Manual Técnico Generador de Grafos

Nombres y Carné:

Byron Miguel Galicia Hernandez Carné: 20190717

Jorge Estuardo Pumay Soy Carne: 201213421

Sección: A-

Laboratorio Lenguajes Formales de programación

Introducción

El programa generador de grafo permite al usuario observar de manera grafica el uso y funcionamiento de los algoritmos de búsqueda en profundidad y búsqueda a lo ancho. Permitiendo graficar los vértices y aristas que el usuario final ingrese en el programa.

Objetivos

- Mostrar gráficamente un grafo formado por n cantidad de vértices y aristas ingresados por el usuario
- Brindar una herramienta que permita al usuario generar un grafo y seleccionar un algoritmo de búsqueda, ya sea a lo ancho o en profundidad.

Requisitos del sistema

- CPU: Core I3 en adelante, AMD o Ryzen 3 en adelante
- RAM: 1 GB en adelante para optimas condiciones del sistema
- Espacio de almacenamiento: 200 KB en adelante
- Sistema operativo: Windows XP en adelante, Linux o Mac OS. Requisito indispensable contar con librería de Python instalada en el sistema.

Programa generador de Grafos

1. **Main.py**: ventana inicial que importa la clase Interfaz para inicializar el programa y mostrar la interfaz grafica del sistema en el ordenador del usuario

```python
from interfaz.Interfaz import Interfaz
import tkinter as tk

def main():
    ventana=tk.Tk()
    ventanaInterfaz=Interfaz(ventana)
    ventana.mainloop()

if __name__ == '__main__':
    main()
```

2. **Interfaz.py:** archivo de Python que contiene las funciones necesarias para la visualización de los datos almacenados en el programa y la lógica del mismo.

2.1 Librerias utilizadas:

2.1.1 Tkinter: utilizada para generar la interfaz grafica

2.1.2 Graphviz: utilizada para manipular y generar los grafos

2.1.3 Networkx: se utiliza para construir y manipular los grafos

2.1.4 Matplotlib: se utiliza para visualizar gráficamente el grafo generado

```python
#clases
from clases.Grafo import Grafo

#librerias
import tkinter as tk
from tkinter import messagebox
from tkinter import ttk
from graphviz import Source
from PIL import Image
import io
import networkx as nx

from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure
```

2.2 cargarWidgets(): esta función define los objetos de la interfaz grafica al igual que establece la funcionalidad de cada botón asociando estos con las diferentes funciones de la clase interfaz

```
def cargarWidgets(self):···
```

2.3 agregarVertice(): esta función almacena un vértice de tipo objeto Grafo y lo muestra en el cuadro de texto grafo mediante las funciones generarGrafo() y dibujarGrafo()

```
def agregarVertice(self):
    arista=self.txtVertice.get()

    if arista:
        self.grafo.agregar_vertice(arista)
        self.txtVertice.delete(0, "end")

        self.txtListaVertices.delete(0,tk.END)

        #llena el listbox con los vertices
        lista_vertices=self.grafo.obtener_vertices()
        for vertice in lista_vertices:
            self.txtListaVertices.insert("end",vertice)

        self.generarGrafo()
        self.dibujargrafo()
    else:
        messagebox.showerror("Error", "No se ingresó ningun vertice")
```

2.4 agregarAristas(): esta función agrega una nueva aristas posterior a haber agregado los vértices y lo representa gráficamente en el cuadro de texto de Grafo. Sino se ingresan ambos vértices previamente definidos muestra un error en pantalla.

```
def agregarArista(self):
    vertice1=self.txtVertice1.get()
    vertice2=self.txtVertice2.get()

    if vertice1 and vertice2:
        self.grafo.agregar_arista(vertice1,vertice2)
        self.txtVertice1.delete(0, "end")
        self.txtVertice2.delete(0, "end")
        self.txtListaAristas.delete(0,tk.END)

        lista_aristas=self.grafo.obtener_aristas()

        for arista in lista_aristas:
            self.txtListaAristas.insert("end",arista[0]+"--"+arista[1])

        self.generarGrafo()
        self.dibujargrafo()
    else:
        messagebox.showerror("Error", "vertice1 o vertice2 estan vacios")
```

2.5 generarGrafo() y dibujarGrafo(): estas funciones utilizan la librería de Networkx que utilizan los métodos de draw() y dra_networkx_ para construir y manipular los grafos generados

```
def generarGrafo(self):
    lista_vertices=self.grafo.obtener_vertices()
    lista_aristas=self.grafo.obtener_aristas()

    for vertice in lista_vertices:
        self.grafoG.add_node(vertice)

    for arista in lista_aristas:
        self.grafoG.add_edge(arista[0],arista[1])

def dibujargrafo(self):
    self.ax.clear()
    nx.draw(self.grafoG, ax=self.ax, with_labels=True)
    self.canvas.draw()
```

2.6 busquedaAnchura(): esta función primeramente ordena los vértices de manera ascendente para encontrar el vertice inicial, luego accede al Grafo previamente generado para recorrer las aristas en ambas direcciones evitando crear un ciclo. Las aristas previamente recorridas son coloreadas de amarillo al igual que los vértices para representar el árbol generado.

```
def busquedaAnchura(self):
    lista_vertices=self.grafo.obtener_vertices()
    lista_vertices_ordenada=sorted(lista_vertices)
    #inicia siempre en orden ascendente por el vertice que esta primero en la lista
    vertice_inicial=lista_vertices_ordenada[0]

    self.axAnchura.clear()
    bfs_edges=list(nx.bfs_edges(self.grafoG,source=vertice_inicial))
    pos = nx.spring_layout(self.grafoG)
    nx.draw(self.grafoG, pos=pos, ax=self.axAnchura, with_labels=True)
    nx.draw_networkx_edges(self.grafoG, pos=pos, edgelist=bfs_edges, edge_color='#FFF05F', ax=self.axAnchura)
    nx.draw_networkx_nodes(self.grafoG, pos=pos, nodelist=[vertice_inicial]+[v for u, v in bfs_edges], node_color='#FFF05F', ax=self.axAnchura)
    nx.draw_networkx_nodes(self.grafoG, pos=pos, nodelist=[vertice_inicial], node_color='#d9dfff', ax=self.axAnchura)
    self.canvasAnchura.draw()
```

2.7 busquedaProundidad(): esta función primeramente ordena los vértices de manera ascendente para encontrar el vertice inicial, luego accede al Grafo previamente generado para recorrer las aristas una a la vez evitando crear un ciclo y recorriendo todos los caminos posibles para llegar del vertice al vertice final. Las aristas previamente recorridas son coloreadas de verde al igual que los vértices para representar el árbol generado.

```
def busquedaProdundidad(self):
    lista_vertices=self.grafo.obtener_vertices()
    lista_vertices_ordenada=sorted(lista_vertices)
    #inicia siempre en orden ascendente por el vertice que esta primero en la lista
    vertice_inicial=lista_vertices_ordenada[0]

    self.axAnchura.clear()
    dfs_edges=list(nx.dfs_edges(self.grafoG,source=vertice_inicial))
    pos = nx.spring_layout(self.grafoG)
    nx.draw(self.grafoG, pos=pos, ax=self.axAnchura, with_labels=True)
    nx.draw_networkx_edges(self.grafoG, pos=pos, edgelist=dfs_edges, edge_color='#4CE27F', ax=self.axAnchura)
    nx.draw_networkx_nodes(self.grafoG, pos=pos, nodelist=[vertice_inicial]+[v for u, v in dfs_edges], node_color='#4CE27F', ax=self.axAnchura)
    nx.draw_networkx_nodes(self.grafoG, pos=pos, nodelist=[vertice_inicial], node_color='#d9dfff', ax=self.axAnchura)
    self.canvasAnchura.draw()
```

3. **Grafo.py:** archivo de Python que almacena las vértices y aristas que el usuario ingresa por medio de la interfaz grafica. Los vértices se almacen mediante una lista en self.vertices

```
from tkinter import messagebox

class Grafo:
    def __init__(self):
        self.vertices = {}
```

3.1 agregar_vertice: función que permite almacenar un vertice, si en dado caso el vertice ya existe, se mostrara un mensaje de error que indique que ya existe

```
def agregar_vertice(self, vertice):
    if vertice not in self.vertices:#si el vertice no existe, lo grega si no marca error
        self.vertices[vertice] = []

        print("sea gregó el vertice",vertice)
    else:
        messagebox.showerror("Error", "El vertice ya existe" )
```

3.2 agregar_arista: esta función realiza una búsqueda en el listado de vértices almacenados, si encuentra el vertice origen y destino, entonces agrega la arista al listado de vértices pero con 2 atributos: vertice origen y vertice destino.

```
def agregar_arista(self, origen, destino):
    #si encuentra el vertice oriegn y destino agrega la arista si no marca error
    if origen in self.vertices and destino in self.vertices:
        self.vertices[origen].append(destino)
        #self.vertices[destino].append(origen)
    else:
        messagebox.showerror("Error", "No se encontró el vertice origen o destino")
```

3.3 obtener_vertices y obtener_aristas: la función de obtener vértices devuelve el valor almacenado en cada posición siendo este las letras de cada objeto en esa lista. Y obtener_aristas() utiliza un ciclo for que recorre cada vertice que esta previamente almacenado con la posición origine y posición destino para poder posteriormente almacenarlas en una lista definida dentro de la misma funcion

```
def obtener_vertices(self):
    return list(self.vertices.keys())

def obtener_aristas(self):
    aristas = []
    for vertice, adyacentes in self.vertices.items():
        for adyacente in adyacentes:
            aristas.append((vertice, adyacente))
    return aristas
```