MANUAL TÉCNICO.

MATEMÁTICA PARA COMPUTACIÓN 2.

Nombres:	Carnet:
Surama De Jesús Lorenzana Lorenzana.	**202200200**
Oscar Alfredo Sierra Sofianos.	**201908320**

Objetivos.

➢ Objetivo General

Aplicar los conceptos generales sobre la teoría de grafos enfocados a programación.

➢ Objetivos Específicos

● Comprender la utilidad de la teoría de grafos.

● Aplicar la teoría de grafos empleando distintos lenguajes de programación.

● Demostrar por medio de un entorno gráfico las aplicaciones de la teoría de grafos.

Utilizando el método de Kruskal para un grafo de n vértices y aristas.

- Usando el algoritmo de Kruskal.

Se tiene que un grafo tiene que ser conexo para utilizar el algoritmo, cumpliendo con esto, se creó un código capaz de medir el peso menor de cada arista, y a través de un ciclo “while” poder verificar que el peso de la arista que se ha tomado sea la más pequeña, comparando de esta manera cada arista hasta completar el algoritmo, de igual manera se verifica que su creación no cree ciclos en el grafo, si esto se cumple se irán uniendo nuestros vértices a través de nuestras aristas.

```
6
7   def kruskal_algorithm(vertices, aristas):
8       grafo = nx.Graph()
9       grafo.add_nodes_from(vertices)
10      grafo.add_weighted_edges_from(aristas)
11
12      mst = nx.Graph()
13      mst.add_nodes_from(vertices)
14
15      aristas = [(w, u, v) for u, v, w in grafo.edges(data='weight')]
16      heapq.heapify(aristas)
17
18      union_find = nx.utils.union_find.UnionFind(vertices)
19
20      while aristas and len(mst.edges) < len(vertices) - 1:
21          w, u, v = heapq.heappop(aristas)
22
23          if union_find[u] != union_find[v]:
24              mst.add_edge(u, v, weight=w)
25              union_find.union(u, v)
26
27      return mst
```

Utilizamos el modulo heapq para darle prioridad a nuestras aristas, esto es porque nuestras aristas nos indicarán su peso, que es el que necesitamos para ir generando nuestro algoritmo de Kruskal.

```
14
15      aristas = [(w, u, v) for u, v, w in grafo.edges(data='weight')]
16      heapq.heapify(aristas)
```

Para la unión de nuestros vértices se utilizarán las aristas con su peso, cuando sea indicado de donde a donde irán conectados se generaran las aristas que es la unión de dichos vértices deseados.

```
    union_find = nx.utils.union_find.UnionFind(vertices)

    while aristas and len(mst.edges) < len(vertices) - 1:
        w, u, v = heapq.heappop(aristas)

        if union_find[u] != union_find[v]:
            mst.add_edge(u, v, weight=w)
            union_find.union(u, v)

    return mst
```

Generamos el grafo al que se le aplicará el algoritmo de Kruskal, en el se verá reflejado los vértices, aristas y pesos. Se utilizó el código que nos genera una imagen de nuestro grafo creado, es por ello que lo utilizamos en formato png, para tener una mejor visualización de este.

```
def dibujar_grafo(grafo, nombre):
    dot = Digraph(name=nombre, format='png')
    for u, v, w in grafo.edges(data=True):
        dot.edge(str(u), str(v), label=str(w['weight']))

    dot.view()
```

Al crear el grafo se utilizó un código que fuera capaz de agregar los vértices y aristas deseados, pero para ello se deben de poder añadir, de esta manera en uno de nuestros métodos se pueden agregar los vértices,

```
def agregar_vertice():
    vertice = entry_vertice.get()
    if vertice:
        vertices.append(vertice)
        entry_vertice.delete(0, tk.END)
        actualizar_listas()
```

Y en el otro las aristas.

```
42
43  def agregar_arista():
44      origen = entry_origen.get()
45      destino = entry_destino.get()
46      peso = entry_peso.get()
47      if origen and destino and peso:
48          aristas.append((origen, destino, int(peso)))
49          entry_origen.delete(0, tk.END)
50          entry_destino.delete(0, tk.END)
51          entry_peso.delete(0, tk.END)
52          actualizar_listas()
```

Se podrá modificar el grafo a nuestro criterio, creando nuevas conexiones a través de nuestros vértices y aristas nuevos con sus respectivos pesos. Acá apreciamos donde a nuestro grafo mst se le pueden agregar de igual manera vértices y aristas, creando nuevos algoritmos.

```
12      mst = nx.Graph()
13      mst.add_nodes_from(vertices)
```

Llegando al objetivo final en el que ya fue creado el grafo o añadido los vértices, aristas y pesos se procede a calcular un segundo grafo que es nuestro generador minimal por kruskal del grafo original, si no se tiene un grafo o vértices, aristas y pesos añadidos nos tirará un mensaje en el que nos pedirá esta información.

```
63  def calcular_mst():
64      if not vertices or not aristas:
65          messagebox.showerror("Error", "Por favor, ingrese al menos un vértice y una arista.")
66          return
67
```

Cuando tengamos todos los datos solicitados el código empezara a generar un árbol generador minimal por Kruskal, en el que recubrirá cada vértice, es decir que los generará con la suma de sus aristas siendo esta suma lo más pequeño posible, sin generar ciclos en las conexiones con los vértices.

```
    grafo_original.add_nodes_from(vertices)
    grafo_original.add_weighted_edges_from(aristas)

    mst = kruskal_algorithm(vertices, aristas)
    dibujar_grafo(grafo_original, 'grafo_original')
    dibujar_grafo(mst, 'arbol_cubrimiento_minimo')
```

Cuando todo fue calculado con éxito nos mostrará un mensaje en el que indicará que ya fue calculado el algoritmo por Kruskal del grafo original y que de igual manera creo nuestro mts que deseábamos. (En el recuadro de datos estará nuestro titulo de lo que estamos calculando “Algoritmo de kruskal”.)

```
    messagebox.showinfo("Información", "Se calcularon y dibujaron el grafo original y el MST.")

app = tk.Tk()
app.title("Algoritmo de Kruskal")
```