NANCY T THOMAS
1947244
5MCA

GRAPH PROBLEM

This graph problem is about finding the shortest path from one city to another city, a map has been used to create connections between cities. The DFS algorithm uses a Graph class and a Node class, it has a list of open nodes and a list of closed nodes. The DFS algorithm will find a path to a destination, but it is not the shortest path. The code and the output is shown below.

EXPLANATION:

The program is a search algorithm implemented using DFS. Here is a graph problem that is the task is to find the shortest distance from a source to a destination place. This can be represented as a graph i.e. all the places are considered as nodes and they are connected to each other through edges which is the route between 2 places and the distance between the places is the weight of that particular route. Here, it is an un-directed graph which means the places are connected in 2 way direction and they can be traversed back and forth. The places and their connections with other places along with their distances is stored in the program. The source and destination place is taken as input from the user and passed into the algorithm to find the path with the shortest distance which is then printed along with the total cost or distance and given as output.

CODE:

```
# This class represent a graph
class Graph:

    # Initialize the class
    def __init__(self, graph_dict=None, directed=True):
        self.graph_dict = graph_dict or {}
        self.directed = directed
        if not directed:
            self.make_undirected()

    # Create an undirected graph by adding symmetric edges
    def make_undirected(self):
        for a in list(self.graph_dict.keys()):
            for (b, dist) in self.graph_dict[a].items():
                self.graph_dict.setdefault(b, {})[a] = dist

    # Add a link from A and B of given distance, and also add the inverse
link if the graph is undirected
```

```python
    def connect(self, A, B, distance=1):
        self.graph_dict.setdefault(A, {})[B] = distance
        if not self.directed:
            self.graph_dict.setdefault(B, {})[A] = distance

    # Get neighbors or a neighbor
    def get(self, a, b=None):
        links = self.graph_dict.setdefault(a, {})
        if b is None:
            return links
        else:
            return links.get(b)

    # Return a list of nodes in the graph
    def nodes(self):
        s1 = set([k for k in self.graph_dict.keys()])
        s2 = set([k2 for v in self.graph_dict.values() for k2, v2 in
v.items()])
        nodes = s1.union(s2)
        return list(nodes)

# This class represent a node
class Node:

    # Initialize the class
    def __init__(self, name:str, parent:str):
        self.name = name
        self.parent = parent
        self.g = 0 # Distance to start node
        self.h = 0 # Distance to goal node
        self.f = 0 # Total cost

    # Compare nodes
    def __eq__(self, other):
        return self.name == other.name

    # Sort nodes
    def __lt__(self, other):
         return self.f < other.f
```

```
    # Print node
    def __repr__(self):
        return ('({0},{1})'.format(self.position, self.f))


# Depth-first search (DFS)
def depth_first_search(graph, start, end):

    # Create lists for open nodes and closed nodes
    open = []
    closed = []
    # Create a start node and an goal node
    start_node = Node(start, None)
    goal_node = Node(end, None)
    # Add the start node
    open.append(start_node)

    # Loop until the open list is empty
    while len(open) > 0:
        # Get the last node (LIFO)
        current_node = open.pop(-1)
        # Add the current node to the closed list
        closed.append(current_node)

        # Check if we have reached the goal, return the path
        if current_node == goal_node:
            path = []
            while current_node != start_node:
                path.append(current_node.name + ': ' +
str(current_node.g))
                current_node = current_node.parent
            path.append(start_node.name + ': ' + str(start_node.g))
            # Return reversed path
            return path[::-1]

        # Get neighbours
        neighbors = graph.get(current_node.name)
        # Loop neighbors
        for key, value in neighbors.items():
            # Create a neighbor node
```

```
            neighbor = Node(key, current_node)
            # Check if the neighbor is in the closed list
            if(neighbor in closed):
                continue
            # Check if neighbor is in open list and if it has a lower f
value
            if(neighbor in open):
                continue
            # Calculate cost so far
            neighbor.g = current_node.g + graph.get(current_node.name,
neighbor.name)
            # Everything is green, add neighbor to open list
            open.append(neighbor)

    # Return None, no path is found
    return None

# The main entry point for this module
def main():

    # Create a graph
    graph = Graph()

    # Create graph connections (Actual distance)
    graph.connect('Frankfurt', 'Wurzburg', 111)
    graph.connect('Frankfurt', 'Mannheim', 85)
    graph.connect('Wurzburg', 'Nurnberg', 104)
    graph.connect('Wurzburg', 'Stuttgart', 140)
    graph.connect('Wurzburg', 'Ulm', 183)
    graph.connect('Mannheim', 'Nurnberg', 230)
    graph.connect('Mannheim', 'Karlsruhe', 67)
    graph.connect('Karlsruhe', 'Basel', 191)
    graph.connect('Karlsruhe', 'Stuttgart', 64)
    graph.connect('Nurnberg', 'Ulm', 171)
    graph.connect('Nurnberg', 'Munchen', 170)
    graph.connect('Nurnberg', 'Passau', 220)
    graph.connect('Stuttgart', 'Ulm', 107)
    graph.connect('Basel', 'Bern', 91)
    graph.connect('Basel', 'Zurich', 85)
    graph.connect('Bern', 'Zurich', 120)
```

```python
    graph.connect('Zurich', 'Memmingen', 184)
    graph.connect('Memmingen', 'Ulm', 55)
    graph.connect('Memmingen', 'Munchen', 115)
    graph.connect('Munchen', 'Ulm', 123)
    graph.connect('Munchen', 'Passau', 189)
    graph.connect('Munchen', 'Rosenheim', 59)
    graph.connect('Rosenheim', 'Salzburg', 81)
    graph.connect('Passau', 'Linz', 102)
    graph.connect('Salzburg', 'Linz', 126)

    # Make graph undirected, create symmetric connections
    graph.make_undirected()

    # Run search algorithm
    start = input("Enter a place to start: ")
    end = input("Enter the destination place: ")
    path = depth_first_search(graph, start, end)
    print(path)
    while True:
      choice = input("Do you want to continue? (y/n) ")
      if(choice == "y"):
        start = input("Enter a place to start: ")
        end = input("Enter the destination place: ")
        path = depth_first_search(graph, start, end)
        print(path)
        print()
      else:
        break

# Tell python to run main method
if __name__ == "__main__":
  main()
```

SCREENSHOTS:

```
Enter a place to start: Ulm
Enter the destination place: Bern
['Ulm: 0', 'Memmingen: 55', 'Zurich: 239', 'Bern: 359']
Do you want to continue? (y/n) n
```