CSL 325
Artificial Intelligence Lab

Department of Computer Science
Bahria University, Islamabad

Lab # 6: DFS, LDS, and IDS

Objectives:

1. Understand and implement **DFS**, **BFS**, and **UCS** algorithms.
2. Compare the differences between these algorithms.
3. Solve problems using these algorithms (e.g., finding a path in a graph).

Tools Used:

Spyder

Lab Tasks:

This same graph can be used for **BFS**, **DFS**, and **UCS** by adjusting how the traversal or cost is handled.

Nodes

A, B, C, D, E, F, G

Edges (with weights)

From	To	Weight
A	B	2
A	C	3
B	D	1
B	E	3
C	F	4
D	G	5
E	G	2
F	G	1

Adjacency Representation (for BFS & DFS)

A → [B, C]
B → [D, E]
C → [F]
D → [G]
E → [G]
F → [G]
G → []

Weighted Graph (for UCS)

A-B (2), A-C (3)
B-D (1), B-E (3)
C-F (4)
D-G (5)
E-G (2)
F-G (1)

Start Node: A, **Goal Node:** G

Task 1: Depth-First Search (DFS)

Implement a Depth-First Search (DFS) algorithm to traverse a graph and find a path from a start node to a goal node. The graph is represented as an adjacency list. Your implementation should return the path from the start node to the goal node if it exists, or indicate that no path is found.
Create a directed or undirected graph using NetworkX. Print the order in which nodes are visited.

Task 2: BFS

- Create a directed or undirected graph using **NetworkX**.
- Implement the **BFS algorithm** to traverse from a given starting node.
- Print the order in which nodes are visited.

For Task 1 and Task 2

- Plot the **initial graph** before traversal using **Matplotlib**.
- Ensure node labels and edge directions are visible.
- Indicate the **starting node** visually.

Task 3: UCS

- Create a **weighted graph** using **NetworkX**.
- Implement **UCS** to find the least-cost path between a start and goal node.
- Print the total cost and the path found.

Weighted Graph Plot

- Plot the **initial weighted graph** using **Matplotlib**.
- Display the **edge weights** on the plot.
- Highlight the **start** and **goal** nodes distinctly.

'''

Task 1: Depth-First Search (DFS)

Implement a Depth-First Search (DFS) algorithm to traverse a graph and find a path from a start node to a goal node. The graph is represented as an adjacency list. Your implementation should return the path from the start node to the goal node if it exists, or indicate that no path is found. Create a directed or undirected graph using NetworkX. Print the order in which nodes are visited.

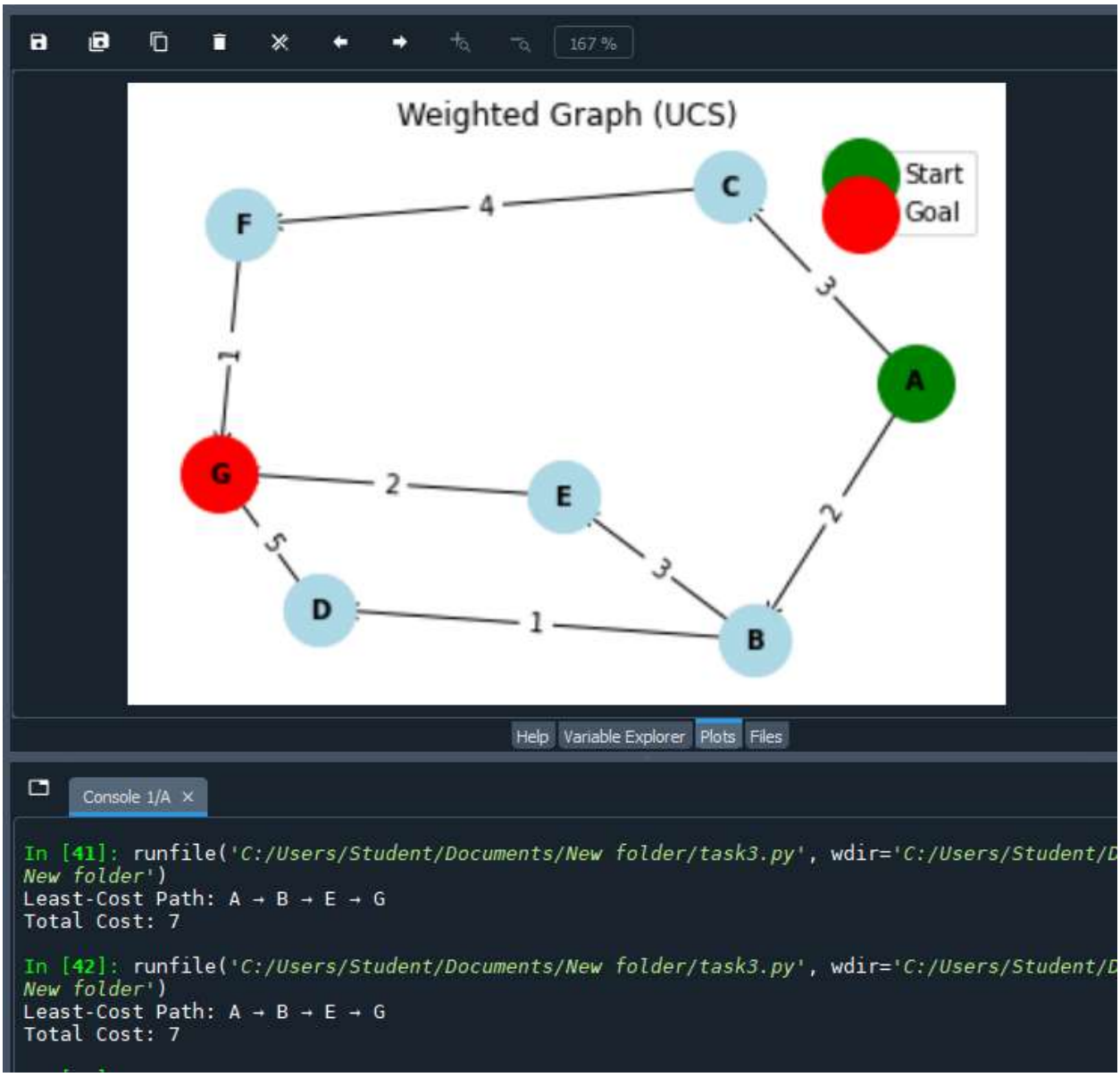
'''

```
import networkx as nx
import matplotlib.pyplot as plt


graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': ['G'],
    'E': ['G'],
    'F': ['G'],
```

```
    'G': []
}

start = 'A'
goal = 'G'

#
G = nx.DiGraph()
for node, neighbors in graph.items():
    for neighbor in neighbors:
        G.add_edge(node, neighbor)

pos = nx.spring_layout(G, seed=42)

plt.figure(figsize=(6, 4))
nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=1200,
font_weight='bold', arrows=True)
nx.draw_networkx_nodes(G, pos, nodelist=[start], node_color='green', label='Start')
nx.draw_networkx_nodes(G, pos, nodelist=[goal], node_color='red', label='Goal')
plt.title(" (DFS)")
plt.legend()
plt.show()
```

```
def dfs(graph, start, goal):
    stack = [(start, [start])]
    visited = []

    while stack:
        node, path = stack.pop()
        if node not in visited:
            visited.append(node)
            if node == goal:
                return path, visited

            for neighbor in reversed(graph[node]):
                stack.append((neighbor, path + [neighbor]))
    return None, visited



dfs_path, dfs_visited = dfs(graph, start, goal)

print("Task1: DEPTH-FIRST SEARCH ")
print("Visited Order:", " -> ".join(dfs_visited))
if dfs_path:
    print("Path Found:", " -> ".join(dfs_path))
else:
```

print("No path found")



""

Task 2: BFS

- Create a directed or undirected graph using NetworkX.
- Implement the BFS algorithm to traverse from a given starting node.
- Print the order in which nodes are visited.

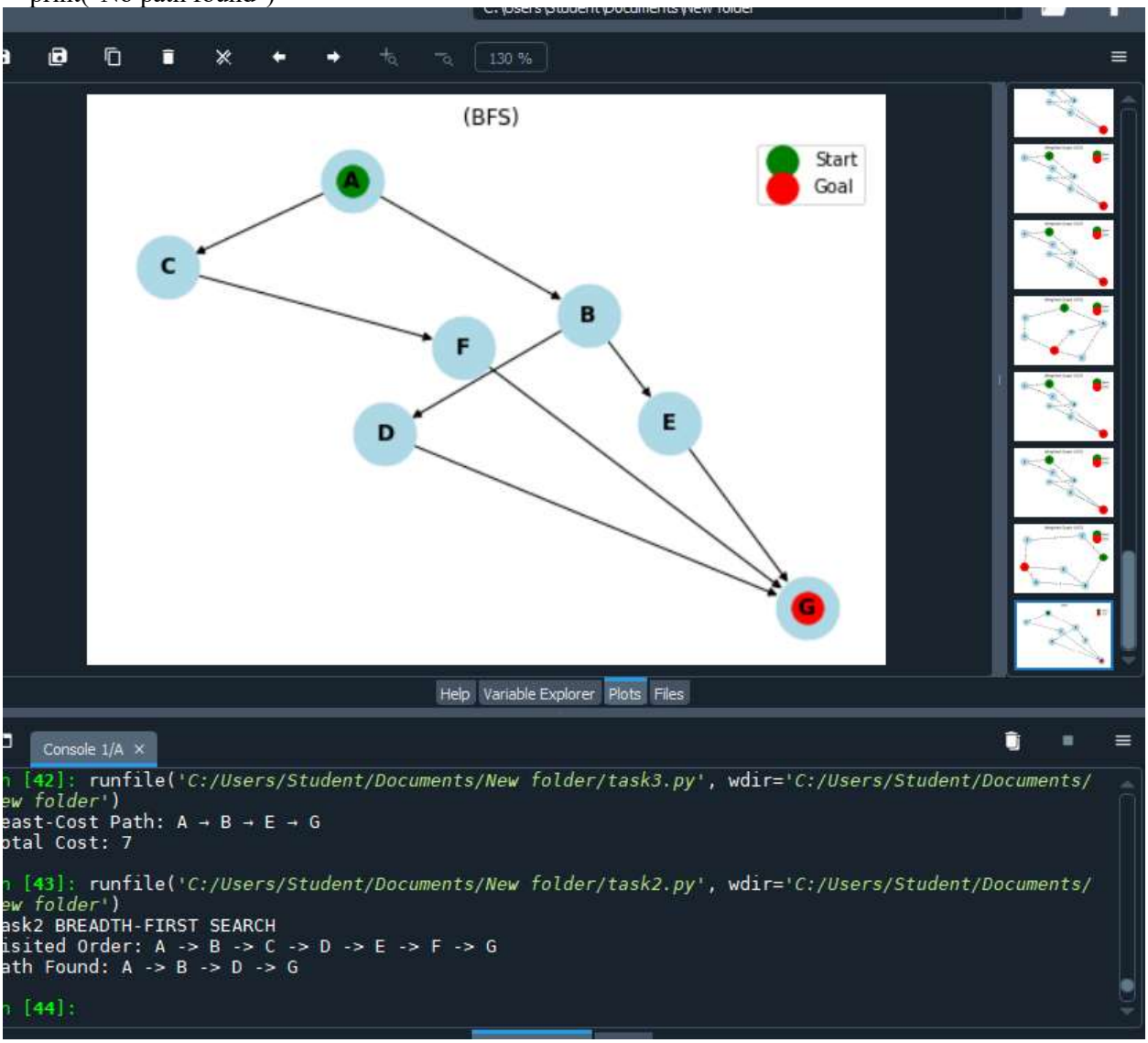
'''

```
import networkx as nx
import matplotlib.pyplot as plt
from collections import deque



graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': ['G'],
    'E': ['G'],
    'F': ['G'],
    'G': []
}
```

```
start = 'A'
goal = 'G'

G = nx.DiGraph()
for node, neighbors in graph.items():
    for neighbor in neighbors:
        G.add_edge(node, neighbor)

pos = nx.spring_layout(G, seed=42)

plt.figure(figsize=(6, 4))
nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=1200,
font_weight='bold', arrows=True)
nx.draw_networkx_nodes(G, pos, nodelist=[start], node_color='green', label='Start')
nx.draw_networkx_nodes(G, pos, nodelist=[goal], node_color='red', label='Goal')
plt.title(" (BFS)")
plt.legend()
plt.show()

def bfs(graph, start, goal):
    queue = deque([(start, [start])])
    visited = []

    while queue:
```

```
        node, path = queue.popleft()
        if node not in visited:
            visited.append(node)
            if node == goal:
                return path, visited
            for neighbor in graph[node]:
                queue.append((neighbor, path + [neighbor]))
    return None, visited


bfs_path, bfs_visited = bfs(graph, start, goal)

print("Task2 BREADTH-FIRST SEARCH ")
print("Visited Order:", " -> ".join(bfs_visited))
if bfs_path:
    print("Path Found:", " -> ".join(bfs_path))
else:
```

print("No path found")



'''

Task 1: Depth-First Search (DFS)

Implement a Depth-First Search (DFS) algorithm to traverse a graph and find a path from a start node to a goal node. The graph is represented as an adjacency list. Your implementation should return the path from the start node to the goal node if it exists, or indicate that no path is found. Create a directed or undirected graph using NetworkX. Print the order in which nodes are visited.

'''

```
import networkx as nx
import matplotlib.pyplot as plt


graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': ['G'],
    'E': ['G'],
    'F': ['G'],
```

```
    'G': []
}

start = 'A'
goal = 'G'

#
G = nx.DiGraph()
for node, neighbors in graph.items():
    for neighbor in neighbors:
        G.add_edge(node, neighbor)

pos = nx.spring_layout(G, seed=42)

plt.figure(figsize=(6, 4))
nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=1200,
font_weight='bold', arrows=True)
nx.draw_networkx_nodes(G, pos, nodelist=[start], node_color='green', label='Start')
nx.draw_networkx_nodes(G, pos, nodelist=[goal], node_color='red', label='Goal')
plt.title(" (DFS)")
plt.legend()
plt.show()
```

```
def dfs(graph, start, goal):
    stack = [(start, [start])]
    visited = []

    while stack:
        node, path = stack.pop()
        if node not in visited:
            visited.append(node)
            if node == goal:
                return path, visited

            for neighbor in reversed(graph[node]):
                stack.append((neighbor, path + [neighbor]))
    return None, visited



dfs_path, dfs_visited = dfs(graph, start, goal)

print("Task1: DEPTH-FIRST SEARCH ")
print("Visited Order:", " -> ".join(dfs_visited))
if dfs_path:
    print("Path Found:", " -> ".join(dfs_path))
else:
```

print("No path found")