



# UNIVERSIDAD DE GUADALAJARA

Red Universitaria e Institución Benemérita de Jalisco

CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E IGENIERÍAS

## ANALISIS DE ALGORITMOS

EQUIPO TR3S

**GARCIA SALDIVAR HUGO GABRIEL**

**MACIEL VARGAS OSWALDO DANIEL**

D06 20/11/2025

*Actividad/Participación*

Implementación y comparación de Prim y Kruskal.

## I. Implementación del código

El objetivo de esta actividad es implementar y comparar los algoritmos Prim y Kruskal para obtener el árbol de expansión mínima (MST) de un grafo. Este grafo contará con las siguientes características: debe ser un grafo no dirigido, ponderado, con al menos 6 nodos y 9 aristas. Esto fue logrado con la siguiente función:

```
def crear_grafo(num_nodos=6, num_aristas=9, peso_min=1, peso_max=20):
```

```
    grafo = {i: [] for i in range(num_nodos)}
```

```
    usadas = set()
```

```
    random.seed(None)
```

```
    while len(usadas) < num_aristas:
```

```
        u = random.randint(0, num_nodos - 1)
```

```
        v = random.randint(0, num_nodos - 1)
```

```
        if u == v:
```

```
            continue
```

```
        arista = tuple(sorted((u, v)))
```

```
        if arista in usadas:
```

```
            continue
```

```
        peso = random.randint(peso_min, peso_max)
```

```
        usadas.add(arista)
```

```
        grafo[u].append((v, peso))
```

```

grafo[v].append((u, peso))

return grafo

```

Esta función construye el grafo con las características que anteriormente definimos, además de definirle un rango de peso menor y mayor.

Ahora lo que sigue es implementar nuestros algoritmos Prim y Kruskal para buscar el MST del grafo anteriormente generado. Para ello creamos 2 funciones diferentes para cada algoritmo. Prim utilizando *heap* que es una cola de prioridad; mantiene siempre el elemento más pequeño al frente:

```

def prim_mst(grafo, inicio=0):

    visitado = set()

    mst = []

    heap = [(0, inicio, None)]

    while heap and len(visitado) < len(grafo):

        costo, nodo, padre = heapq.heappop(heap)

        if nodo in visitado:

            continue

        visitado.add(nodo)

        if padre is not None:

            mst.append((padre, nodo, costo))

        for vecino, peso in grafo[nodo]:

            if vecino not in visitado:

                heapq.heappush(heap, (peso, vecino, nodo))

    return mst

```

Y Kruskal que utiliza *Union-Find*. Esta función detecta los ciclos, con su detección podemos evitar estos ciclos y que se cumpla correcta el funcionamiento de un MST.

```
def kruskal_mst(grafo):  
  
    aristas = []  
  
    for u in grafo:  
  
        for v, peso in grafo[u]:  
  
            if u < v: # Evitar duplicados  
  
                aristas.append((peso, u, v))  
  
    aristas.sort()  
  
  
    padre = {n: n for n in grafo}  
  
    rango = {n: 0 for n in grafo}  
  
  
    def find(n):  
  
        if padre[n] != n:  
  
            padre[n] = find(padre[n])  
  
        return padre[n]  
  
  
    def union(a, b):  
  
        ra = find(a)  
  
        rb = find(b)  
  
        if ra != rb:  
  
            if rango[ra] < rango[rb]:  
                rango[ra] += 1  
                padre[rb] = ra  
            else:  
                rango[rb] += 1  
                padre[ra] = rb
```

```

padre[ra] = rb

elif rango[ra] > rango[rb]:
    padre[rb] = ra

else:
    padre[rb] = ra
    rango[ra] += 1

return True

return False

mst = []

for peso, u, v in aristas:
    if union(u, v):
        mst.append((u, v, peso))

return mst

```

Y finalmente una función para imprimir la matriz de adyacencia, esta matriz sirve tanto para comprender donde existen conexiones entre nodos como para definir los pesos que tienen las aristas que hacen esta conexión.

```

def imprimir_matriz(grafo):

    num_nodos = len(grafo)

    matriz = [[0]*num_nodos for _ in range(num_nodos)]

    for u in grafo:
        for v, peso in grafo[u]:
            matriz[u][v] = peso

    print("Matriz de adyacencia del grafo:")

```

```
for fila in matriz:
```

```
    print(fila)
```

## II. Diferencias en los procesos

Las diferencias entre los algoritmos son muy interesantes, mientras que Prim tiene un enfoque en nodos, Kruskal tiene un enfoque con las Aristas.

Al algoritmo Prim solo le importan las aristas que tocan el árbol que ya ha construido. No le importa si hay una arista de peso 1 al otro lado del grafo; si no está conectada a lo que ya visitó, no la puede ver todavía.

Se puede decir que Prim sigue este flujo de trabajo: primeramente, se comienza de un nodo inicial y a partir de ahí se crece continuamente, se debe de tener al menos un solo componente conectado, finalmente explora sus vecinos mas cercanos y toma el más barato.

Por otro lado, Kruskal tiene una visión total. Ordena todas las aristas del grafo de menor a mayor antes de empezar. Elige la mejor opción global disponible, sin importar la conectividad actual.

Es decir, Kruskal empieza con todos los nodos aislados y va agregando las aristas más baratas de todo el mapa, sin importar dónde estén. Su objetivo es buscar la arista mas barata de todo el grafo y, si no crea un ciclo, la une.

## III. Diferencias en resultados

Ahora para comparar los resultados entre los 2 algoritmos, vamos a utilizar el siguiente grafo representado por la matriz de adyacencia:

```
[0, 1, 0, 4, 9, 0]
[1, 0, 1, 8, 3, 11]
[0, 1, 0, 0, 0, 0]
[4, 8, 0, 0, 7, 0]
[9, 3, 0, 7, 0, 10]
[0, 11, 0, 0, 10, 0]
```

De manera muy interesante en este ejemplo ambos resultados en ambos algoritmos fueron los mismos:

<b>Aristas seleccionadas por PRIM:</b> 0 -- 1 (peso 1) 1 -- 2 (peso 1) 1 -- 4 (peso 3) 0 -- 3 (peso 4) 4 -- 5 (peso 10) <b>Peso total del MST: 19</b>
---

<b>Aristas seleccionadas por KRUSKAL:</b> 0 -- 1 (peso 1) 1 -- 2 (peso 1) 1 -- 4 (peso 3) 0 -- 3 (peso 4) 4 -- 5 (peso 10) <b>Peso total del MST: 19</b>
--

Ahora pondremos otro caso a prueba, utilizaremos el siguiente grafo:

[0, 1, 15, 0, 7, 0] [1, 0, 0, 0, 12, 13] [15, 0, 0, 0, 16, 7] [0, 0, 0, 0, 10, 13] [7, 12, 16, 10, 0, 0] [0, 13, 7, 13, 0, 0]
--

Y obtendremos los siguientes resultados:

<b>Aristas seleccionadas por PRIM:</b> 0 -- 1 (peso 1) 0 -- 4 (peso 7) 4 -- 3 (peso 10) 1 -- 5 (peso 13) 5 -- 2 (peso 7) <b>Peso total del MST: 38</b>
--

<b>Aristas seleccionadas por KRUSKAL:</b> 0 -- 1 (peso 1) 0 -- 4 (peso 7) 2 -- 5 (peso 7) 3 -- 4 (peso 10) 1 -- 5 (peso 13) <b>Peso total del MST: 38</b>
---

La diferencia en el orden de los pasos ocurre porque Kruskal selecciona aristas basándose en una prioridad global (de menor a mayor peso), lo que le permite elegir aristas en cualquier parte del grafo y conectar partes del grafo aisladas. En cambio, Prim está obligado a mantener un crecimiento continuo desde el nodo inicial, por lo que solo puede seleccionar aristas que estén conectadas directamente al nodo actual. Sin embargo, aunque el camino para construirlo sea distinto, el resultado final (MST) suele ser prácticamente el mismo en ambos algoritmos, o por lo menos nosotros no encontramos un caso en donde esto no sucediera.

## IV. Complejidad aproximada

Apoyándonos con la información de las diapositivas sabemos que la complejidad temporal para Prim es de  $O(E \log V)$ . Esto lo obtenemos multiplicando en numero de aristas ( $E$ ) por el costo de operar *heap* ( $O(\log V)$ ) dándonos como resultado  $O(E \log V)$ .

Por otro lado tenemos la complejidad temporal de Kruskal que es de  $O(E \log E)$  o equivalentemente a  $O(E \log V)$ . El paso más costoso es ordenar las aristas por peso, lo cual toma  $O(E \log E)$ . El proceso de iterar y unir conjuntos usando *Union-Find* es extremadamente rápido, casi lineal. Por lo tanto, el tiempo total está dominado por el ordenamiento inicial:  $O(E \log E)$ .

## V. Conclusiones del equipo

Ambos algoritmos nos parecieron bastante interesantes tanto por sus usos, rapidez, implementación y principalmente por la lógica que llevan detrás. En general se nos hizo mas sencillo implementar Prim pues al usar de una manera más directa una técnica de tipo voraz fue más simple de hacerlo.

Algo que queremos señalar como equipo es que podemos decir que en los algoritmos donde se utilice una técnica voraz de forma tan directa puede ser tanto beneficioso como perjudicante dependiendo de la situación. Si bien estas técnicas se caracterizan por tener un tiempo de ejecución increíblemente corto, no podemos negar que evitan muchos factores y consideraciones que en un futuro podrían servirnos.

Por ello lo mejor es analizar el problema para definir cuál es la técnica mas eficiente tanto en tiempo, lógica y mantenimiento con datos muy grandes o con diversas consideraciones.