

## 1. Main

En Python, el bloque `if __name__ == "__main__":` se utiliza para determinar si el archivo actual está siendo ejecutado como el programa principal o si ha sido importado como un módulo en otro archivo. Cuando se ejecuta un script de Python directamente desde la línea de comandos o desde un entorno de desarrollo, Python establece automáticamente el valor de `__name__` a `"__main__"`. Esto significa que el bloque de código dentro de `if __name__ == "__main__":` se ejecutará. Por otro lado, cuando se importa un archivo Python como un módulo en otro archivo, el valor de `__name__` se establece en el nombre del archivo (sin la extensión `.py`). En este caso, el bloque `if __name__ == "__main__":` no se ejecutará, ya que el archivo no se está ejecutando como el programa principal.



```
1 import tkinter as tk
2 from Interfaz.MainView import GrafoVisualizer
3
4 if __name__ == "__main__":
5     root = tk.Tk()
6     app = GrafoVisualizer(root)
7     root.mainloop()
```

Este código crea una ventana de aplicación utilizando `tkinter` y muestra un grafo utilizando la clase `GrafoVisualizer` dentro de esa ventana. El programa se ejecutará hasta que el usuario cierre la ventana. Este código en Python utiliza la biblioteca `tkinter` para crear una interfaz gráfica de usuario (GUI) y visualizar un grafo. Aquí está el desglose del funcionamiento:


- `import tkinter as tk`: Esto importa el módulo `tkinter` y lo renombra como `tk`, lo que facilita su uso en el código.
- `from Interfaz.MainView import GrafoVisualizer`: Esto importa la clase `GrafoVisualizer` desde el archivo `MainView.py` dentro del paquete `Interfaz`.

Presumiblemente, esta clase es la que define la interfaz y la lógica para visualizar el grafo.

- `if __name__ == "__main__":`: Esta línea comprueba si el script está siendo ejecutado como el programa principal. Es una buena práctica en Python para garantizar que el código dentro de este bloque solo se ejecute cuando el script se ejecuta directamente y no cuando se importa como un módulo en otro script.
- `root = tk.Tk()`: Aquí se crea una instancia de la clase Tk de tkinter, que representa la ventana principal de la aplicación.
- `app = GrafoVisualizer(root)`: Se crea una instancia de la clase GrafoVisualizer, pasando root (la ventana principal) como su padre. Esto probablemente inicializa la interfaz de usuario y configura la visualización del grafo dentro de la ventana.
- `root.mainloop()`: Este método inicia el bucle principal de eventos de la aplicación. Esto hace que la ventana aparezca en pantalla y permite que la aplicación responda a las interacciones del usuario (como hacer clic en botones, escribir en campos de texto, etc.). El programa permanecerá en este bucle hasta que la ventana se cierre por el usuario.

## 2. Lógica

### 1. Módulo



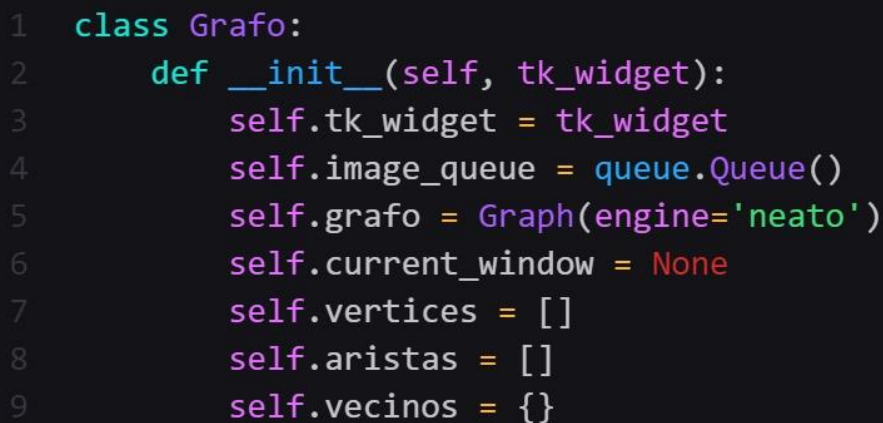
```
1  from graphviz import Graph
2  import tkinter.messagebox as messagebox
3  from PIL import Image, ImageTk
4  import tkinter as tk
5  import queue
```

Este fragmento de código importa varios módulos que serán utilizados para crear una aplicación que probablemente involucre la visualización de gráficos (usando Graphviz), la manipulación de imágenes (usando PIL), la creación de una interfaz gráfica de usuario (usando tkinter), y la implementación de una estructura de datos tipo cola (usando queue). Aquí está el desglose de lo que hace cada línea:

- `from graphviz import Graph`: Esta línea importa la clase Graph del módulo graphviz. graphviz es una biblioteca de Python que permite crear y renderizar gráficos mediante el uso del software Graphviz.

- `import tkinter.messagebox as messagebox`: Esto importa el módulo `messagebox` del paquete `tkinter` y lo renombra como `messagebox`. El módulo `messagebox` proporciona funciones para mostrar cuadros de diálogo de mensaje en una aplicación `tkinter`.
- `from PIL import Image, ImageTk`: Esta línea importa las clases `Image` y `ImageTk` del módulo `PIL`. `PIL` es el acrónimo de "Python Imaging Library", que proporciona capacidades para abrir, manipular y guardar muchos tipos diferentes de imágenes.
- `import tkinter as tk`: Esto importa el módulo `tkinter` y lo renombra como `tk`. `tkinter` es una biblioteca estándar de Python que proporciona herramientas para crear interfaces gráficas de usuario (GUI).
- `import queue`: Esta línea importa el módulo `queue`, que proporciona estructuras de datos tipo cola (o fila) seguras para subprocesos en Python. La cola es una estructura de datos que sigue el principio "primero en entrar, primero en salir" (FIFO), lo que significa que el primer elemento que entra en la cola es el primero en salir de ella.

## 2. Clase Grafo



```

1  class Grafo:
2      def __init__(self, tk_widget):
3          self.tk_widget = tk_widget
4          self.image_queue = queue.Queue()
5          self.grafo = Graph(engine='neato')
6          self.current_window = None
7          self.vertices = []
8          self.aristas = []
9          self.vecinos = {}

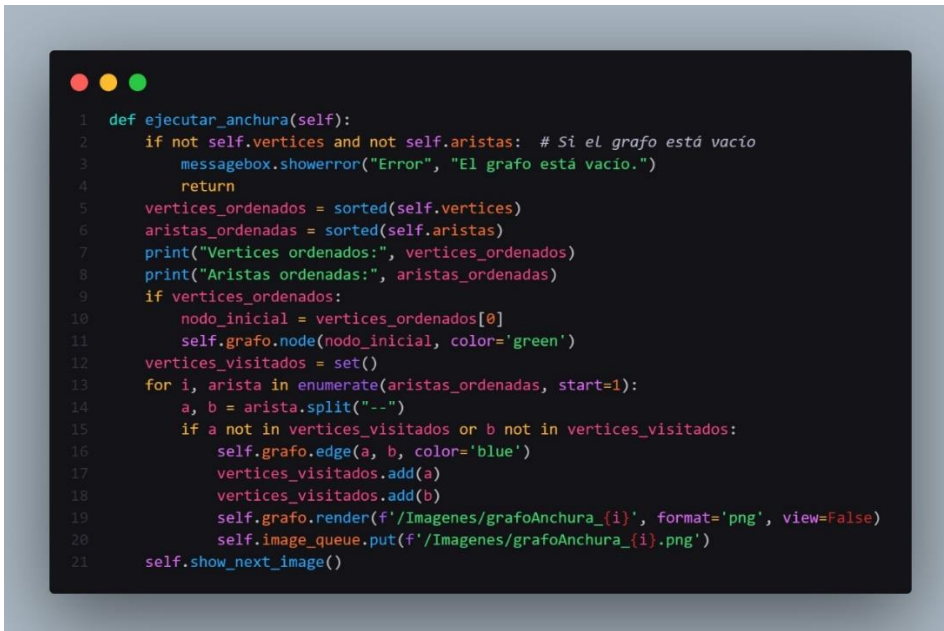
```

Esta clase `Grafo` representa un grafo y proporciona métodos y atributos para trabajar con él, incluida la creación y visualización del grafo. Aquí está el desglose de lo que hace:

- `class Grafo`: Define una clase llamada `Grafo`. Las clases en Python son una forma de organizar y encapsular funciones y variables relacionadas.

- `def __init__(self, tk_widget)::` Este es el método constructor de la clase Grafo, que se llama automáticamente cuando se crea un nuevo objeto Grafo. Toma un parámetro llamado `tk_widget`, que parece ser un widget de tkinter que se utilizará para mostrar la representación gráfica del grafo.
- `self.tk_widget = tk_widget:` Guarda una referencia al widget de tkinter pasado como argumento en el atributo `tk_widget` de la instancia de la clase Grafo.
- `self.image_queue = queue.Queue():` Inicializa una cola vacía utilizando la clase `Queue` del módulo `queue`. Esta cola se utilizará para almacenar imágenes generadas del grafo.
- `self.grafo = Graph(engine='neato'):` Crea una instancia de la clase `Graph` del módulo `graphviz`, que se importó anteriormente. Se configura el motor de renderizado del grafo como `'neato'`. `neato` es uno de los motores de `Graphviz` que se utiliza para dibujar grafos no dirigidos.
- `self.current_window = None:` Inicializa el atributo `current_window` como `None`. Este atributo probablemente se utilizará para realizar un seguimiento de la ventana actual en la que se muestra la representación gráfica del grafo.
- `self.vertices = []:` Inicializa una lista vacía llamada `vertices`. Esta lista se utilizará para almacenar los vértices del grafo.
- `self.aristas = []:` Inicializa una lista vacía llamada `aristas`. Esta lista se utilizará para almacenar las aristas del grafo.
- `self.vecinos = {}:` Inicializa un diccionario vacío llamado `vecinos`. Este diccionario se utilizará para almacenar los vecinos de cada vértice del grafo.

### 3. Ejecutar Anchura



```

1  def ejecutar_anchura(self):
2      if not self.vertices and not self.aristas: # Si el grafo está vacío
3          messagebox.showerror("Error", "El grafo está vacío.")
4          return
5      vertices_ordenados = sorted(self.vertices)
6      aristas_ordenadas = sorted(self.aristas)
7      print("Vertices ordenados:", vertices_ordenados)
8      print("Aristas ordenadas:", aristas_ordenadas)
9      if vertices_ordenados:
10         nodo_inicial = vertices_ordenados[0]
11         self.grafo.node(nodo_inicial, color='green')
12         vertices_visitados = set()
13         for i, arista in enumerate(aristas_ordenadas, start=1):
14             a, b = arista.split("--")
15             if a not in vertices_visitados or b not in vertices_visitados:
16                 self.grafo.edge(a, b, color='blue')
17                 vertices_visitados.add(a)
18                 vertices_visitados.add(b)
19                 self.grafo.render(f'/Imagenes/grafosAnchura_{i}', format='png', view=False)
20                 self.image_queue.put(f'/Imagenes/grafosAnchura_{i}.png')
21         self.show_next_image()

```

Este método realiza una búsqueda en anchura en el grafo y muestra el proceso paso a paso, generando imágenes de cada estado del grafo durante la búsqueda. Aquí está el desglose de lo que hace:

- Comprueba si el grafo está vacío. Si no hay vértices ni aristas en el grafo, muestra un mensaje de error y devuelve.
- Ordena los vértices y aristas del grafo.
- Si hay vértices en el grafo, el primer vértice ordenado se colorea de verde para indicar que es el nodo inicial de la búsqueda en anchura.
- Inicializa un conjunto vacío para almacenar los vértices visitados.
- Itera sobre cada arista ordenada. Para cada arista, si alguno de sus vértices no ha sido visitado, se colorea la arista de azul, los vértices se marcan como visitados, y se renderiza y guarda una imagen del grafo en ese estado.
- Llama al método `show_next_image` para mostrar la siguiente imagen en la cola de imágenes generadas durante el proceso de búsqueda en anchura.

#### 4. Ejecutar Profundidad

```

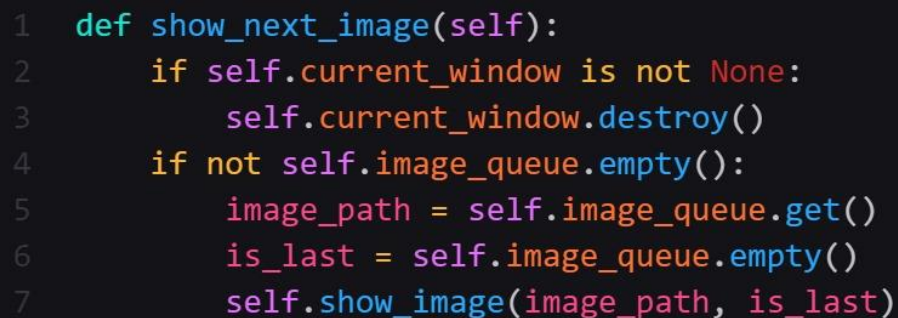
1  def ejecutar_profundidad(self):
2      if not self.vertices and not self.aristas: # Si el grafo está vacío
3          messagebox.showerror("Error", "El grafo está vacío.")
4          return
5      def dfs(nodo, visitados, nodo_inicial, i=1):
6          visitados.add(nodo)
7          if nodo == nodo_inicial:
8              self.grafo.node(nodo, color='red')
9          else:
10             self.grafo.node(nodo, color='green')
11             self.grafo.render(f'/Imagenes/grafoprofundidad_{i}', format='png', view=False)
12             self.image_queue.put(f'/Imagenes/grafoprofundidad_{i}.png')
13             for vecino in self.vecinos.get(nodo, []):
14                 if vecino not in visitados:
15                     self.grafo.edge(nodo, vecino, color='red')
16                     dfs(vecino, visitados, nodo_inicial, i+1)
17             self.grafo.clear() # Limpiar el grafo antes de ejecutar DFS
18             vertices_ordenados = sorted(self.vertices)
19             aristas_ordenadas = sorted(self.aristas)
20             # Construir diccionario de vecinos
21             self.vecinos = {v: [] for v in vertices_ordenados}
22             for arista in aristas_ordenadas:
23                 a, b = arista.split("--")
24                 self.vecinos[a].append(b)
25                 self.vecinos[b].append(a)
26             print("Vertices ordenados:", vertices_ordenados)
27             print("Aristas ordenadas:", aristas_ordenadas)
28             if vertices_ordenados:
29                 nodo_inicial = vertices_ordenados[0]
30                 dfs(nodo_inicial, set(), nodo_inicial)
31             self.show_next_image()

```

Este método realiza una búsqueda en profundidad en el grafo y muestra el proceso paso a paso, generando imágenes de cada estado del grafo durante la búsqueda. Aquí está el desglose de su funcionamiento:

- Comprueba si el grafo está vacío. Si no hay vértices ni aristas en el grafo, muestra un mensaje de error y devuelve.
- Define una función interna `dfs` que realiza la búsqueda en profundidad recursivamente.
- Limpia el grafo antes de ejecutar la búsqueda en profundidad.
- Ordena los vértices y aristas del grafo.
- Construye un diccionario de vecinos donde cada clave es un vértice y el valor es una lista de sus vecinos.
- Si hay vértices en el grafo, el primer vértice ordenado se marca como el nodo inicial y se inicia la búsqueda en profundidad desde ese nodo.
- Llama al método `show_next_image` para mostrar la siguiente imagen en la cola de imágenes generadas durante el proceso de búsqueda en profundidad.

## 5. Visualización de las imágenes en la ejecución



```

1  def show_next_image(self):
2      if self.current_window is not None:
3          self.current_window.destroy()
4      if not self.image_queue.empty():
5          image_path = self.image_queue.get()
6          is_last = self.image_queue.empty()
7          self.show_image(image_path, is_last)

```

Este método se encarga de gestionar la visualización de las imágenes generadas durante la ejecución de los algoritmos en el grafo. Cada vez que se llama, muestra la siguiente imagen en la cola en una nueva ventana, si hay imágenes disponibles. Aquí está el desglose de lo que hace:

- Comprueba si hay una ventana de imagen actual abierta. Si es así, la destruye para limpiarla y prepararse para mostrar la siguiente imagen.
- Comprueba si la cola de imágenes no está vacía.
- Si hay una imagen en la cola, extrae la ruta de la imagen de la cola.
- Verifica si la imagen que se está mostrando es la última en la cola.
- Llama al método `show_image` para mostrar la imagen en una nueva ventana, pasando la ruta de la imagen y un indicador que indica si la imagen es la última en la cola.



## 6. Visualización de imágenes en cola

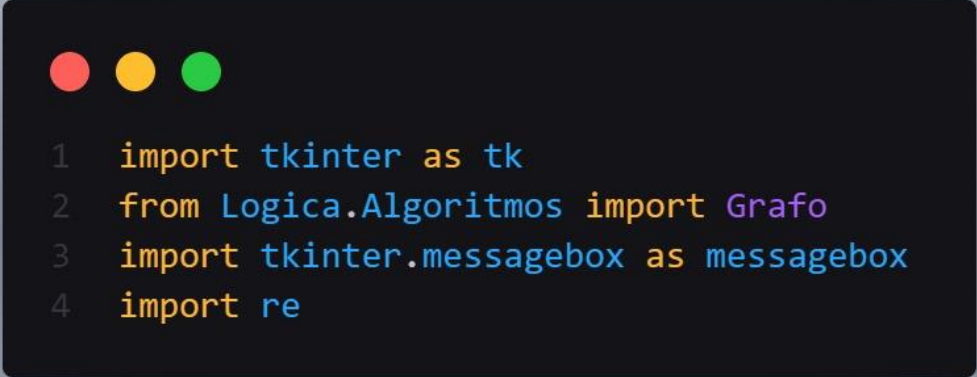
```
1 def show_image(self, image_path, is_last):
2     image = Image.open(image_path)
3     photo = ImageTk.PhotoImage(image)
4     self.current_window = tk.Toplevel(self.tk_widget)
5     self.current_window.geometry("+300+200")
6     label = tk.Label(self.current_window, image=photo)
7     label.image = photo
8     label.pack(side=tk.LEFT)
9     if not is_last:
10         self.current_window.after(2000, self.show_next_image)
11     self.current_window.protocol("WM_DELETE_WINDOW", self.show_next_image)
```

Este método se encarga de crear una nueva ventana y mostrar una imagen en ella, y también controla el avance automático a la siguiente imagen si hay más imágenes en la cola. Aquí está el desglose de lo que hace:

- Abre la imagen ubicada en la ruta `image_path` utilizando la función `Image.open()` del módulo PIL (Python Imaging Library).
- Crea un objeto `PhotoImage` utilizando la imagen abierta. Este objeto es necesario para mostrar la imagen en un widget de la interfaz gráfica.
- Crea una nueva ventana secundaria utilizando `Toplevel` de tkinter, asociada al widget principal (`self.tk_widget`). Esta ventana se guarda en el atributo `current_window` para poder referirse a ella más tarde.
- Establece la geometría (tamaño y posición) de la ventana secundaria en la pantalla.
- Crea un widget `Label` dentro de la ventana secundaria para mostrar la imagen, utilizando el objeto `PhotoImage` creado anteriormente.
- Empaqueta el widget `Label` en la ventana secundaria. Esto coloca el widget en la ventana y lo muestra.
- Si la imagen que se está mostrando no es la última en la cola de imágenes (`is_last` es `False`), programa una llamada al método `show_next_image` después de 2000 milisegundos (2 segundos). Esto permite avanzar automáticamente a la siguiente imagen después de un breve intervalo.
- Configura el comportamiento de cierre de la ventana secundaria. Si el usuario cierra la ventana, se llama al método `show_next_image`. Esto garantiza que se avance a la siguiente imagen incluso si el usuario cierra la ventana manualmente.

### 3. Interfaz

#### 1. Módulo



```
1 import tkinter as tk
2 from Logica.Algoritmos import Grafo
3 import tkinter.messagebox as messagebox
4 import re
```

Este fragmento de código importa los módulos necesarios para crear una interfaz gráfica, trabajar con grafos y mostrar mensajes de cuadro de diálogo en una aplicación. Importa los módulos necesarios para crear una interfaz gráfica de usuario (GUI) utilizando tkinter, trabajar con algoritmos de grafos desde un archivo llamado Algoritmos.py en el paquete Logica, y mostrar mensajes de cuadro de diálogo utilizando tkinter.messagebox. Aquí está el desglose:

- `import tkinter as tk`: Importa el módulo tkinter y lo renombra como tk, para utilizarlo en la creación de la interfaz gráfica.
- `from Logica.Algoritmos import Grafo`: Importa la clase Grafo desde el archivo Algoritmos.py dentro del paquete Logica. Esta clase probablemente contiene la lógica para trabajar con grafos y ejecutar algoritmos sobre ellos.
- `import tkinter.messagebox as messagebox`: Importa el módulo messagebox de tkinter, que proporciona funciones para mostrar cuadros de diálogo de mensaje en la interfaz gráfica.
- `import re`: Importa el módulo re, que proporciona operaciones de coincidencia de expresiones regulares en Python. Es posible que se use para realizar validaciones o manipulaciones de cadenas de texto en la interfaz gráfica.



## 1. Visualización del Grafo

```
1 class GrafoVisualizer(tk.Frame):
2     def __init__(self, root, *args, **kwargs):
3         tk.Frame.__init__(self, root, *args, **kwargs)
4         self.root = root
5         self.grafo = Grafo(self.root)
6         self.root.title("Visualizador de Grafos")
7         self.root.geometry("900x600") # Establecer el tamaño de la ventana
8
9         self.label_vertice = tk.Label(self.root, text="Ingrese un vértice:")
10        self.label_vertice.place(relx=0.25, rely=0.3, anchor='center')
11
12        self.textbox_vertice = tk.Entry(self.root)
13        self.textbox_vertice.place(relx=0.25, rely=0.35, anchor='center')
14
15        self.boton_agregar_vertice = tk.Button(self.root, text="Agregar Vértice", command=self.agregar_vertice)
16        self.boton_agregar_vertice.place(relx=0.25, rely=0.4, anchor='center')
17
18        self.label_arista = tk.Label(self.root, text="Ingrese una arista (en formato A--B):")
19        self.label_arista.place(relx=0.75, rely=0.3, anchor='center')
20
21        self.textbox_arista = tk.Entry(self.root)
22        self.textbox_arista.place(relx=0.75, rely=0.35, anchor='center')
23
24        self.boton_agregar_arista = tk.Button(self.root, text="Agregar Arista", command=self.agregar_arista)
25        self.boton_agregar_arista.place(relx=0.75, rely=0.4, anchor='center')
26
27        self.label_resultado_vertices = tk.Label(self.root, text="Vértices:")
28        self.label_resultado_vertices.place(relx=0.25, rely=0.5, anchor='center')
29
30        self.textbox_resultado_vertices = tk.Text(self.root, height=5, width=30)
31        self.textbox_resultado_vertices.place(relx=0.25, rely=0.55, anchor='center')
32
33        self.label_resultado_aristas = tk.Label(self.root, text="Aristas:")
34        self.label_resultado_aristas.place(relx=0.75, rely=0.5, anchor='center')
35
36        self.textbox_resultado_aristas = tk.Text(self.root, height=5, width=30)
37        self.textbox_resultado_aristas.place(relx=0.75, rely=0.55, anchor='center')
38
39        self.boton_visualizar_grafo = tk.Button(self.root, text="Visualizar Grafo", command=self.visualizar_grafo)
40        self.boton_visualizar_grafo.place(relx=0.33, rely=0.7, anchor='center')
41
42        self.boton_algoritmo_anchura = tk.Button(self.root, text="Algoritmo en Anchura", command=self.grafo.ejecutar_anchura)
43        self.boton_algoritmo_anchura.place(relx=0.5, rely=0.7, anchor='center')
44
45        self.boton_algoritmo_profundidad = tk.Button(self.root, text="Algoritmo en Profundidad", command=self.grafo.ejecutar_profundidad)
46        self.boton_algoritmo_profundidad.place(relx=0.67, rely=0.7, anchor='center')
47
```

Esta clase define una interfaz gráfica de usuario para interactuar con un grafo, permitiendo al usuario ingresar vértices y aristas, ver la lista de vértices y aristas, visualizar el grafo y ejecutar algoritmos sobre él. Aquí está el desglose de lo que hace:

- `def __init__(self, root, *args, **kwargs):`: Este es el método constructor de la clase `GrafoVisualizer`. Toma como argumento el widget raíz de tkinter (`root`) y cualquier otro argumento o palabra clave opcional. Dentro del constructor, se inicializa la interfaz gráfica y se configuran los elementos de la misma.
- `tk.Frame.__init__(self, root, *args, **kwargs)`: Llama al constructor de la clase base `tk.Frame` para inicializar el marco de la interfaz gráfica.
- `self.root = root`: Guarda una referencia al widget raíz de tkinter.
- `self.grafo = Grafo(self.root)`: Crea una instancia de la clase `Grafo` (presumiblemente definida en otro lugar) que se utilizará para realizar operaciones en el grafo.
- `self.root.title("Visualizador de Grafos")`: Establece el título de la ventana de la aplicación.
- `self.root.geometry("900x600")`: Establece las dimensiones de la ventana de la aplicación.

- A partir de aquí, se crean varios widgets de tkinter, como etiquetas (Label), entradas (Entry), botones (Button), y áreas de texto (Text), y se colocan en la interfaz gráfica utilizando el método place(). Estos widgets se utilizan para ingresar vértices y aristas, mostrar los vértices y aristas del grafo, y ejecutar algoritmos sobre el grafo.
- Los botones "Visualizar Grafo", "Algoritmo en Anchura" y "Algoritmo en Profundidad" tienen asociadas funciones para ejecutar acciones cuando se hace clic en ellos. Estas funciones llaman a los métodos visualizar\_grafo, ejecutar\_anchura y ejecutar\_profundidad del objeto Grafo respectivamente.

## 2. Agregar Vertice



```

1  def agregar_vertice(self):
2      vertice = self.textbox_vertice.get().lower() # Convertir a minúsculas
3      if not vertice: # Si el cuadro de texto está vacío
4          messagebox.showerror("Error", "El cuadro de texto está vacío.")
5          return
6      if vertice in self.grafo.vertices: # Si el vértice ya existe
7          messagebox.showerror("Error", "El vértice ya existe.")
8          return
9      self.grafo.vertices.append(vertice)
10     self.grafo.grafo.node(vertice)
11     print("Vertices:", self.grafo.vertices)
12     self.textbox_resultado_vertices.insert(tk.END, f"{vertice}\n")
13     self.textbox_vertice.delete(0, tk.END) # Limpiar el cuadro de texto
14

```

Este método permite al usuario agregar un vértice al grafo. Verifica que el vértice no esté vacío ni sea duplicado, lo agrega al grafo y actualiza la interfaz gráfica para reflejar el cambio. Aquí está el desglose de lo que hace:

- Obtiene el texto ingresado en el cuadro de entrada de vértice (textbox\_vertice) y lo convierte a minúsculas.
- Comprueba si el cuadro de texto está vacío. Si lo está, muestra un mensaje de error y devuelve.
- Comprueba si el vértice ya existe en la lista de vértices del grafo. Si es así, muestra un mensaje de error y devuelve.
- Agrega el vértice a la lista de vértices del grafo y lo añade como un nodo al grafo visual.
- Imprime la lista de vértices actualizada en la consola.
- Inserta el vértice agregado en el cuadro de texto textbox\_resultado\_vertices para mostrarlo al usuario.
- Limpia el cuadro de entrada de vértice para permitir que el usuario ingrese otro vértice.

### 3. Agregar Arista

```
1 def agregar_arista(self):
2     arista = self.textbox_arista.get()
3     if not arista: # Si el cuadro de texto está vacío
4         messagebox.showerror("Error", "El cuadro de texto está vacío.")
5         return
6     if not re.match(r"\w+--\w+", arista): # Si la entrada no sigue la estructura correcta
7         messagebox.showerror("Error", "La entrada no sigue la estructura correcta. (Ejemplo: A-B)")
8         return
9     a, b = arista.split("--")
10    if a.lower() == b.lower(): # Si los elementos de la arista son iguales
11        messagebox.showerror("Error", "Los elementos de la arista deben ser diferentes.")
12        return
13    if a.lower() not in self.grafo.vertices or b.lower() not in self.grafo.vertices: # Si los elementos de la arista no existen en los vértices
14        messagebox.showerror("Error", "Los elementos de la arista deben existir en los vértices.")
15        return
16    if {a.lower(), b.lower()} in [set(x.split("--")) for x in self.grafo.aristas]: # Si la arista ya existe en cualquier orden
17        messagebox.showerror("Error", "La arista ya existe.")
18        return
19    self.grafo.aristas.append(arista)
20    self.grafo.grafo.edge(a, b)
21    print("Aristas:", self.grafo.aristas)
22    self.textbox_resultado_aristas.insert(tk.END, f"{arista}\n")
23    self.textbox_arista.delete(0, tk.END) # Limpiar el cuadro de texto
24
```

Este método permite al usuario agregar una arista al grafo. Verifica que la arista cumpla con ciertos criterios (como el formato correcto y la no existencia previa), la agrega al grafo y actualiza la interfaz gráfica para reflejar el cambio. Aquí está el desglose de lo que hace:

- Obtiene el texto ingresado en el cuadro de entrada de arista (textbox\_arista).
- Comprueba si el cuadro de texto está vacío. Si lo está, muestra un mensaje de error y devuelve.
- Utiliza una expresión regular para verificar si la entrada sigue el formato correcto de una arista (por ejemplo, "A-B"). Si no sigue el formato correcto, muestra un mensaje de error y devuelve.
- Divide la entrada en dos vértices, a y b, separados por "--".
- Comprueba si los vértices a y b son iguales. Si lo son, muestra un mensaje de error y devuelve.
- Comprueba si los vértices a y b existen en la lista de vértices del grafo. Si alguno de ellos no existe, muestra un mensaje de error y devuelve.
- Comprueba si la arista ya existe en cualquier orden en la lista de aristas del grafo. Si ya existe, muestra un mensaje de error y devuelve.
- Agrega la arista a la lista de aristas del grafo y la agrega al grafo visual.
- Imprime la lista de aristas actualizada en la consola.
- Inserta la arista agregada en el cuadro de texto textbox\_resultado\_aristas para mostrarla al usuario.
- Limpia el cuadro de entrada de arista para permitir que el usuario ingrese otra arista.

#### 4. Visualización del grafo actual

```
1 def visualizar_grafo(self):
2     if not self.grafo.vertices and not self.grafo.aristas: # Si el grafo está vacío
3         messagebox.showerror("Error", "El grafo está vacío.")
4         return
5     self.grafo.grafo.render('/Imagenes/grafo', format='png', view=False)
6     self.grafo.show_image('/Imagenes/grafo.png', True)
```

Este método renderiza el grafo actual como una imagen PNG y lo muestra en la interfaz gráfica de usuario utilizando el método `show_image`. Si el grafo está vacío, muestra un mensaje de error en su lugar. Aquí está el desglose de lo que hace:

- Comprueba si el grafo está vacío, es decir, si no hay vértices ni aristas en el grafo. Si el grafo está vacío, muestra un mensaje de error y devuelve.
- Renderiza el grafo utilizando el método `render` del objeto grafo que se encuentra en la clase Grafo. El grafo se renderiza como un archivo de imagen PNG en la ubicación especificada `'/Imagenes/grafo.png'`. La opción `view=False` evita que el visor predeterminado se abra después de la renderización.
- Llama al método `show_image` del objeto grafo para mostrar la imagen renderizada del grafo en la interfaz gráfica de usuario. El segundo argumento `True` indica que esta es la última imagen en la cola de imágenes, por lo que no se programará la visualización de la siguiente imagen.