Artificial Intelligence

CS-6

Lab: 08

22/10/2024

Task 1:

Write a program to define a graph using nodes and their relationship.

- Use a dictionary whose keys are the nodes of the graph.
- For each key, the corresponding value is a list containing the nodes that are connected by a direct arc from this node.

CODE:

```
def find_shortest_path(graph, start, end, path=[]):
path = path + [start]
   # If we reach the destination node, return the path
   if start == end:
return path
   # If the starting node has no neighbors
if start not in graph:
       return None      shortest
= None    # Explore neighbors
for neighbor in graph[start]:
       if neighbor not in path:  # Avoid cycles           new_path =
find_shortest_path(graph, neighbor, end, path) #recursive call           if
new_path:  # If a path was found              if shortest is None or len(new_path) <
len(shortest):
                shortest = new_path
return shortest
```

```
# Example graph
graph = {

   'A': ['B', 'C'],

   'B': ['D'],

   'C': ['E'],

   'D': ['C', 'E'],

   'E': []

}


print(find_shortest_path(graph, 'A', 'D'))
```

Task 2:

Create a graph with 7 nodes and 5 edges, for the following

- Directed
- Un-directed
- Weighted
- Find Shortest path
- Find neighbor of the node
- Check if edge exists or not

CODE:

```
 graph    =    {}

vertices_no = 0


# Function to add a vertex to the graph

def add_vertex(v):    global graph

global vertices_no

   if v in graph:

      print("Vertex", v, "already exists.")
```

```
    else:
        vertices_no += 1
graph[v] = []

# Function to add an edge between two vertices with an edge weight def
add_edge(v1, v2, e, directed=True):   # By default the graph is directed
global graph     if v1 not in graph:
        print("Vertex", v1, "does not exist.")
    elif v2 not in graph:
        print("Vertex", v2, "does not exist.")
    else:
        graph[v1].append([v2, e])        if not
directed:  # If the graph is undirected
graph[v2].append([v1, e])

# Function to print the graph
def print_graph():     global
graph     for vertex in graph:
for edges in graph[vertex]:
            print(vertex, "->", edges[0], "edge weight:", edges[1])

# Modified find_shortest_path function def
find_shortest_path(graph, start, end, path=[], visited=set()):
    path = path + [start]
    # If we reach the destination node, return the path
    if start == end:
return path
```

```
    # If the starting node has no neighbors
if start not in graph:
        return None

    shortest = None     # Explore
neighbors     for neighbor_info in
graph[start]:
        neighbor = neighbor_info[0]  # Get the neighbor node (ignoring weight)         if
neighbor not in path:  # Avoid cycles             new_path = find_shortest_path(graph,
neighbor, end, path)  # recursive call              if new_path:  # If a path was found
if shortest is None or len(new_path) < len(shortest):
                    shortest = new_path
return shortest

# Function to find neighbors of a node
def find_neighbors(node):     if node
in graph:
        return [neighbor[0] for neighbor in graph[node]]
    else:
        print("Node", node, "does not exist.")
        return []

# Function to check if an edge exists
def edge_exists(v1, v2):     if v1 in
graph:         for neighbor in
graph[v1]:             if neighbor[0] ==
v2:
                return True
```

```
    return False

# Driver code #
Add vertices
for i in range(1, 8):
add_vertex(i)

# Add directed edges with weights (creating 5 edges)
add_edge(1, 2, 1) add_edge(1,
3, 2) add_edge(2, 4, 4)
add_edge(3, 5, 3) add_edge(4,
6, 1)

# Print the graph print("Directed Graph
Representation:") print_graph()

# Add Undirected edges with weights (creating 5 edges)
add_edge(1, 2, 1, directed=False) add_edge(1, 3, 2,
directed=False) add_edge(2, 4, 4, directed=False)
add_edge(3, 5, 3, directed=False) add_edge(4, 6, 1,
directed=False)

print("\n\nUndirected Graph Representation:") print_graph()

# Find the shortest path shortest_path =
find_shortest_path(graph, 1, 5) print(f"\nShortest path
from node 1 to 5: {shortest_path}")
```

```
# Find neighbors of a specific node node = 2

print(f"\nNeighbors of node {node}: {find_neighbors(node)}")


# Check if an edge exists
v1 = 2

v2 = 4

print(f"\nEdge exists between {v1} and {v2}: {edge_exists(v1, v2)}")


v1 = 1 v2 = 6

exists = edge_exists(v1, v2) print(f"Edge exists

between {v1} and {v2}: {exists}")
```