ahora bien, de acuerdo a las características asociadas a los enlaces del grafo, se pueden diferenciar tres clasificaciones de grafos [10].

- **Grafos no dirigidos**: En este tipo de grafos los enlaces que contiene son bidireccionales y no poseen peso o bien, todos los enlaces poseen el mismo peso. Un ejemplo de este tipo de grafo se presenta en la figura anterior.
- **Grafos dirigidos**: También llamados *dirigrafos*. Se caracterizan por poseer enlaces unidireccionales entre un nodo y otro. Un ejemplo de este tipo de grafo se presenta en la siguiente figura.



Note que, en este caso, el enlace que conecta los nodos **B** y **D** tiene una dirección que le permite transicionar únicamente del estado **B** al estado **D**. Nunca de **D** hacia **B**.

- **Grafos pesados**: En este tipo de grafos, cada enlace tiene un peso asociado, normalmente diferentes entre sí. En la siguiente imagen puede encontrar una representación de cómo luce un grafo pesado.



Note que en este caso el enlace que conecta los nodos **A** y **B** tiene un peso de 7, mientras que el grafo que conecta los nodos **B** y **E** tiene un peso de 12.

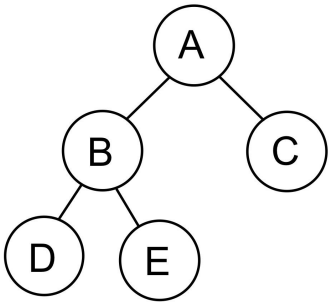
Independientemente del tipo de grafo, es importante resaltar que en el contexto de las estructuras de datos, se pueden considerar los nodos como contenedores de objetos.

1.3. ¿Qué es un árbol?

Los árboles por su parte son otro tipo de estructuras de datos compuestos por nodos y enlaces. La diferencia radica en que los árboles poseen una estructura jerárquica. Es decir, siempre existe un nodo raíz a partir del cual parten diferentes nodos [4].

A continuación se encuentra una represen-

tación gráfica de cómo luciría un árbol de 5 nodos.



Note que en una estructura de árbol no se permiten tener ciclos entre nodos. A diferencia de los grafos.

1.4. Algoritmo de Kruskal

Descubierto en 1956 por Joseph Kruskal [9], este algoritmo parte de un grafo pesado no dirigido para encontrar un árbol de mínima expansión [3]. El árbol de mínima expansión se define como un subconjunto acíclico del grafo que conecta todos los nodos y para el cual la sumatoria de los pesos de cada enlace es el mínimo posible [3]

De esta forma, Kruskal plantea que para encontrar el árbol de mínima expansión se deben ir añadiendo los enlaces de menor peso siempre y cuando estos no formen un subconjunto cíclico.

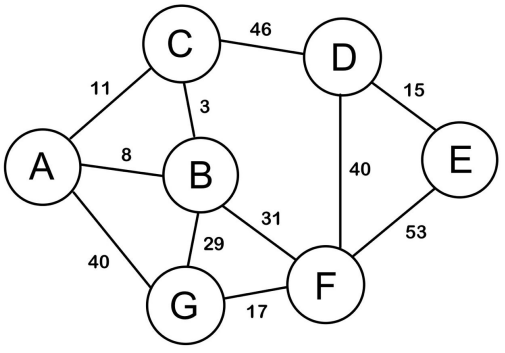
Para ello, se ordenan en forma ascendente los enlaces de acuerdo al peso asociado a ellos. El algoritmo de Kruskal comienza con un árbol vacío, seguidamente se analiza cada enlace de acuerdo al orden anterior. Si la añadidura de este enlace no genera un ciclo en el subconjunto evaluado entonces se agrega el enlace. En caso contrario, se omite y se procede al análisis del siguiente enlace [1].

2. Discusión

En la presente sección se profundizará de mayor manera en cómo funciona el algoritmo de Kruskal. Para ello, se planteará un caso para el cuál se buscará el árbol de expansión mínima de forma manual. Esto para entrar en detalle de cómo se seleccionan los enlaces que se agregarán al árbol. Seguidamente, se procederá a estudiar una implementación del algoritmo en Python.

2.1. ¿Cómo funciona?

Para comprender mejor como funciona el algoritmo de Kruskal considere el siguiente ejemplo. Primero, se parte de un grafo pesado no dirigido de 7 nodos.

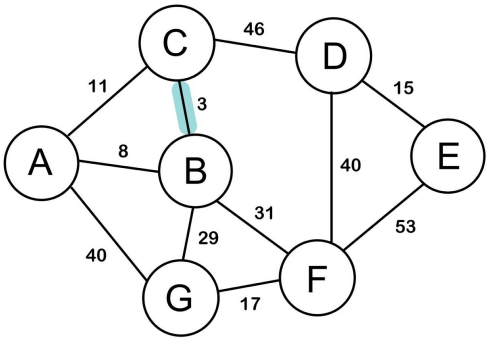


Seguidamente, se deben ordebar los enlaces

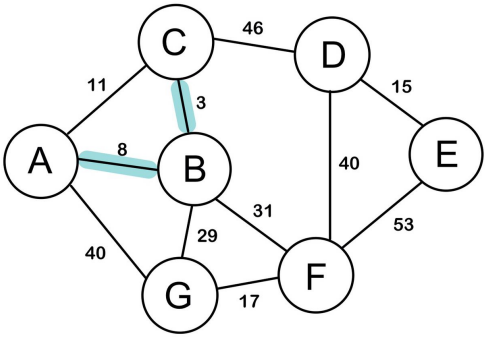
de manera ascendente. Para ello, se crea una tabla cuyas columnas representan los siguientes datos: Nodo, Nodo, Peso.

Nodo	Nodo	Peso
B	C	3
A	B	8
A	C	11
D	E	15
F	G	17
B	G	29
B	F	31
A	G	40
D	F	40
C	D	46
E	F	53

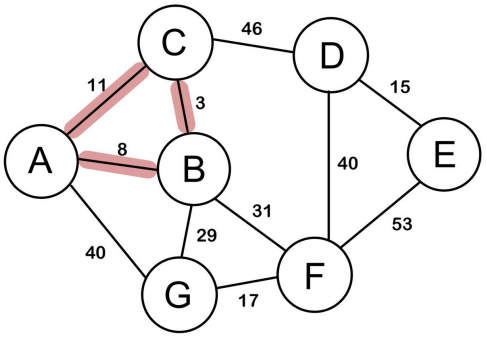
Con esta tabla, se procede a analizar cada enlace. Comenzando desde el de menor peso hasta el de mayor peso. Primero, como el árbol está vacío, se añade el enlace de peso 3.



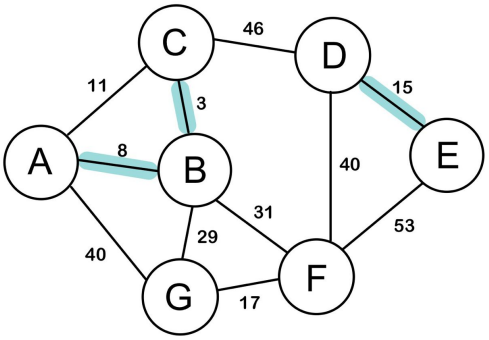
Se procede a analizar el siguiente enlace, que sería el de peso 8. Al no formar ningún ciclo, se añade el enlace AB al árbol.



El siguiente enlace a analizar sería el de peso 11, mas note que al agregar este enlace se crearía un ciclo entre los nodos ABC, tal como lo muestra la siguiente figura

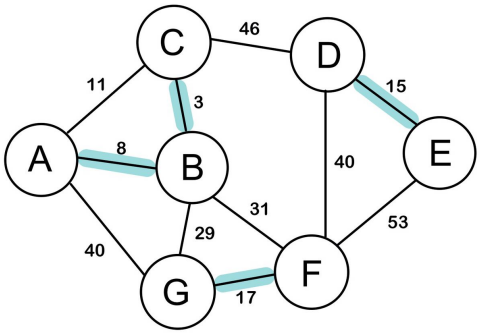


Por esta razón, se omite el noto AC y se procede a analizar el nodo de peso 15. Se agrega ya que no forma ningún ciclo dentro del árbol de expansioń

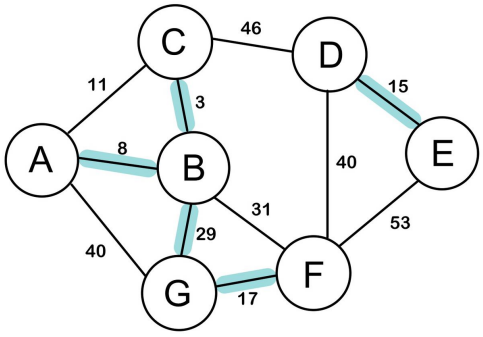


Lo mismo sucede para el enlace con peso 17,

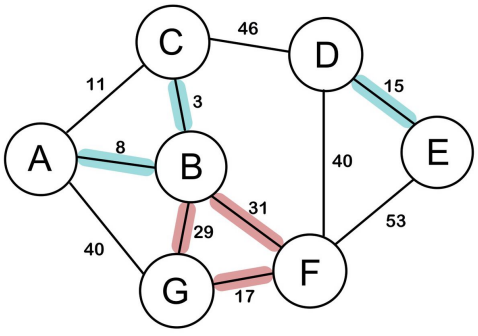
entre los nodos FG



Y para el enlace de peso 29 entre los nodos BG

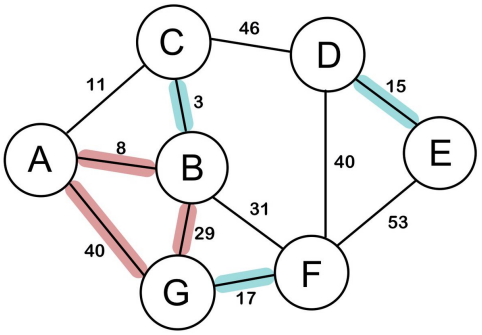


Ahora, note que el enlase siguiente es aquel de peso 31 entre los nodos BF. No obstante, al agregar este enlace al árbol de mínima expansión se crearía un ciclo entre los nodos BFG

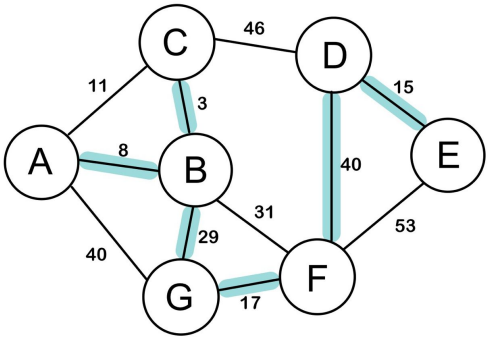


Ahora note que hay dos enlaces con peso 40. Para el algoritmo de Kruskal es indiferente si se escoge uno o el otro. Aunque ambos resultados podrían, potencialmente, generar resultados diferentes cualquier árbol obtenido por este método es un árbol de mínima expansión.

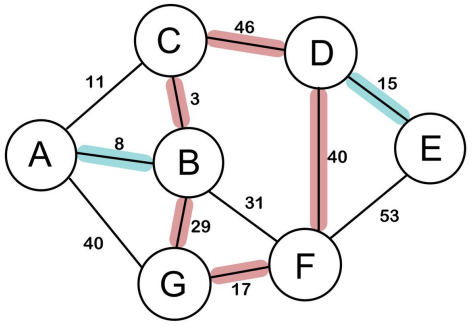
Dicho esto, analizaremos primero el enlace AG, de agregarse este enlace se crearía un ciclo entre los nodos ABG, por lo que se omite dicho enlace.



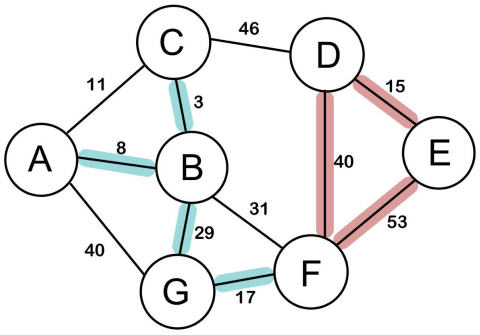
Continuando con el otro enlace de peso 40 entre los nodos DF, este sí se puede añadir al árbol ya que no genera ningún ciclo



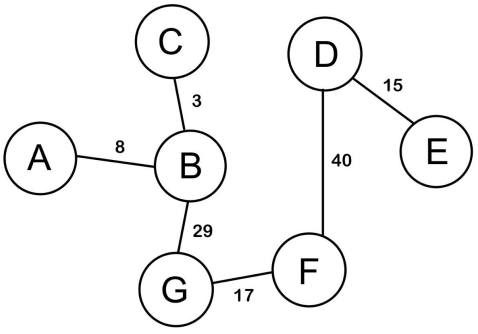
De los enlaces restantes, el de peso 46 entre los nodos CD generaría un ciclo en los nodos BCDFG



De igual forma, el enlace de peso 53 generaria el ciclo DEF



Con esto, concluye el análisis de los enlaces para el grafo original. Finalmente, se obtiene el siguiente árbol de mínima expansión



El peso de este árbol puede calcularse como

$$\sum = 3 + 8 + 15 + 17 + 29 + 40 = 112$$

2.2. Implementación en python

Para implementar el algoritmo en python se debe establecer primero cuáles son los subprocesos que han de realizarse a la hora de ejecutar el algoritmo. A partir del ejemplo anterior se logran identificar los siguientes pasos clave

- Definir los nodos y los enlaces del grafo, así como los pesos asociados a ellos.
- Ordenar los enlaces de acuerdo a su peso y de manera ascendente.
- Analizar cada enlace (de menor a mayor peso) y verificar si su añadidura crea un ciclo en el árbol o no.
- Si el enlace en cuestión no crea ningún ciclo se añade al árbol

Para comenzar con la implementación del algoritmo debemos definir una clase - llamada grafo - que contendrá la información necesaria para obtener luego el árbol de mínima expansión.

Esta clase tendrá como atributos la cantida, de vértices presentes en el grafo y un vector para describir los enlaces del grafo. Cada elemento del vector tendrá como elementos los dos nodos que encierran el enlace y su peso. Una representación matemática de este vector luciría como

$$\{\{1, 2, 7\}, \{1, 3, 5\}, \{2, 4, 3\}, \{3, 4, 9\}, ...\}$$

Con esto, podemos proceder a crear la clase del grafo de la siguiente manera. Donde los atributos se inicializan en el constructor como el número de vértices que recibe como parámetro y la lista vacía de enlaces.

```python
class Grafo:
    def __init__(self, vertices):
        self.Vertices = vertices
        self.Enlaces = []
```

Ahora bien, debemos crear un método capaz de llenar la lista recién creada en el constructor. Para ello, el método deberá recibir como parámetro los nodos entre los que se encuentra el enlace y su peso. Una vez se obtengan estos datos del usuario, la información se añadirá a la lista de enlaces. El método, que forma parte de la clase Grafo, luciría como

```python
def agregar_enlace(self, nodo1, nodo2, peso):
    self.Enlaces.append([nodo1, nodo2, peso])
```

Con ello, ahora debemos crear un método que sea capaz de ordenar la lista de enlaces de acuerdo al tercer atributo de cada elemento, lo que vendría siendo el peso del enlace. Esta función formaría parte de la clase Grafo, por lo que no necesita recibir ningún parámetro adicional.

Para esto, haremos uso de un método incorporado en python, asociado a los arreglos para ordenar sus elementos. Este método se llama *sort*.

```python
def ordenar_enlaces(self):
    self.Enlaces.sort(key=lambda x:x[2])
```

Donde el *keyword* lambda le indica al método sort que debe ordenar el objeto sobre el cuál opera de acuerdo a la condición brindada. En este caso, de acuerdo al tercer miembro de cada elemento del arreglo, lo que vendría siendo el peso del enlace según la función *agregar_enlace*. Para más información sobre las funciones lambda en python revisar el siguiente enlace *https* : *//stackoverflow.com/questions/13669252/what−is − key − lambda*

Ahora bien, una vez definido el grafo y ordenados sus enlaces, se puede proceder a implementar el algoritmo de Kruskal para obtener el árbol de expansión mínima. La parte fundamental de este algoritmo será identificar si los nodos del enlace en cuestión se encuentran en un mismo subconjunto. De ser así, agregar este enlace al árbol añadiría un ciclo. Debemos evitar ciclos ya que, por definición, las estructuras de árbol son estrictamente jerárquicas.

A continuación, se encuentra la funciónń que ejecuta el algoritmo de kruskal.

```python
def kruskal(self):
    self.ordenar_enlaces()
    Arbol = []
    Sets = 1
    SubSets = [-1 for i in range(self.Vertices)]
    for iteracion in range(len(self.Enlaces)):
        nodo1, nodo2, peso = self.Enlaces[iteracion]

        if (SubSets[nodo1] == SubSets[nodo2] == -1):
            Arbol.append(self.Enlaces[iteracion])
            SubSets[nodo1] = Sets
            SubSets[nodo2] = Sets
            Sets = Sets + 1

        elif (SubSets[nodo1] != SubSets[nodo2]):
            Arbol.append(self.Enlaces[iteracion])
            if (SubSets[nodo1] == -1):
                SubSets[nodo1] = SubSets[nodo2]
            elif (SubSets[nodo2] == -1):
                SubSets[nodo2] = SubSets[nodo1]
            else:
                newSubset = SubSets[nodo1]
                for element in range (len(SubSets)):
                    if (SubSets[element] == newSubset):
                        SubSets[element] = SubSets[nodo2]

    return Arbol
```

El algoritmo comienza ordenando la lista de enlaces de acuerdo a la función *ordenar_enlaces* vista anteriormente. Luego, crea un objeto llamado **Arbol** y lo inicializa como una lista vacía. En esta lista se almacenarán los enlaces

que pueden formar parte del árbol de expansión mínima. Seguidamente, se crea un vector del tamaño de la cantidad de vertices del grafo y se asigna a cada casilla el número -1.

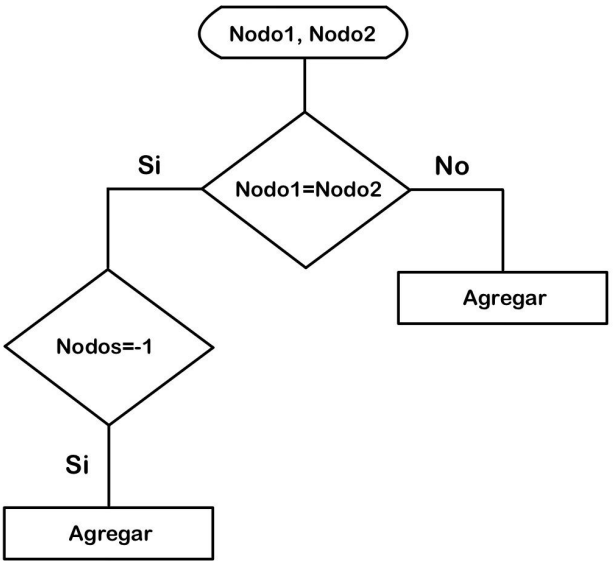
Este es el detalle más importante de todo el algoritmo, ya que cuando un vértice se encuentre con *subset* de -1 indicará que este no ha sido asociado a ningún subconjunto. Si varios vértices poseen el mismo número de subset entonces pertenecen al mismo subconjunto y cualquier enlace con nodos en el mismo subconjunto forma un ciclo.

De esta manera, el algoritmo verifica primero si los nodos del enlace seleccionado se encuentran fuera de cualquier subconjunto (subset de -1), de ser el caso se agrega el enlace.

Luego, si los subsets de los nodos son diferentes, pero alguno de ellos tiene valor de -1 (lo que quiere decir que no pertenece a ningún subconjunto todavía) se agrega el nodo independiente al subconjunto del otro nodo.

Por último, si ambos nodos se encuentran en subconjuntos diferentes entonces cada elemento que se encuentre en el subconjunto del nodo1 se asociará ahora al subconjunto del nodo2.

El siguiente diagrama de flujo resume, a muy alto nivel, la lógica que se sigue para decidir si un enlace se agrega o no



- Si el subset de los nodos es diferente, se agrega el enlace

- Si el subset de los nodos es igual, se agrega el enlace solo cuando ambos subset son igual a menos 1.

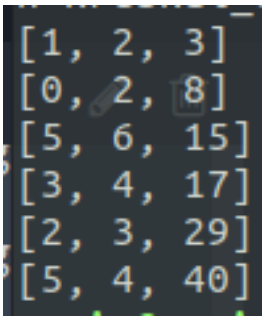
2.3. Corrida de ejemplo

Para ejecutar el programa se debe crear una instancia de la clase Grafo, y luego utilizar la función *agregar_enlace* para definir la configuración del grafo. Seguidamente, creamos un objeto llamado arbol que se inicializa como la aplicación de la función kruskal sobre el grafo definido anteriormente. Luego, imprimimos cada elemento de la lista del arbol. En código, esto luciría como

```
g = Grafo(7)
g.agregar_enlace(0, 2, 8)
g.agregar_enlace(0, 1, 11)
g.agregar_enlace(2, 4, 31)
g.agregar_enlace(5, 6, 15)
g.agregar_enlace(1, 5, 46)
g.agregar_enlace(5, 4, 40)
g.agregar_enlace(4, 6, 53)
g.agregar_enlace(3, 4, 17)
g.agregar_enlace(2, 3, 29)
g.agregar_enlace(1, 2, 3)
g.agregar_enlace(0, 3, 40)

arbol = g.kruskal()
for enlace in range(len(arbol)):
    print (arbol[enlace])
```

Para ejecutar el programa, se definirán los enlaces tal que se forme el grafo mostrado en la sección 1. Esto, de manera que al obtener los resultados se compruebe si se obtuvo el árbol de expansión mínima correcto o no. Los enlaces del árbol obtenidos a partir del programa se muestran a continuación



Note entonces que, al comparar los enlaces de la imagen anterior, se encuentra que estos son los mismos que aquellos encontrados manualmente para el grafo de la sección 1. Con lo que se infiere que el programa funciona de manera correcta.

2.4. Complejidad

De acuerdo a lo estudiado en clase, se entiende complejidad como un análisis cuantitativo de los diferentes algoritmos. Esto, en términos de cuántos pasos pueden involucrar diferentes partes de su ejecución.Para evaluar el algoritmo, es utilizará el modelo de RAM.

Debe encontrarse una función **f(n)** que describa el comportamiento del algoritmo. Donde n es la cantidad de datos de entrada.

Note que la función de Kruskal se compone por un ciclo *for* que realiza una iteración por cada elemento en el arreglo de enlaces. Luego, este ciclo tiene un ciclo *for* interno que recorre el arreglo de subsets, el cual tiene como tamaño la cantidad de vértices del grafo. No obstante, este ciclo se recorre únicamente cuando los subset de los nodos son diferentes entre sí y diferentes de -1.

Sin embargo, para realizar el análisis de la complejidad, estudiaremos los casos de acotación. Esto es la mayor cantidad de pasos posibles y la menor cantidad de pasos posibles.

Se da la mayor cantidad de pasos posibles cuando por cada iteración del ciclo externo se ejecuta el ciclo interno. Para este caso, ocurren self.Vertices iteraciones del ciclo exterior y por cada una de ellas ocurrirían len(SubSets) iteraciones del ciclo interno.

Siendo así, el máximo valor de complejidad para esta implementación del algoritmo de Kruskal sería

$$O(self.Vertices * len(SubSets))$$

Pero, note que el arreglo SubSets se inicializa como un arreglo de tamaño igual a la cantidad

de vertices. Si llamamos a la cantidad de vértices del grafo como **n** entonces el máximo valor de complejidad toma la siguiente forma

$$O(n^2)$$

No obstante, el algoritmo de Kruskal no va a llegar, en ninguna instancia, a procesar esta cantidad de instrucciones ya que para este caso, cada enlace seleccionado debería encontrarse en subconjuntos diferentes diferentes a él mismo. Esto no puede ocurrir ya que, por definición todos los vértices comienzan siendo su propio subconjunto.

De igual forma, ocurrirán casos donde los vértices del nodo a analizar formen parte de un mismo conjunto, por lo que no habría necesidad de ejecutar el ciclo interno.

3. Conclusión

El algoritmo de Kruskal es una manera eficiente de obtener el mínimo árbol de expansión. La esencia de este algoritmo es determinar cuándo se forma un ciclo al agregar un enlace al grafo. De igual forma, a partir de la sección de complejidad se observa que esta implemtación del algoritmo en específico es incapaz de llegar a obtener su máximo nivel de complejidad, ya que nunca se recorrerá el ciclo interno por cada iteración del ciclo externo.

Referencias

[1] Mikhail Atallah. *Algorithms and theory of computation handbook*. CRC Press, Boca Raton, 1999.

[2] O. Cairó. *Fundamentos de programación. Piensa en C*. Pearson Educación, Mexico, 1 edition, 2006.

[3] Leiserson C. Rivest R. Stein C. Cormen, T. *Introduction to algorithms*. McGraw-Hill, London, England., 2 edition, 2003.

[4] Juan Corrales. *Tecnicos de soporte informatico de la Comunidad de Castilla y Leon : [personal laboral*. MAD, Alcala de Gudaira (Sevilla, 2006.

[5] X. Franch. *Estructuras de datos : especificacion, diseno e implementacion*. Edicions UPC, Barcelona, 1996.

[6] Kingsley K. Kingsley, A. *Beginning Programming*. Wiley Publishing, Inc, Indianapolis, Indiana, 2005.

[7] D. Knuth. *El arte de programar ordenadores. Algoritmos fundamentales*, volume 1. Editorial Reverté, Barcelona, 2002.

[8] S. Loosemore. *The GNU C library reference manual*. Free Software Foundation, Cambridge, MA, 2001.

[9] Diana Mesias. Selección de rutas en una red de sensor inalámbrica, en base al nivel de batería y distancia entre nodos sensores mediante la utilización del algoritmo de kruskal. Master's thesis, Pointificia Universidad Católica del Ecuador, Ecuador, 11 2016.

[10] Narciso Oliet. *Estructuras de datos y metodos algoritmicos : 213 ejercicios resueltos*. Garceta, Madrid, 2013.

[11] R Sedgewick. *Algoritmos en C++*. Addison-Wesley Diaz de Santos, Argentina, 1995.

[12] Aguado F. Gago F. Ladra M. Perez G. Vidal C. Vieties, A. *Teoria de grafos : ejercicios resueltos y propuestos, laboratorio con Sage*. Paraninfo, Madrid, 2014.