Universidad Autónoma de Guadalajara
Powered by Arizona State University

Nombre: Luis Alberto Vargas González

Fecha: 16/06/2025

Actividad y unidad: U3 A1 Implementación de DFS

Clase: Diseño y Análisis de Algoritmos.

Maestría en Ciberseguridad.

Introducción.

En esta práctica de programación se realizó un algoritmo de DFS, búsqueda profunda en donde nuestro principal objetivo es encontrar nodos que están conectados entre sí, dado su representación en conjuntos, podemos construir su representación lógica en el CPU y por ende así podemos identificar que nodos están conectados, cuales no y también representar si hay un conjunto de nodos conectados o más de uno, llamados componentes, el cual es el caso de este programa.

Cabe recalcar que la forma de representación que se hace en el código del grafo que nos fue proporcionado este dado en lista de adyacencia. La cual es la que nos dará de forma más visual que nodos están conectados entre sí, y nos representa cuantos componentes hay en los nodos.

Código fuente.

```
# Cargar datos del grafo
nodes = list(range(50))
edges = [
    (0, 29), (0, 46), (0, 21), (0, 14), (0, 38), (0,
31), (1, 41), (1, 31), (1, 21), (1, 17),
    (2, 9), (2, 26), (2, 5), (2, 25), (2, 4), (3, 18),
(3, 30), (3, 47), (4, 28), (4, 9),
    (4, 8), (5, 44), (5, 12), (6, 37), (6, 10), (7,
23), (7, 22), (7, 39), (9, 19), (9, 28),
    (9, 27), (11, 33), (13, 25), (13, 38), (13, 29),
(14, 26), (14, 28), (14, 39), (15, 22),
    (15, 31), (15, 19), (15, 41), (16, 46), (16, 26),
(16, 38), (16, 27), (17, 40), (17, 29),
    (18, 45), (18, 42), (18, 35), (18, 33), (18, 47),
(20, 36), (20, 49), (20, 42), (22, 26),
    (22, 34), (23, 31), (23, 32), (23, 40), (24, 31),
(24, 44), (25, 38), (26, 31), (27, 32),
    (29, 48), (29, 41), (30, 47), (30, 37), (33, 36),
(33, 49), (34, 48), (35, 45), (36, 45),
    (37, 49), (37, 45), (37, 47), (38, 41), (40, 48),
(41, 44), (42, 49), (43, 48), (45, 47)
]
```

```python
# Crear representación en lista de adyacencia
from collections import defaultdict

graph = defaultdict(list)
for u, v in edges:
    graph[u].append(v)
    graph[v].append(u)  # porque es un grafo no dirigido

# Algoritmo DFS para hallar un componente conexo
def dfs(node, visited, component):
    visited.add(node)
    component.append(node)
    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs(neighbor, visited, component)

# Hallar todos los componentes conexos
visited = set()
components = []

for node in nodes:
    if node not in visited and node in graph:
        component = []
```

```
        dfs(node, visited, component)
        components.append(component)

# Mostrar los componentes(Agrupaciones de nodos
conexos))
for i, comp in enumerate(components):
    print(f"Componente {i + 1}: {sorted(comp)}")
```

De manera breve explicaré el código: Al principio del código tenemos la carga de datos del grafo, que previamente se nos proporciona, y esto es muy importante ya que aquí definimos la cantidad de nodos que hay en el grafo, en este caso 50 y además definimos mediante una lista de tuplas, ya que esto nos permite contener dentro de la misma estructura (lista)todas las aristas que se representan como tuplas para permitirnos poder identificar esas aristas y que Python no las “mezcle “ y entienda las aristas como un solo bloque.

Cabe aclarar que se usó la librería collections ya que esta contiene defaultdict que es una estructura de datos llamada diccionario, pero en lugar de ser un diccionario tradicional como el que tiene Python que se llama dict, esta estructura nos permite indexar automáticamente en caso de no existir previamente en el diccionario un nodo, o un par de ellos y evitar escribir esto:

graph = {}

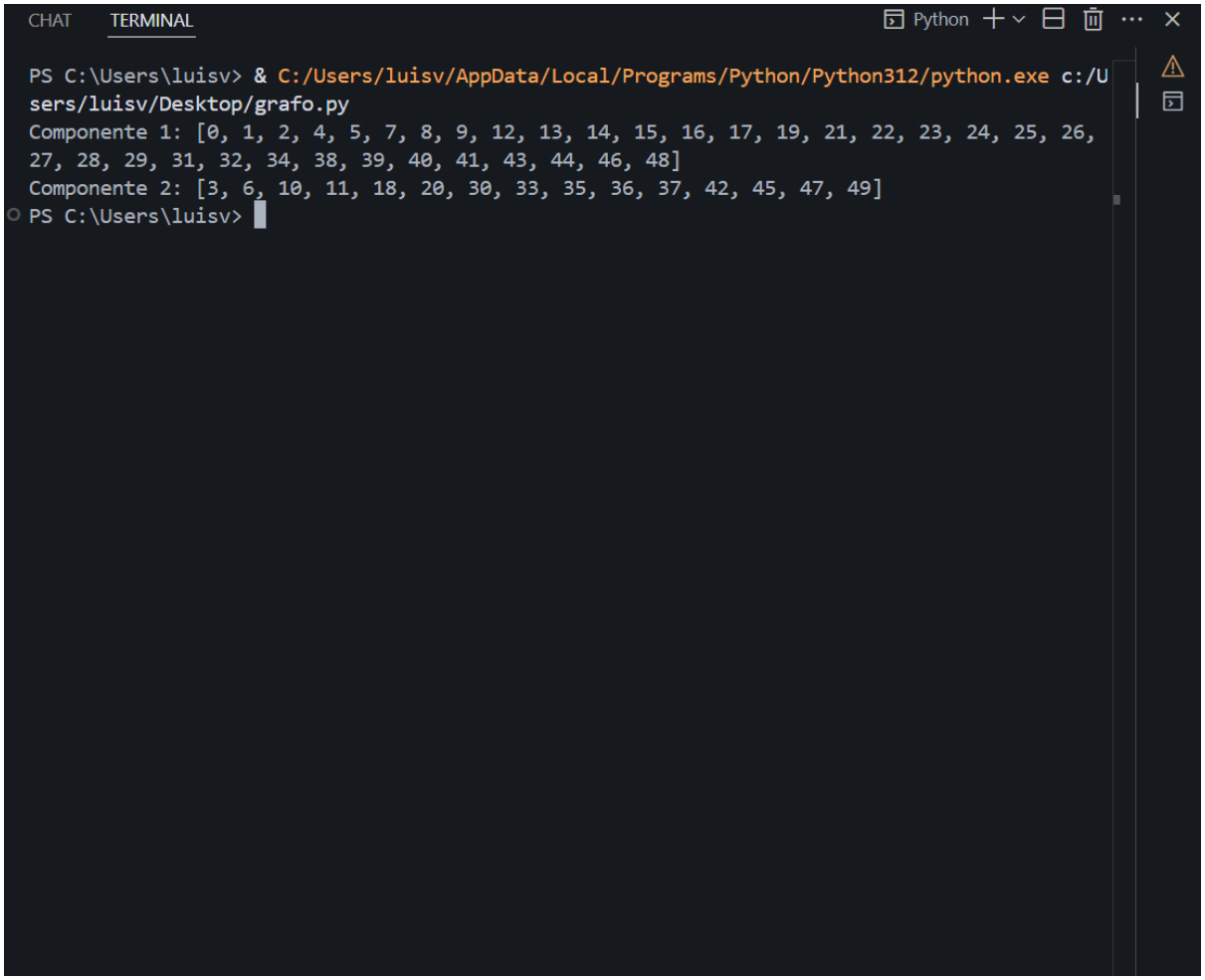
for u, v in edges:

```
    if u not in graph:
        graph[u] = []
    if v not in graph:
        graph[v] = []
    graph[u].append(v)
    graph[v].append(u)
```

Con lo cual nos ahorramos código y condicionales.

Justo después de esto es donde encontramos la función de DFS, la cual es la que nos permite poder encontrar los nodos conectados entre sí simplemente recorriendo cada nodo y añadiéndolo con el método append() al componente donde pertenezca y en caso de que un nodo vecino no este añadido se añade a la lista de visitados para posteriormente evaluar si dado que un nodo no está en el set de visitados, pero si está en el grafo , esto nos ayuda a construir el 2do componente , lo cual es vital para este ejercicio porque tenemos un grafo no dirigido.

Finalmente recorremos con un ciclo for los nodos en cada componente, para poder mostrarlos por pantalla y poder diferencias los componentes de manera clara.


CHAT
TERMINAL
Python
PS C:\Users\luisv> & C:/Users/luisv/AppData/Local/Programs/Python/Python312/python.exe c:/Users/luisv/Desktop/grafo.py
Componente 1: [0, 1, 2, 4, 5, 7, 8, 9, 12, 13, 14, 15, 16, 17, 19, 21, 22, 23, 24, 25, 26, 27, 28, 29, 31, 32, 34, 38, 39, 40, 41, 43, 44, 46, 48]
Componente 2: [3, 6, 10, 11, 18, 20, 30, 33, 35, 36, 37, 42, 45, 47, 49]
PS C:\Users\luisv>

Conclusión.

Podemos argumentar que; la realización de esta práctica fue muy sencilla ya que el tema de grafos es uno de los más sencillo dentro de la estructura de datos, claro que esto no significa que no tenga su cierta complejidad pues como pudimos notar aquí tenemos un grafo medianamente grande, pero para poder recorrer profundamente un grafo más grande (alguno que tenga más de 1000 nodos) como puede ser una red de telecomunicaciones empresarial o corporativa donde podemos tener incluso billones de nodos como es la red de internet actual. Y el cálculo se intensifica a mayor cantidad de nodos, componentes que diferenciar y claro, entre más acciones en el grafo realicemos