University School of Automation and Robotics
GURU GOBIND SINGH INDRAPRASTHA UNIVERSITY
East Delhi Campus, Surajmal Vihar
Delhi - 110092

ARTIFICIAL INTELLIGENCE LAB

File

COURSE CODE: ARD251

2023-2024

SUBMITTED TO:

Dr. Manoj Kumar

SUBMITTED BY:

Ansh Singh (13219011722)

INDEX IIOT-B2

S. No	Program	Date	Signature
1.			
2.			
3.			
4.			
5.			
6.			
7.			
8.			
9.			
10.			

PROGRAM-1: Write a program to implement breadth first search

CODE :

```python
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = [] # List for visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node): #function for BFS
  visited.append(node)
  queue.append(node)

  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    print (m, end = " ")

    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

# Driver Code
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')    # function calling
```

```
Following is the Breadth-First Search
5 3 7 2 4 8
```

PROGRAM-2: Write a program to implement **Depth First Search**

CODE:

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node):  #function for dfs
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

```
Following is the Depth-First Search
5
3
2
4
8
7
```

PROGRAM-3: Python implementation of the A* algorithm for solving search problem.

Code:

```python
!pip install simpleai
from simpleai.search import SearchProblem, astar

TARGET_GOAL = 'HELLO WORLD'

class CustomHelloProblem(SearchProblem):
    def actions(self, current_state):
        # Define possible actions based on the current state
        if len(current_state) < len(TARGET_GOAL):
            return list(' ABCDEFGHIJKLMNOPQRSTUVWXYZ')
        else:
            return []

    def result(self, current_state, action):
        # Define the result of taking a particular action
        return current_state + action

    def is_goal(self, current_state):
        # Check if the current state is the goal state
        return current_state == TARGET_GOAL

    def heuristic(self, current_state):
        # Estimate how far the current state is from the goal state
        wrong_positions = sum([1 if current_state[i] != TARGET_GOAL[i] else 0
                               for i in range(len(current_state))])
        missing_chars = len(TARGET_GOAL) - len(current_state)
        return wrong_positions + missing_chars

# Create an instance of the custom problem with an initial state of an empty string
custom_problem = CustomHelloProblem(initial_state='')

# Use A* algorithm to find a solution to the problem
result_solution = astar(custom_problem)

# Print the final state and the path to reach it
print(result_solution.state)
print(result_solution.path())
```

OUTPUT:

```
Requirement already satisfied: simpleai in /usr/local/lib/python3.10/dist-packages (0.8.3)
HELLO WORLD
[(None, ''), ('H', 'H'), ('E', 'HE'), ('L', 'HEL'), ('L', 'HELL'), ('O', 'HELLO'), (' ', 'HELLO '), ('W', 'HELLO W'), ('O', 'HELLO WO'), ('R', 'HELLO WOR'), ('L', 'HELLO WORL'), ('D', 'HELLO WORLD')]
```

Problem 4: write a python code for A* Algorithm demonstration.

Code:

```
import heapq

def custom_astar_search(graph, start_node, goal_node, heuristics):
    # Initialize the open list with the starting node and a cost of 0
    open_nodes = [(0, start_node)]
    # Initialize the closed list as an empty set (nodes already evaluated)
    closed_nodes = set()
    # Initialize a dictionary to store the cost to reach each node, initially set to infinity
    g_costs = {node: float('inf') for node in graph}
    # Set the cost to reach the starting node as 0
    g_costs[start_node] = 0
    # Initialize a dictionary to store the parent node of each node
    parents = {}

    # Main loop: continue until there are nodes in the open list
    while open_nodes:
        # Pop the node with the lowest f_cost (g_cost + heuristic) from the open list
        _, current_node = heapq.heappop(open_nodes)

        # Check if the current node is the goal node
        if current_node == goal_node:
            # Reconstruct and return the path from the goal to the start
            path = [current_node]
            while current_node != start_node:
                current_node = parents[current_node]
                path.append(current_node)
            path.reverse()
            return path

        # If the current node is not the goal, add it to the closed list
        if current_node not in closed_nodes:
            closed_nodes.add(current_node)

            # Explore neighbors of the current node and update their g_costs and f_costs
            for neighbor, cost in graph[current_node]:
                # Calculate the tentative g_cost for the neighbor node
                tentative_g_cost = g_costs[current_node] + cost
```

```python
                # Calculate the tentative g_cost for the neighbor node
                tentative_g_cost = g_costs[current_node] + cost
                # If this g_cost is lower than the previously recorded g_cost, update it
                if tentative_g_cost < g_costs[neighbor]:
                    g_costs[neighbor] = tentative_g_cost
                    # Calculate the f_cost for the neighbor node (g_cost + heuristic)
                    f_cost = tentative_g_cost + heuristics[neighbor]
                    # Add the neighbor node to the open list with its f_cost
                    heapq.heappush(open_nodes, (f_cost, neighbor))
                    # Record the parent of the neighbor node
                    parents[neighbor] = current_node

    # If the open list becomes empty and the goal is not reached, return None (no path found)
    return None

# Define the graph structure and heuristic values for the nodes
custom_graph = {
    'S': [('A', 1), ('B', 4)],
    'A': [('B', 2), ('C', 5), ('D', 12)],
    'B': [('C', 2)],
    'C': [('D', 3)],
    'D': [],
}

custom_heuristics = {
    'S': 7,
    'A': 6,
    'B': 2,
    'C': 1,
    'D': 0,
}

# Define the start and goal nodes
custom_start_node = 'S'
custom_goal_node = 'D'

# Call the A* search function to find the path from the start to the goal
custom_path = custom_astar_search(custom_graph, custom_start_node, custom_goal_node, custom_heuristics)
```

```python
# Call the A* search function to find the path from the start to the goal
custom_path = custom_astar_search(custom_graph, custom_start_node, custom_goal_node, custom_heuristics)

# Print the result: either the path found or a message indicating no path found
if custom_path:
    print("Path from", custom_start_node, "to", custom_goal_node, ":", ' -> '.join(custom_path))
else:
    print("No path found from", custom_start_node, "to", custom_goal_node)
```

```
Path from S to D : S -> A -> B -> C -> D
```

Output:

```
Path from S to D : S -> A -> B -> C -> D
```

Problem 5: WAP in python to implement Alpha-Beta pruning to find the optical value of a node.

CODE:

```
tree = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': [2, 3],
    'E': [5, 9],
    'F': [0, 1],
    'G': [7, 5],
}

def minimax_alpha_beta(node, alpha, beta, is_maximizing):
    if isinstance(node, str):
        children = tree[node]
        if is_maximizing:
            value = -float('inf')
            for child in children:
                value = max(value, minimax_alpha_beta(child, alpha, beta, False))
                alpha = max(alpha, value)
                if alpha >= beta:
                    break
            return value
        else:
            value = float('inf')
            for child in children:
                value = min(value, minimax_alpha_beta(child, alpha, beta, True))
                beta = min(beta, value)
                if alpha >= beta:
                    break
            return value
    else:
        return node

optimal_value = minimax_alpha_beta('A', -float('inf'), float('inf'), True)
print("Optimal Value for the Root Node (A):", optimal_value)
```

Output:

```
Optimal Value for the Root Node (A): 3
```

Problem 6: to solve 8 puzzle problem using heuristic search technique.

CODE:

```
from simpleai.search import astar, SearchProblem

# Define the PuzzleSolver class that inherits from SearchProblem
class CustomEightPuzzleSolver(SearchProblem):
    def actions(self, current_state):
        # Get the possible moves (actions) for the current state
        # Find the empty tile (represented as 0)
        empty_tile = None
        for row in current_state:
            if 0 in row:
                empty_tile = (current_state.index(row), row.index(0))
                break

        available_actions = []
        # Check if it's possible to move tiles into the empty space
        if empty_tile[0] > 0:
            available_actions.append((-1, 0))  # Move tile above the empty space
        if empty_tile[0] < 2:
            available_actions.append((1, 0))   # Move tile below the empty space
        if empty_tile[1] > 0:
            available_actions.append((0, -1))  # Move tile to the left of the empty space
        if empty_tile[1] < 2:
            available_actions.append((0, 1))   # Move tile to the right of the empty space

        return available_actions

    def result(self, current_state, action):
        # Apply the specified action to the current state
        new_state = [list(row) for row in current_state]
        empty_tile = None

        for row in new_state:
            if 0 in row:
                empty_tile = (new_state.index(row), row.index(0))
                break

        move_row, move_col = action
        target_row = empty_tile[0] + move_row
        target_col = empty_tile[1] + move_col
```

```python
        target_col = empty_tile[1] + move_col

        # Swap the empty space with the target tile
        new_state[empty_tile[0]][empty_tile[1]] = new_state[target_row][target_col]
        new_state[target_row][target_col] = 0

        return tuple(tuple(row) for row in new_state)

    def is_goal(self, current_state):
        # Check if the current state matches the goal state
        return current_state == CUSTOM_GOAL_STATE

    def heuristic(self, current_state):
        # Calculate the Manhattan distance heuristic for the current state
        distance = 0
        for row in range(3):
            for col in range(3):
                tile = current_state[row][col]
                if tile != 0:
                    goal_row, goal_col = CUSTOM_GOAL_POSITIONS[tile]
                    distance += abs(row - goal_row) + abs(col - goal_col)
        return distance

# Define the goal state and initial state
CUSTOM_GOAL_STATE = ((1, 2, 3), (4, 5, 6), (7, 8, 0))
CUSTOM_INITIAL_STATE = ((1, 0, 2), (6, 3, 4), (7, 5, 8))

# Define the goal positions for each tile
CUSTOM_GOAL_POSITIONS = {
    1: (0, 0), 2: (0, 1), 3: (0, 2),
    4: (1, 0), 5: (1, 1), 6: (1, 2),
    7: (2, 0), 8: (2, 1), 0: (2, 2)
}

# Create an instance of the CustomEightPuzzleSolver and find the solution
custom_problem = CustomEightPuzzleSolver(CUSTOM_INITIAL_STATE)
custom_result = astar(custom_problem)

# Print the solution (sequence of actions)
```

```python
# Print the solution (sequence of actions)
if custom_result:
    print("Solution found!")
    print("Number of moves:", len(custom_result.path()))
    for action, state in custom_result.path():
        print("Move:", action)
        for row in state:
            print(row)
else:
    print("No solution found.")
```

Output:

```
Solution found!
Number of moves: 16
Move: None
(1, 0, 2)
(6, 3, 4)
(7, 5, 8)
Move: (0, 1)
(1, 2, 0)
(6, 3, 4)
(7, 5, 8)
Move: (1, 0)
(1, 2, 4)
(6, 3, 0)
(7, 5, 8)
Move: (0, -1)
(1, 2, 4)
(6, 0, 3)
(7, 5, 8)
Move: (0, -1)
(1, 2, 4)
(0, 6, 3)
(7, 5, 8)
Move: (-1, 0)
(0, 2, 4)
(1, 6, 3)
(7, 5, 8)
Move: (0, 1)
(2, 0, 4)
(1, 6, 3)
(7, 5, 8)
Move: (0, 1)
(2, 4, 0)
(1, 6, 3)
(7, 5, 8)
Move: (1, 0)
(2, 4, 3)
(1, 6, 0)
(7, 5, 8)
Move: (0, -1)
(2, 4, 3)
(1, 0, 6)
(7, 5, 8)
Move: (-1, 0)
(2, 0, 3)
(1, 4, 6)
(7, 5, 8)
Move: (0, -1)
(0, 2, 3)
(1, 4, 6)
(7, 5, 8)
Move: (1, 0)
(1, 2, 3)
(0, 4, 6)
(7, 5, 8)
Move: (0, 1)
(1, 2, 3)
(4, 0, 6)
(7, 5, 8)
Move: (1, 0)
(1, 2, 3)
(4, 5, 6)
(7, 0, 8)
Move: (0, 1)
(1, 2, 3)
(4, 5, 6)
(7, 8, 0)
```

Problem 7: Write a python code to create a TIC-TAC-TOE game where a player can play against a computer opponent that uses the minimum algorithm.

CODE:

```
# Define the Tic-Tac-Toe board as a 3x3 grid
board = [[' ', ' ', ' '],
         [' ', ' ', ' '],
         [' ', ' ', ' ']]

# Function to display the Tic-Tac-Toe board
def display_board(board):
    for row in board:
        print('|'.join(row))
        print('-' * 5)

# Function to check if the game has ended
def is_game_over(board):
    # Check rows, columns, and diagonals for a win
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != ' ':
            return True
        if board[0][i] == board[1][i] == board[2][i] != ' ':
            return True
    if board[0][0] == board[1][1] == board[2][2] != ' ':
        return True
    if board[0][2] == board[1][1] == board[2][0] != ' ':
        return True
    # Check for a tie (no empty spaces left)
    if all(board[i][j] != ' ' for i in range(3) for j in range(3)):
        return True
    return False

# Function to evaluate the game state
def evaluate(board):
    # Check if the computer wins
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] == 'O':
            return 1
```

```
        if board[0][i] == board[1][i] == board[2][i] == 'O':
            return 1
    if board[0][0] == board[1][1] == board[2][2] == 'O':
        return 1
    if board[0][2] == board[1][1] == board[2][0] == 'O':
        return 1

    # Check if the player wins
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] == 'X':
            return -1
        if board[0][i] == board[1][i] == board[2][i] == 'X':
            return -1
    if board[0][0] == board[1][1] == board[2][2] == 'X':
        return -1
    if board[0][2] == board[1][1] == board[2][0] == 'X':
        return -1

    # The game is a tie
    return 0

# Minimax function for the computer's move
def minimax(board, depth, is_maximizing):
    if is_game_over(board):
        return evaluate(board)

    if is_maximizing:
        best_score = -float('inf')
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = 'O'
                    score = minimax(board, depth + 1, False)
                    board[i][j] = ' '
                    best_score = max(score, best_score)
        return best_score

    else:
        best_score = float('inf')
```

```
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = 'X'
                    score = minimax(board, depth + 1, True)
                    board[i][j] = ' '
                    best_score = min(score, best_score)
        return best_score

# Function to make the computer's move
def make_computer_move(board):
    best_move = None
    best_score = -float('inf')
    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':
                board[i][j] = 'O'
                score = minimax(board, 0, False)
                board[i][j] = ' '
                if score > best_score:
                    best_score = score
                    best_move = (i, j)
    if best_move:
        board[best_move[0]][best_move[1]] = 'O'

# Main game loop
while not is_game_over(board):
    display_board(board)
    player_row = int(input("Enter row (0, 1, 2): "))
    player_col = int(input("Enter column (0, 1, 2): "))
    if board[player_row][player_col] == ' ':
        board[player_row][player_col] = 'X'
        if not is_game_over(board):
            make_computer_move(board)
    else:
        print("Invalid move. Try again.")

display_board(board)
result = evaluate(board)
```

```python
if result == 1:
    print("Computer wins!")
elif result == -1:
    print("Player wins!")
else:
    print("It's a tie!")
```

Output :

```
 | |
-----
 | |
-----
 | |
-----
Enter row (0, 1, 2): 0
Enter column (0, 1, 2): 0
X| |
-----
 |O|
-----
 | |
-----
Enter row (0, 1, 2): 2
Enter column (0, 1, 2): 2
X|O|
-----
 |O|
-----
 | |X
-----
Enter row (0, 1, 2): 0
Enter column (0, 1, 2): 2
X|O|X
-----
 |O|O
-----
 | |X
-----
Enter row (0, 1, 2): 1
Enter column (0, 1, 2): 0
X|O|X
-----
X|O|O
-----
 |O|X
-----
Computer wins!
```

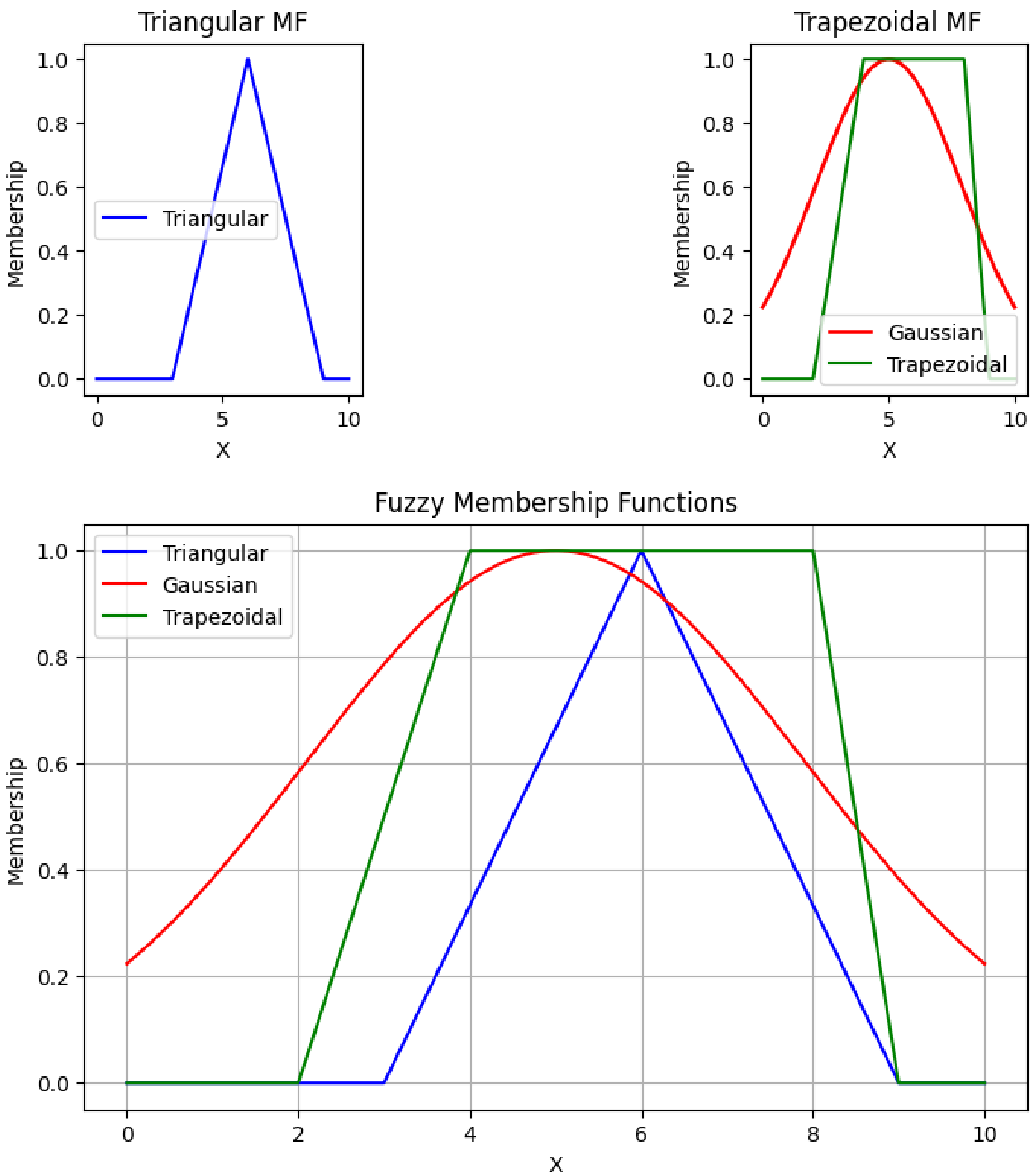
PROBLEM 8 : Write a programme visualizing Fuzzy membership function.

CODE:

```
import numpy as np
import matplotlib.pyplot as plt
import skfuzzy as fuzz

#Generate universe variables
x = np.linspace(0, 10, 1000)

# Generate fuzzy membership functions

triangular_mf= fuzz.trimf(x, [3, 6, 9])

gaussian_mf = fuzz.gaussmf(x, np.mean(x), np.std(x))

trapezoid_mf = fuzz.trapmf(x, [2, 4, 8, 9])

#Plotting triangular membership function

plt.figure(figsize=(8, 3))

plt.subplot(131)
plt.title("Triangular MF")

plt.plot(x, triangular_mf, 'b', linewidth=1.5, label="Triangular")

plt.xlabel('X')

plt.ylabel('Membership')

plt.legend()

# Plotting Gaussian membership function

plt.subplot(133)


plt.title('Gaussian MF')

plt.plot(x, gaussian_mf, 'r', linewidth=1.8, label='Gaussian')

plt.xlabel('X')

plt.ylabel('Membership')

plt.legend()

#Plotting trapezoidal membership function
```

```
plt.subplot(133)

plt.title("Trapezoidal MF")

plt.plot(x, trapezoid_mf, 'g', linewidth=1.5, label='Trapezoidal')

plt.xlabel('X')

plt.ylabel('Membership')

plt.legend()

# Show the plots

plt.tight_layout

plt.show()

# Plotting all membership functions on one graph

plt.figure(figsize=(8, 5))

plt.plot(x, triangular_mf, 'b', linewidth=1.5, label='Triangular')

plt.plot(x, gaussian_mf, 'r', linewidth=1.5, label='Gaussian')

plt.plot(x, trapezoid_mf, 'g', linewidth=1.5, label="Trapezoidal")

#Customize the plot

plt.title('Fuzzy Membership Functions')

plt.xlabel('X')

plt.ylabel('Membership')

plt.legend()

plt.grid(True)
# Show the plot

plt.show()
```

OUTPUT:



Triangular MF
Triangular
Membership
X
Trapezoidal MF
Gaussian
Trapezoidal
Membership
X
Fuzzy Membership Functions
Triangular
Gaussian
Trapezoidal
Membership
X

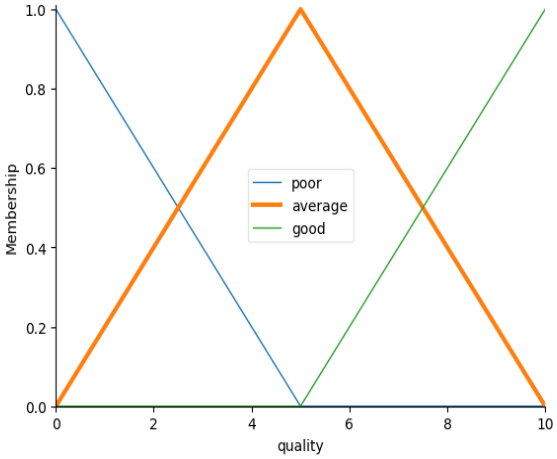
PROBLEM 9: : Write a programme implimentimg a tipping control system.

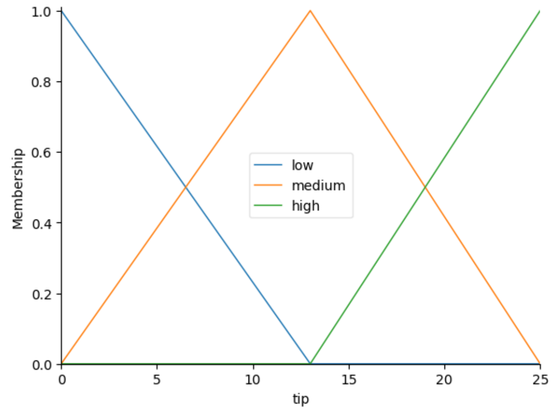
CODE:

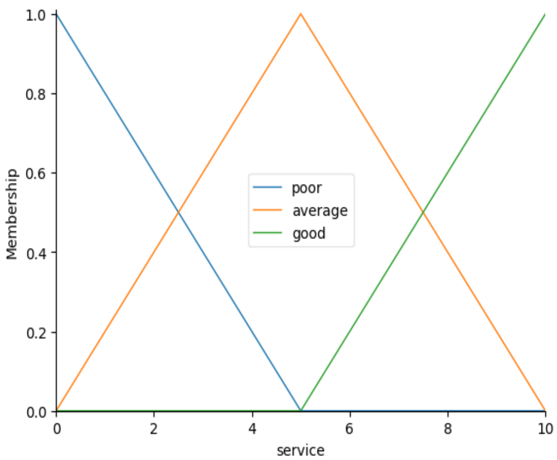
```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

# Define antecedents and consequent
quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')
service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')
tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')

# Auto-membership function population is possible with .automf(3, 5, or 7)
quality.automf(3)
service.automf(3)

# Custom membership functions
tip['low'] = fuzz.trimf(tip.universe, [0, 0, 13])
tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])
tip['high'] = fuzz.trimf(tip.universe, [13, 25, 25])

# View membership functions
quality['average'].view()
service.view()
tip.view()

# Define rules
rule1 = ctrl.Rule(quality['poor'] & service['poor'], tip['low'])
rule2 = ctrl.Rule(service['average'], tip['medium'])
rule3 = ctrl.Rule(service['good'] | quality['good'], tip['high'])

# View rules
rule1.view()

# Create control system
tipping_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
tipping = ctrl.ControlSystemSimulation(tipping_ctrl)

# Pass inputs
tipping.input['quality'] = 10
tipping.input['service'] = 10

# Compute the result
tipping.compute()
print(tipping.output['tip'])

# View the result
tip.view(sim=tipping)
```

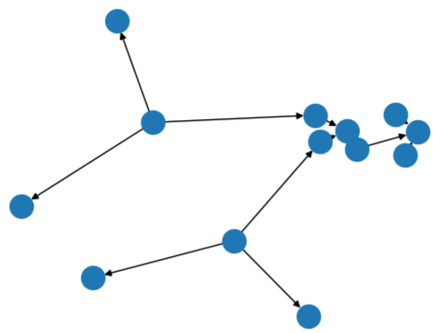
OUTPUT:

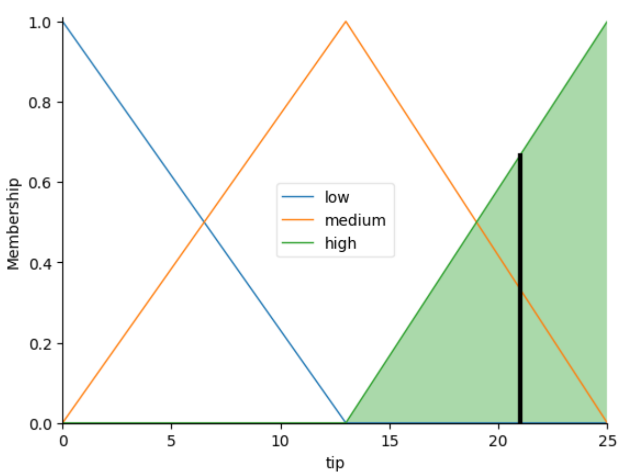
```
21.0
```


1.0
0.8
0.6
0.4
0.2
0.0
Membership
0
2
4
6
8
10
quality
poor
average
good


1.0
0.8
0.6
0.4
0.2
0.0
Membership
0
5
10
15
20
25
tip
low
medium
high


1.0
0.8
0.6
0.4
0.2
0.0
Membership
0
2
4
6
8
10
service
poor
average
good




1.0
0.8
0.6
0.4
0.2
0.0
Membership
0
5
10
15
20
25
tip
low
medium
high

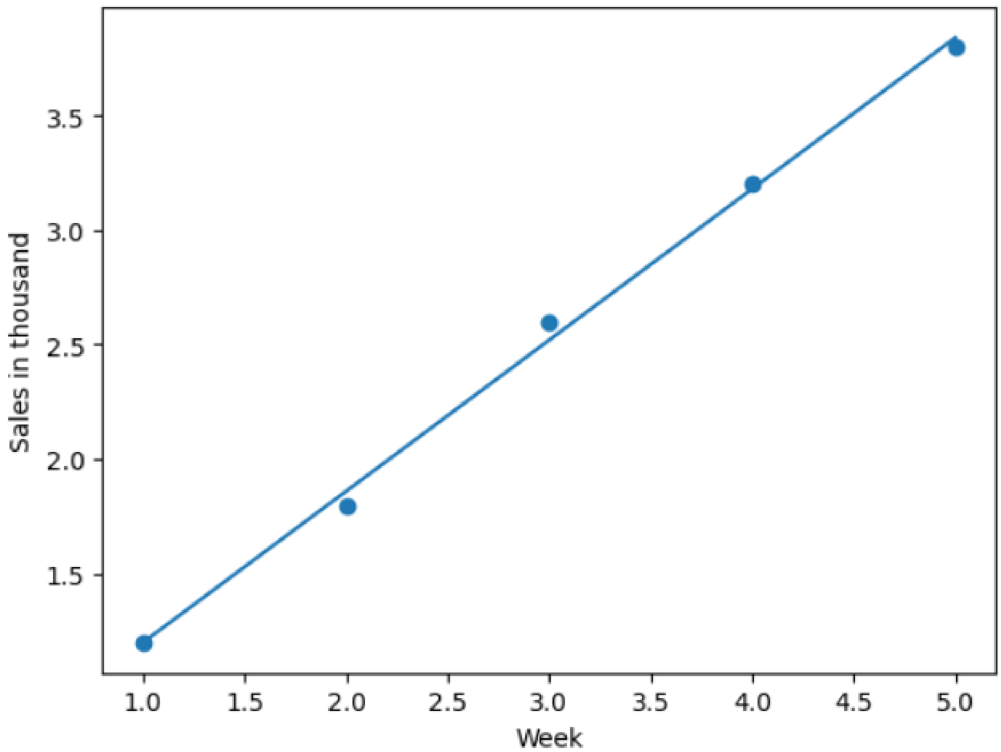
Problem 10 :

Code :

```
import numpy as np
import matplotlib.pyplot as plt

# Input data
week_numbers = np.array([1, 2, 3, 4, 5])
sales_in_thousand = np.array([1.2, 1.8, 2.6, 3.2, 3.8])

# Scatter plot of the data
plt.scatter(week_numbers, sales_in_thousand)

# Calculate linear regression parameters
n = len(week_numbers)
x_mean = np.mean(week_numbers)
y_mean = np.mean(sales_in_thousand)
xy_mean = np.mean(week_numbers * sales_in_thousand)
x2_mean = np.mean(week_numbers ** 2)

slope = (xy_mean - x_mean * y_mean) / (x2_mean - x_mean ** 2)
intercept = y_mean - slope * x_mean

# Print slope and intercept
print(f"Slope: {slope:.2f}\nIntercept: {intercept:.2f}")

# Define the linear regression function
def linear_regression(x):
    return slope * x + intercept

# Plot the linear regression line
plt.plot(week_numbers, linear_regression(week_numbers))
plt.xlabel("Week")
plt.ylabel("Sales in thousand")
plt.show()

# Predict sales for the 7th and 12th weeks
predicted_sales_7th_week = linear_regression(7)
predicted_sales_12th_week = linear_regression(12)

print("The predicted sales for the 7th week: ", round(predicted_sales_7th_week, 2))
print("The predicted sales for the 12th week: ", round(predicted_sales_12th_week, 2))
```

Output:

```
Slope: 0.66
Intercept: 0.54
```




```
The predicted sales for the 7th week:  5.16
The predicted sales for the 12th week:  8.46
```