**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:**

