



**GRUPO 5**

# Manual Tecnico: Teoria de grafos

MATEMATICA PARA  
COMPUTACION 2



# Índice

- 01 **Introducción**
  - 02 **Objetivos**
  - 03 **Dirigido**
  - 04 **Especificación técnica**
  - 05 **Lógica del Programa**
  - 06 **Flujo de la Aplicación**
  - 07 **Requisitos de instalación**
  - 08 **Créditos**
-



# 1. Introduccion

Este proyecto es una aplicación gráfica desarrollada en Python utilizando las bibliotecas Tkinter y Graphviz para visualizar la creación y recorrido de grafos utilizando los algoritmos de Búsqueda en Anchura (BFS) y Búsqueda en Profundidad (DFS). Los usuarios pueden agregar vértices y aristas, visualizar el grafo, y ejecutar los algoritmos de recorrido BFS y DFS.



## 2. Objetivos

- Crear una interfaz gráfica interactiva para la manipulación de grafos.
- Implementar visualización gráfica del grafo original y su recorrido BFS y DFS.
- Facilitar la comprensión de los algoritmos BFS y DFS mediante la visualización de su ejecución paso a paso.



## 3. Dirigido

- Este programa está destinado principalmente a estudiantes y profesionales que deseen aprender o enseñar teoría de grafos, algoritmos de búsqueda, y su visualización en un entorno práctico y sencillo de utilizar.

## 4. Especificación Técnica

- .Lenguaje de programación: Python 3.8+
- Bibliotecas principales:
- Tkinter: Para la creación de la interfaz gráfica.
- PIL (Pillow): Para manejar imágenes en la interfaz.
- Graphviz: Para la generación visual del grafo.
- OS: Para gestionar rutas de archivos y directorios.
- Entorno de desarrollo: El código puede ejecutarse en cualquier entorno compatible con Python 3, y necesita tener instaladas las bibliotecas mencionadas.
- Sistema operativo compatible: Windows, macOS, y distribuciones Linux.

# 5. Lógica del Programa

El programa está dividido en dos clases principales:

## Clase GraphApp

Esta clase gestiona la interfaz gráfica y la interacción con el usuario. Aquí se permiten operaciones como agregar vértices, agregar aristas, visualizar el grafo, y ejecutar BFS o DFS.

Métodos principales:

**setup\_ui:** Configura la interfaz gráfica (botones, etiquetas, listbox, etc.).

```
1 def setup_ui(self):
2     self.root.geometry("600x600")
3
4     tk.Label(self.scrollable_frame, text="Vértice:", bg='lightblue', font=('Helvetica', 12)).grid(row=0, column=0)
5     self.vertex_entry.grid(row=0, column=1)
6     tk.Button(self.scrollable_frame, text="Agregar Vértice", command=self.add_vertex, bg='green', fg='white', font=('Helvetica', 10)).grid(row=0, column=2)
7
8     tk.Label(self.scrollable_frame, text="Arista (formato: A-B):", bg='lightblue', font=('Helvetica', 12)).grid(row=1, column=0)
9     self.edge_entry.grid(row=1, column=1)
10    tk.Button(self.scrollable_frame, text="Agregar Arista", command=self.add_edge, bg='green', fg='white', font=('Helvetica', 10)).grid(row=1, column=2)
11
12    tk.Button(self.scrollable_frame, text="Generar Grafo", command=self.render_graph, bg='blue', fg='white', font=('Helvetica', 10)).grid(row=2, column=1)
```

**add\_vertex:** Agrega un vértice al grafo y lo muestra en la interfaz.

```
1 def add_vertex(self):
2     vertex = self.vertex_entry.get()
3     if vertex:
4         self.graph.add_vertex(vertex)
5         self.vertex_listbox.insert(tk.END, vertex)
6         messagebox.showinfo("Éxito", f"Vértice {vertex} agregado.")
7     else:
8         messagebox.showerror("Error", "Debe ingresar un vértice.")
```

**add\_edge:** Agrega una arista entre dos vértices especificados.

```
1 def add_edge(self):
2     edge = self.edge_entry.get()
3     try:
4         u, v = edge.split('--')
5         self.graph.add_edge(u.strip(), v.strip())
6         self.edge_listbox.insert(tk.END, f"{u} -- {v}")
7         messagebox.showinfo("Éxito", f"Arista {u} -- {v} agregada.")
8     except ValueError:
9         messagebox.showerror("Error", "Formato incorrecto. Use A-B.")
10
```

**render\_graph:** Genera y muestra la imagen del grafo original.

```
1 def render_graph(self):
2
3     self.graph.render_graph('original')
4     self.show_image(os.path.join(ASSETS_DIR, 'original.png'), self.original_canvas)
5
```

**run\_bfs:** Ejecuta el algoritmo BFS desde un vértice inicial y muestra el resultado.

```
1 def run_bfs(self):
2     start_vertex = self.vertex_entry.get()
3     if start_vertex not in self.graph.graph:
4         messagebox.showerror("Error", "Vértice no encontrado.")
5         return
6     bfs_result = self.graph.bfs(start_vertex)
7     self.graph.render_graph('bfs', bfs_result)
8     self.show_image(os.path.join(ASSETS_DIR, 'bfs.png'), self.bfs_canvas)
```

**run\_dfs:** Ejecuta el algoritmo DFS desde un vértice inicial y muestra el resultado.

```
1 def run_dfs(self):
2     start_vertex = self.vertex_entry.get()
3     if start_vertex not in self.graph.graph:
4         messagebox.showerror("Error", "Vértice no encontrado.")
5         return
6     dfs_result = self.graph.dfs(start_vertex)
7     self.graph.render_graph('dfs', dfs_result)
8     self.show_image(os.path.join(ASSETS_DIR, 'dfs.png'), self.dfs_canvas)
```

**clear\_graph:** Limpia toda la información del grafo actual, permitiendo empezar desde cero.

```
1 def clear_graph(self):
2
3     self.vertex_entry.delete(0, tk.END)
4     self.edge_entry.delete(0, tk.END)
5     self.vertex_listbox.delete(0, tk.END)
6     self.edge_listbox.delete(0, tk.END)
```




# Clase Graph

Esta clase representa la estructura del grafo y contiene la implementación de los algoritmos BFS y DFS. También es responsable de la generación visual del grafo utilizando Graphviz.


Métodos principales:

**add\_vertex:** Añade un vértice al diccionario del grafo.



```
1 def add_vertex(self, v):
2     if v not in self.graph:
3         self.graph[v] = []
```

**add\_edge:** Añade una arista entre dos vértices (es un grafo no dirigido, por lo que las aristas se agregan en ambas direcciones).



```
1 def add_edge(self, u, v):
2     if u in self.graph and v in self.graph:
3         self.graph[u].append(v)
4         self.graph[v].append(u)
```

**bfs:** Implementa el algoritmo de Búsqueda en Anchura.

```
1 def bfs(self, start):
2
3     visited = set()
4     queue = deque([start])
5     bfs_order = []
6
7     while queue:
8         vertex = queue.popleft()
9         if vertex not in visited:
10             visited.add(vertex)
11             bfs_order.append(vertex)
12             queue.extend([neighbor for neighbor in self.graph[vertex] if neighbor not in visited])
13
14     return bfs_order
```

**bfs:** Implementa el algoritmo de Búsqueda en Anchura.

```
1 def dfs(self, start):
2
3     visited = set()
4     dfs_order = []
5     self._dfs_recursive(start, visited, dfs_order)
6     return dfs_order
```

**render\_graph:** Genera una imagen PNG del grafo actual utilizando la herramienta Graphviz.

```
1 def render_graph(self, name, result=None):
2     dot = graphviz.Graph()
3     for vertex in self.graph:
4         dot.node(vertex)
5     for vertex in self.graph:
6         for neighbor in self.graph[vertex]:
7             dot.edge(vertex, neighbor)
```

## 6. Flujo de la Aplicación

- El usuario inicia la aplicación desde la interfaz gráfica.
- El usuario ingresa los vértices y aristas para construir el grafo.
- Puede visualizar el grafo original.
- Puede ejecutar el algoritmo BFS o DFS desde un vértice específico.
- La aplicación genera imágenes de los grafos resultantes (original, BFS y DFS) que se muestran en la interfaz.
- En cualquier momento, el usuario puede limpiar el grafo para empezar de nuevo.

## 7. Requisitos de instalacion

- Instalar Python 3.8+ desde el sitio oficial:  
<https://www.python.org/downloads/>
- Instalar las bibliotecas necesarias:
- `pip install tkinter pillow graphviz`
- Descargar e instalar Graphviz desde:  
<https://graphviz.org/download/>
- Configurar la variable de entorno de Graphviz en el sistema operativo, asegurando que la herramienta dot sea accesible desde la terminal.

# 8. Credits:

- Desarrolladores:  
Silvia María Mejía Fernández,  
Jancarlo Giovanni Schwarz,  
Oliver Eduardo Toc Mateos y  
Byron Manuel Hernández López
- Fecha de entrega: 25/10/2024