Bisma Farhat

39967

AI

Task1:

\+ Code   + Text

```
graph = {'A':['B','C'],
         'B':['D'],
         'C':['E'],
         'D':['C','E'],
         'E':[]}
def find_path(graph, start, end, path=[]):
    path = path + [start]
    if start == end:
        return path
    if start not in graph:
        return None
    for node in graph[start]:
        if node not in path:
            newpath = find_path(graph, node, end, path)
            if newpath: return newpath
    return None
print(find_path(graph, 'A', 'D'))
```

```
['A', 'B', 'D']
```

Task2:

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

```python
    'G': []
}

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

weighted_graph = {
    'A': [('B', 2)],
    'B': [('C', 4)],
    'C': [('D', 1)],
    'D': [('E', 7)],
    'E': [('F', 3)],
    'F': [],
    'G': []
}
import heapq

def dijkstra(graph, start, end):

    queue = [(0, start)]

    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    visited = set()

    while queue:
        current_distance, current_node = heapq.heappop(queue)

        if current_node in visited:
            continue

        visited.add(current_node)

        if current_node == end:
            return current_distance

        for neighbor, weight in graph[current_node]:
```

```python
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(queue, (distance, neighbor))

    return None

shortest_path_cost = dijkstra(weighted_graph, 'A', 'F')
print("Shortest Path Cost:", shortest_path_cost)

def find_neighbors(graph, node):
    if node in graph:
        return graph[node]
    else:
        return None


neighbors = find_neighbors(weighted_graph, 'B')
print("Neighbors of B:", neighbors)
def edge_exists(graph, v1, v2):
    return any(neighbor == v2 for neighbor, _ in graph.get(v1, []))


edge_check = edge_exists(weighted_graph, 'A', 'B')
print("Edge A -> B exists:", edge_check)
```

```
Shortest Path Cost: 17
Neighbors of B: [('C', 4)]
Edge A -> B exists: True
```