Lab 5

22/07/2024

Brief Summary:

The script addresses the following tasks and problems:

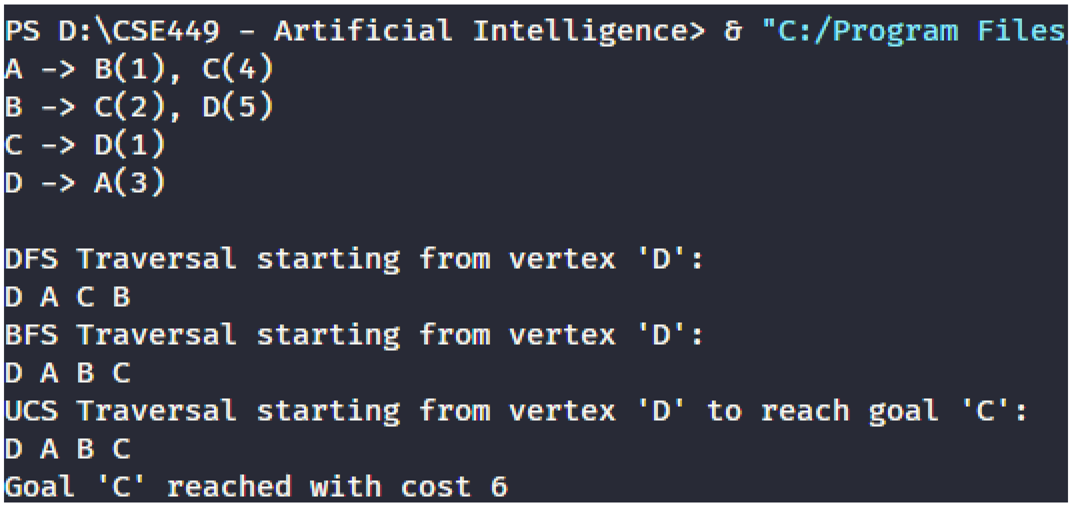
- **Depth-First Search (DFS)**
    - Uses a stack to explore nodes.
    - Goes as deep as possible before backtracking.
    - Prints nodes as they are visited.
- **Breadth-First Search (BFS)**
    - Uses a queue to explore nodes level by level.
    - Ensures that all nodes at the present depth are visited before moving on to nodes at the next depth level.
    - Prints nodes as they are visited.
- **Uniform Cost Search (UCS)**
    - Uses a priority queue (min-heap) to explore nodes.
    - Explores the least costly path first, ensuring that the path with the minimum cumulative cost is chosen.
    - Prints nodes as they are visited and announces when the goal is reached with the associated cost.

Code:

```
from collections import deque
import heapq

def dfs(graph, start):
    visited = set()
    stack = [start]
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            print(vertex, end=" ")
            visited.add(vertex)
            stack.extend(
                [neighbor for neighbor, _ in graph[vertex] if neighbor not
in visited]
```

```
            )

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    while queue:
        vertex = queue.popleft()
        if vertex not in visited:
            print(vertex, end=" ")
            visited.add(vertex)
            queue.extend(
                [neighbor for neighbor, _ in graph[vertex] if neighbor not
in visited]
            )

def ucs(graph, start, goal):
    visited = set()
    queue = [(0, start)]  # priority queue with (cost, node)
    while queue:
        cost, vertex = heapq.heappop(queue)
        if vertex not in visited:
            print(vertex, end=" ")
            visited.add(vertex)
            if vertex == goal:
                print(f"\nGoal '{goal}' reached with cost {cost}")
                return
            for neighbor, edge_cost in graph[vertex]:
                if neighbor not in visited:
                    heapq.heappush(queue, (cost + edge_cost, neighbor))
    print(f"\nGoal '{goal}' not reachable")

class Graph:
    def __init__(self):
        self.graph = {}

    def add_node(self, node):
        if node not in self.graph:
            self.graph[node] = []

    def add_edge(self, from_node, to_node, cost=1):
        if from_node in self.graph:
            self.graph[from_node].append((to_node, cost))
        else:
            self.graph[from_node] = [(to_node, cost)]

    def display_graph(self):
        for node, edges in self.graph.items():
            edges_str = ', '.join([f"{neighbor}({cost})" for neighbor, cost
in edges])
            print(f"{node} -> {edges_str}")

graph = Graph()

graph.add_node("A")
graph.add_node("B")
graph.add_node("C")
graph.add_node("D")

graph.add_edge("A", "B", 1)
graph.add_edge("A", "C", 4)
graph.add_edge("B", "C", 2)
graph.add_edge("B", "D", 5)
graph.add_edge("C", "D", 1)
graph.add_edge("D", "A", 3)

graph.display_graph()

start_vertex = "D"
goal_vertex = "C"
print(f"\nDFS Traversal starting from vertex '{start_vertex}':")
dfs(graph.graph, start_vertex)

print(f"\nBFS Traversal starting from vertex '{start_vertex}':")
bfs(graph.graph, start_vertex)

print(f"\nUCS Traversal starting from vertex '{start_vertex}' to reach goal
'{goal_vertex}':")
ucs(graph.graph, start_vertex, goal_vertex)
```

Output:



```
PS D:\CSE449 - Artificial Intelligence> & "C:/Program Files
A -> B(1), C(4)
B -> C(2), D(5)
C -> D(1)
D -> A(3)

DFS Traversal starting from vertex 'D':
D A C B
BFS Traversal starting from vertex 'D':
D A B C
UCS Traversal starting from vertex 'D' to reach goal 'C':
D A B C
Goal 'C' reached with cost 6
```