UNIVERSIDAD AMERICANA

Facultad de Ingeniería y Arquitectura

ALGORITMOS Y ESTRUCTURAS DE DATOS - GRUPO 2

Código #1

Nombre:

- Freddy Adrian peralta
- Jose Gabriel Cano Blandón
- Marcos Alessandro Lagos Rivera
- Axell Antonio Castillo Tapia

Docente

- Silvia

Managua, Nicaragua
15 de Junio del 2024

Informe de Desarrollo – Práctica de Grafos

Objetivo General

Implementar un grafo en Python utilizando una estructura modular, incluyendo operaciones básicas, recorridos y análisis de conectividad, con interacción a través de consola.

1. Diseño General del Programa

Para abordar la práctica, se optó por una estructura modular, dividiendo el código en dos archivos:
- grafo.py: Contiene la clase Grafo con todos los métodos necesarios para representar y manipular grafos.
- main.py: Proporciona una interfaz de usuario por consola, con menús interactivos para facilitar la ejecución de pruebas.
Esta separación permite mantener el código organizado y facilita la reutilización y mantenimiento del mismo.

2. Implementación de la Clase Grafo

La clase Grafo está diseñada para soportar grafos dirigidos y no dirigidos. Se eligió como estructura base un diccionario de listas de adyacencia, ya que es eficiente tanto en tiempo como en espacio.
Se implementaron los siguientes métodos:
- __init__(es_dirigido=False): Crea un grafo vacío.
- agregar_vertice(vertice): Agrega un vértice al grafo.
- agregar_arista(u, v, peso=1): Añade una arista entre dos vértices.
- obtener_vecinos(vertice): Devuelve una lista de vecinos del vértice.
- existe_arista(u, v): Verifica si existe una conexión entre dos nodos.
- mostrar_grafo(): Imprime el grafo como lista de adyacencia.

3. Recorridos

Se desarrollaron dos algoritmos clásicos:
- BFS (Breadth-First Search): Implementado con collections.deque para eficiencia. Devuelve los vértices en orden de visita desde un nodo inicial.

- DFS (Depth-First Search): Implementado de forma recursiva. Ideal para explorar caminos más profundos rápidamente.
Ambos algoritmos ayudan a analizar la estructura del grafo.

4. Análisis de Conectividad y Rutas

Se agregaron dos funciones adicionales:
- es_conexo(): Determina si el grafo no dirigido está conectado.
- encontrar_camino(inicio, fin): Devuelve una lista de nodos que representan un camino simple entre dos vértices.

5. Validaciones y Experiencia de Usuario

En main.py se incluyeron las siguientes validaciones:
- Verificación de entradas vacías.
- Manejo de errores cuando un vértice no existe.
- Entrada de vértices separados por espacios.
- Aristas opcionalmente ponderadas (u v peso).
El usuario puede elegir entre grafo dirigido o no dirigido al iniciar el programa.

6. Pruebas Realizadas

Se hicieron pruebas con grafos conectados y desconectados. También se evaluó la búsqueda de caminos entre vértices no conectados y la ejecución de recorridos a partir de vértices inexistentes.
Ejemplos probados:
- Vértices: A B C D E F
- Aristas: A B, A C, B D, C D, D E
- Recorridos: BFS('A'), DFS('A')
- Camino: de A a E, de A a F

7. Conclusiones

- La separación en módulos mejora la claridad y escalabilidad del código.
- Las listas de adyacencia son adecuadas para grafos dispersos.
- El uso de estructuras eficientes como deque mejora el rendimiento del BFS.
- Las validaciones permiten una mejor experiencia al usuario final.
- El grafo modelado es una excelente base para implementar algoritmos más complejos.