

Experiment No: 1

Aim: -Prepare PEAS properties. Implement PEAS property for any one AI agent.

Objective: -To understand PEAS property of AI agent.

Theory:-An AI agent is a software system or autonomous entity that perceives its surroundings, makes decisions using reasoning or rules, and takes actions to independently achieve a specific goal without requiring constant human intervention

Types of AI Agents

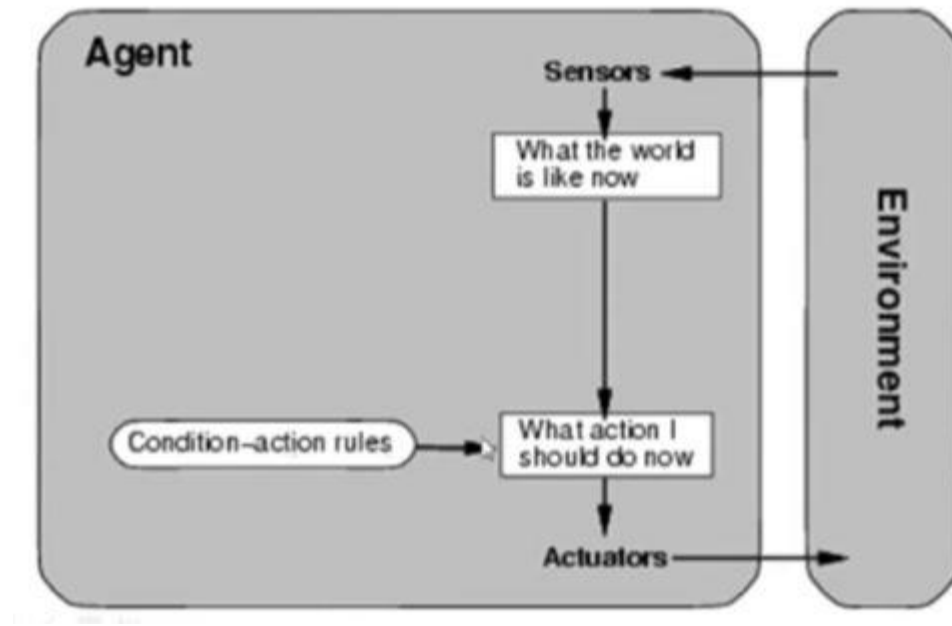
Agents are categorized by how they process context, memory, and objectives:

- **Simple Reflex Agents:** Follow strict, immediate condition-action rules (if-then); they rely entirely on the current input and have no memory.
- **Model-Based Reflex Agents:** Track an internal state to handle situations where some environment information is hidden or partially observable.
- **Goal-Based Agents:** Look ahead by planning multiple steps to reach a specific target or objective.
- **Utility-Based Agents:** Optimize decisions by calculating which possible action yields the highest "utility" or value (e.g., balancing speed vs. safety).
- **Learning Agents:** Capable of adapting, learning from past experiences, and improving their performance over time using feedback.
- **Modern Agentic AI (Multi-Agent/LLM-based):** Utilize large language models to use tools (APIs, code interpreters) and collaborate with other independent agents to accomplish complex workflows.

Types of Environments

The setting in which an agent operates dictates how it must behave and what level of complexity is required to function effectively:

- **Fully vs. Partially Observable:** Whether the agent's sensors can see the complete state of the world at all times (e.g., Chess is fully observable, Poker is partially).
- **Deterministic vs. Stochastic:** In a deterministic environment, the next state is 100% predictable based on the current action; stochastic environments introduce randomness (like the stock market).
- **Static vs. Dynamic:** Static environments do not change while the AI is deliberating, while dynamic environments (like driving a car in traffic) continuously evolve.
- **Single-Agent vs. Multi-Agent:** Whether the agent operates entirely alone or interacts/competes with other independent agents.
- **Discrete vs. Continuous:** Environments with a finite, countable number of actions/states (discrete, like a crossword) versus those with infinite changing variables (continuous, like driving speed).



PEAS Representation:

PEAS stand for Performance Measures, Environment, Actuators, and Sensors.

Performance Measure: If the objectives function to judge the performance of the agent. For example, in case of pick and place robot, no of correct parts in a bin can be the performance measure.

Environment: It the real environment where the agent need to deliberate actions.

Actuators: These are the tools, equipment or organs using which agent perform actions in the environment. This works as output of the agent.

Sensors: These are tools, organs using which agent captures the state of the environment. This works as input to the agent.

PEAS descriptor for Automated Car Driver:

Performance Measure: Safety: Automated system should be able to drive the car safely without dashing anywhere. Optimum speed: Automated systems should be able to maintain the optimal speed depending upon the surroundings. Comfortable journey: Automated systems should be able to give a comfortable journey to the end user.

Environment: Roads: Automated car drivers should be able to drive on any kind of road ranging from city roads to highways. Traffic conditions: You will find different sort of traffic conditions for different types of roads,signals.

Actuators: Steering wheel: used to direct the car in desired directions. Accelerator gear: To increase or decrease speed of the car.

Sensors: To take i/p from the environment in car driving example cameras,GPS radar system etc.

In this way, an AI problem can be identified and its PEAS description is to be stated.

CONCLUSION: Thus, a case study PEAS properties. Implement PEAS property for any one AI agent is studied.

VIVA QUESTIONS

1. What does PEAS stand for in Artificial Intelligence?

2. Why is the PEAS framework important when designing an AI agent?

3. What is the role of the Performance Measure in a PEAS description?

4. Differentiate between Sensors and Actuators with an example.

5. Can two AI agents performing different tasks have the same PEAS description?

Why or why not?

Experiment No. 1

Title: PEAS Properties of an AI-Based Smart Security Camera

Aim: To prepare the PEAS properties and implement the PEAS representation for an AI-based Smart Security Camera.

Theory

An AI agent is a system that observes its surroundings, makes decisions using artificial intelligence, and performs actions to achieve a specific goal.

A Smart Security Camera is an AI-based system that monitors an area, detects suspicious activities, identifies possible intruders, and sends alerts to the owner.

PEAS stands for Performance Measure, Environment, Actuators, and Sensors. It is used to describe the working of an AI agent.

PEAS Representation

PEAS Component	Description
P – Performance Measure	Detect intruders accurately, reduce false alarms, provide quick alerts, and improve security.
E – Environment	Homes, offices, shops, buildings, parking areas, and other monitored places.
A – Actuators	Alarm or siren, mobile notification system, camera movement controls, and warning lights.
S – Sensors	Camera, motion sensor, microphone, and infrared sensor for night vision.

Working of an AI-Based Smart Security Camera

1. **Capture:** The camera captures images and videos of the surrounding area.
2. **Detection:** AI analyzes the video to detect people, movement, or suspicious activities.
3. **Decision:** The system determines whether the activity requires an alert.
4. **Action:** It activates an alarm or sends a notification to the owner's mobile phone.
5. **Monitoring:** The camera continues monitoring the area for further activities.

Conclusion: Thus, the PEAS properties of an AI-based Smart Security Camera were studied. The system uses sensors to monitor its environment, AI to detect suspicious activities, and actuators to provide alerts and improve security.

Experiment No. 2

Aim

To implement any one informed search technique using the A* algorithm.

Theory

Informed search is a search technique used in Artificial Intelligence to find a solution efficiently. It uses additional information, called a heuristic, to guide the search toward the goal.

The A* algorithm is a popular informed search algorithm used to find the shortest path between a starting point and a goal in a weighted graph. It considers both the actual cost and the estimated cost to select the best path.

The A* algorithm uses the following formula:

$$f(n) = g(n) + h(n)$$

Where:

- $g(n)$: Actual cost from the starting node to the current node.
- $h(n)$: Estimated cost from the current node to the goal node.
- $f(n)$: Total estimated cost of the path.

Working of A* Algorithm

1. Start from the initial node.
2. Calculate the $f(n)$ value for the available nodes.
3. Select the node with the lowest $f(n)$ value.
4. Check whether the selected node is the goal node.
5. If the goal is not reached, explore its neighbouring nodes.
6. Repeat the process until the goal is found or no path is available.

Code & Output:

```
import heapq

def astar(start, goal):
    graph = {
        'A': {'B':1, 'C':3},
        'B': {'D':2, 'E':4},
        'C': {'F':2},
        'D': {'G':3},
        'E': {'G':1},
        'F': {'G':2},
        'G': {}
    }
```

```

q = [(0, start)]
parent = {start: None}
cost = {start: 0}

while q:
    _, node = heapq.heappop(q)

    if node == goal:
        path = []
        while node:
            path.append(node)
            node = parent[node]
        return path[::-1], cost[goal]

    for nxt, weight in graph[node].items():
        new_cost = cost[node] + weight

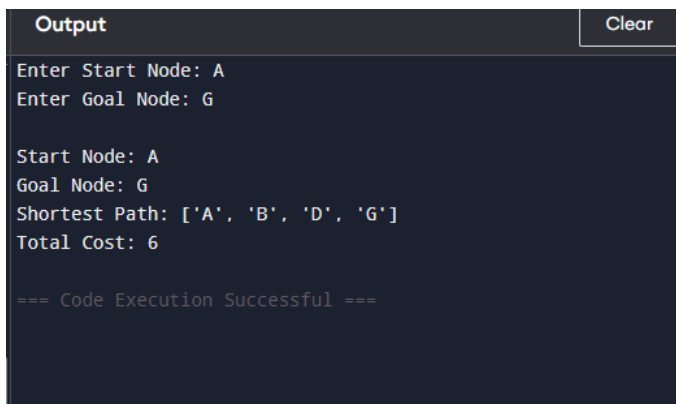
        if nxt not in cost or new_cost < cost[nxt]:
            cost[nxt] = new_cost
            parent[nxt] = node
            heapq.heappush(q, (new_cost, nxt))

start = input("Enter Start Node: ")
goal = input("Enter Goal Node: ")

path, cost = astar(start, goal)

print("\nStart Node:", start)
print("Goal Node:", goal)
print("Shortest Path:", path)
print("Total Cost:", cost)

```



The screenshot shows a dark-themed output window with a 'Clear' button in the top right corner. The text inside the window displays the input and output of the A* algorithm. The input consists of 'Enter Start Node: A' and 'Enter Goal Node: G'. The output shows 'Start Node: A', 'Goal Node: G', 'Shortest Path: ['A', 'B', 'D', 'G']', and 'Total Cost: 6'. At the bottom, it states '=== Code Execution Successful ==='.

```

Output
Clear

Enter Start Node: A
Enter Goal Node: G

Start Node: A
Goal Node: G
Shortest Path: ['A', 'B', 'D', 'G']
Total Cost: 6

=== Code Execution Successful ===

```

Conclusion

Hence, the informed search technique using the A* algorithm was successfully implemented.

Experiment No. 3

Aim

To implement uninformed search using the Water Jug Problem.

Theory

Uninformed search, also called blind search, is a search technique used in Artificial Intelligence to find a solution without using heuristic information. It uses the initial state, available actions, and goal state to solve a problem.

The Water Jug Problem is a common problem in Artificial Intelligence in which two jugs of different capacities are used to measure an exact amount of water.

For example, we have two jugs: one with a capacity of 4 litres and another with a capacity of 3 litres. The goal is to measure exactly 2 litres of water.

The following operations can be performed:

- Fill: Fill a jug completely with water.
- Empty: Empty all the water from a jug.
- Pour: Transfer water from one jug to another until the first jug is empty or the second jug is full.

Working of the Water Jug Problem

7. Start with both jugs empty.
8. Fill the 4-litre jug completely.
9. Pour water into the 3-litre jug until it becomes full.
10. Empty the 3-litre jug.
11. Transfer the remaining water into the 3-litre jug.
12. Fill the 4-litre jug again.
13. Pour water into the 3-litre jug until it becomes full.
14. The 4-litre jug now contains exactly 2 litres of water.

Code & Output:

```
from collections import deque
```

```
def water_jug(m, n, target):  
    visited = set()  
    queue = deque([(0, 0)])
```

```
    while queue:  
        a, b = queue.popleft()
```

```

    if (a, b) in visited:
        continue
    visited.add((a, b))

    if a == target or b == target or a + b == target:
        return True

    next_states = [
        (m, b), # Fill jug1
        (a, n), # Fill jug2
        (0, b), # Empty jug1
        (a, 0), # Empty jug2
        (a - min(a, n - b), b + min(a, n - b)), # Pour jug1 -> jug2
        (a + min(b, m - a), b - min(b, m - a)) # Pour jug2 -> jug1
    ]

    for state in next_states:
        if state not in visited:
            queue.append(state)

return False

m = int(input("Enter capacity of Jug 1: "))
n = int(input("Enter capacity of Jug 2: "))
target = int(input("Enter target amount: "))

if water_jug(m, n, target):
    print("Possible")
else:
    print("Not Possible")

```

```

Output

Enter capacity of Jug 1: 2
Enter capacity of Jug 2: 3
Enter target amount: 2
Possible

=== Code Execution Successful ===

```

Conclusion

Hence, the Water Jug Problem was successfully implemented using uninformed search.

Experiment No. 4 : SLP-NN Using Learning Rule

Aim

To implement a Single Layer Perceptron Neural Network (SLP-NN) using the learning rule in Python.

Theory

1. SLP is a simple neural network with an input layer and an output neuron.
2. Weights and bias are initialized to zero.
3. The network calculates the output using the weighted sum of inputs.
4. The error is calculated as Target – Actual Output.
5. Weights and bias are updated using the learning rule until the network learns the correct output.

Learning Rule:

$$w_new = w_old + \eta (T - y) X$$

Where:

- w = weight
- η = learning rate
- T = target output
- y = predicted output
- X = input

Python Program

```
# SLP-NN using Learning Rule

import numpy as np

# Input data (AND gate)
X = np.array([[0, 0],
              [0, 1],
              [1, 0],
              [1, 1]])

# Target output
T = np.array([0, 0, 0, 1])

# Initialize weights and bias
w = np.zeros(2)
b = 0
lr = 0.1

# Training
for epoch in range(10):
    for i in range(len(X)):
        y = 1 if np.dot(X[i], w) + b > 0 else 0
        error = T[i] - y
        w = w + lr * error * X[i]
```

```
b = b + lr * error

# Display output
print("Final Weights:", w)
print("Final Bias:", b)
print("Output:")
for x in X:
    y = 1 if np.dot(x, w) + b > 0 else 0
    print(x, "=", y)
```

Output

Final Weights: [0.2 0.1]

Final Bias: -0.1

Output:

[0 0] = 0

[0 1] = 0

[1 0] = 0

[1 1] = 1

Conclusion

The Single Layer Perceptron was implemented successfully using the learning rule, and it correctly learned the AND gate outputs.

Experiment No. 5: MLP Using Supervised Learning

Aim

To implement a Multilayer Perceptron (MLP) using a supervised learning algorithm in Python.

Theory

1. MLP is a type of artificial neural network.
2. It contains an input layer, one or more hidden layers, and an output layer.
3. Supervised learning uses input data with known target outputs.
4. The model learns by adjusting weights and biases during training.
5. After training, the model predicts outputs for new inputs.

Important Formulae:

- Weighted sum: $z = \sum(w_i x_i) + b$
- ReLU activation: $f(z) = \max(0, z)$
- Sigmoid activation: $\sigma(z) = 1 / (1 + e^{-z})$
- Weight update: $w_{\text{new}} = w_{\text{old}} - \eta(\partial L / \partial w)$
- Bias update: $b_{\text{new}} = b_{\text{old}} - \eta(\partial L / \partial b)$

Where: x = input, w = weight, b = bias, η = learning rate, and L = loss (prediction error).

Python Program

```
# MLP using Supervised Learning

from sklearn.neural_network import MLPClassifier

# Input data (AND gate)
X = [[0, 0],
      [0, 1],
      [1, 0],
      [1, 1]]

# Target output
y = [0, 0, 0, 1]

# Create MLP model
model = MLPClassifier(
    hidden_layer_sizes=(4,),
    activation='relu',
    solver='lbfgs',
    max_iter=2000,
    random_state=1
)

# Train the model
model.fit(X, y)

# Predict output
```

```
print("Predicted Output:")
for x in X:
    print(x, "=", model.predict([x])[0])
```

Output

Predicted Output:

```
[0, 0] = 0
[0, 1] = 0
[1, 0] = 0
[1, 1] = 1
```

Conclusion

The MLP neural network was implemented successfully using supervised learning and trained to predict the AND gate outputs.

EXPERIMENT No. 6

Associative Memory Network Using Hebbian Learning

1. Aim

To implement Associative Memory Network for a given problem statement.

2. Theory

An Associative Memory Network stores patterns and remembers them later. Hebbian learning is a rule that adjusts connections between neurons. A Hopfield Network uses this rule to store patterns. When a noisy pattern is given as input, the network updates its values step by step (iterations) until the pattern becomes stable.

3. Algorithm

1. Start the program.
2. Import the NumPy library.
3. Create a class named AssociativeMemory.
4. Initialize the weight matrix with zeros.
5. Store binary patterns using the Hebbian learning rule.
6. Give a noisy pattern as input.
7. Update the pattern using iterations.
8. Stop when the pattern becomes stable or the iteration limit is reached.
9. Display the recalled pattern.
10. Stop the program.

4. Python Program

```
import numpy as np

class AssociativeMemory:

    def __init__(self, size):
        self.size = size
        self.weights = np.zeros((size, size))

    def train(self, patterns):
        for p in patterns:
            x = np.where(np.array(p) == 0, -1, 1)
            self.weights += np.outer(x, x)

        np.fill_diagonal(self.weights, 0)

    def recall(self, pattern, iterations=10):
        x = np.where(np.array(pattern) == 0, -1, 1)

        for i in range(iterations):
            old = x.copy()

            for j in range(self.size):
                value = np.dot(self.weights[j], x)

                if value > 0:
                    x[j] = 1
```

```

        elif value < 0:
            x[j] = -1

    print("Iteration", i + 1, ":",
          np.where(x == 1, 1, 0))

    if np.array_equal(x, old):
        break

    return np.where(x == 1, 1, 0)

patterns = [
    [1, 1, 1, 0, 0, 0],
    [0, 0, 0, 1, 1, 1]
]

memory = AssociativeMemory(6)
memory.train(patterns)

noisy = [1, 1, 0, 0, 0, 0]

print("Noisy Input:", noisy)
result = memory.recall(noisy)

print("Recalled Pattern:", result)

```

5. Sample Output

```

Noisy Input: [1, 1, 0, 0, 0, 0]
Iteration 1 : [1 1 1 0 0 0]
Iteration 2 : [1 1 1 0 0 0]
Recalled Pattern: [1 1 1 0 0 0]

```

6. Result

The Associative Memory Network was successfully implemented. It stored binary patterns using Hebbian learning and recalled the original pattern from a noisy input.

7. Conclusion

The network can store patterns and recall them using repeated iterations.

Experiment No. 7: Fuzzy Membership Functions

Aim

To implement fuzzy membership functions.

Theory

A fuzzy membership function represents the degree to which an input belongs to a fuzzy set. Its membership value ranges from 0 to 1.

A triangular membership function is defined by three parameters a, b, and c:

- a = starting point
- b = peak point
- c = ending point

Formula:

$$\mu(x) = 0, \quad x \leq a \text{ or } x \geq c$$

$$\mu(x) = (x - a)/(b - a), \quad a < x \leq b$$

$$\mu(x) = (c - x)/(c - b), \quad b < x < c$$

At the peak b, the membership value is 1. At the starting and ending points, it is 0.

Python Code (Jupyter Notebook)

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

# Triangular membership function
def triangular_mf(x, a, b, c):
    if x <= a or x >= c:
        return 0.0
    elif x == b:
        return 1.0
    elif x < b:
        return (x - a) / (b - a)
    else:
        return (c - x) / (c - b)

# Parameters
a, b, c = 20, 40, 60

# Input values
x_values = [10, 20, 30, 40, 50, 60, 70]

# Calculate membership values
print("Input\tMembership")
for x in x_values:
    print(x, "\t", round(triangular_mf(x, a, b, c), 2))

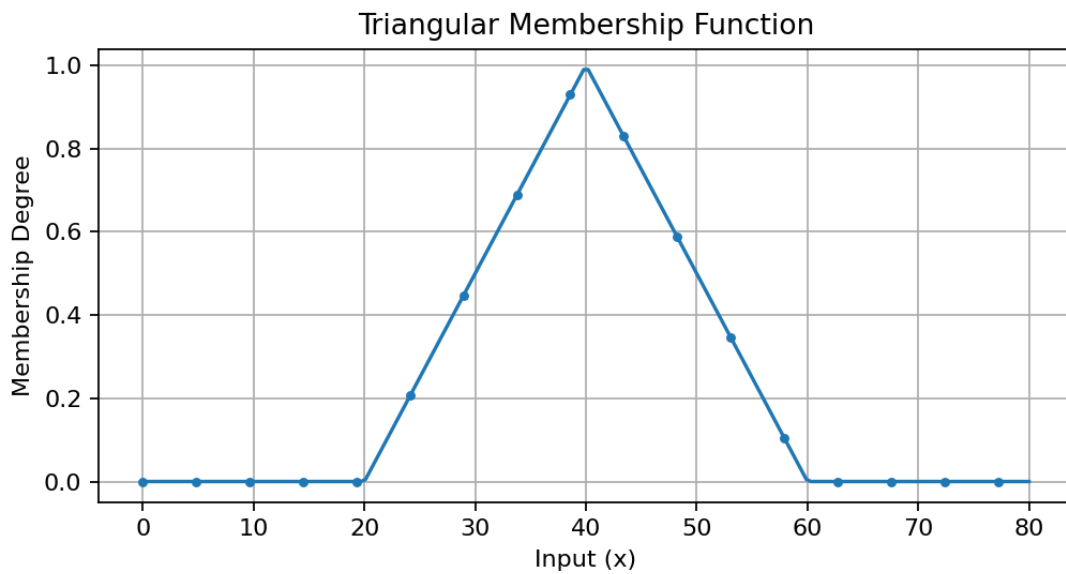
# Draw the diagram
x = np.linspace(0, 80, 200)
y = [triangular_mf(i, a, b, c) for i in x]
```

```
plt.plot(x, y, marker='.', label='Triangular MF')
plt.xlabel("Input (x)")
plt.ylabel("Membership Degree")
plt.title("Triangular Membership Function")
plt.grid(True)
plt.legend()
plt.show()
```

Output

Input	Membership
10	0.0
20	0.0
30	0.5
40	1.0
50	0.5
60	0.0
70	0.0

Triangular Membership Function Diagram



Conclusion

The triangular fuzzy membership function was implemented successfully. The program calculates membership values between 0 and 1 and plots the triangular membership function.

Experiment No. 8: Fuzzy Control System Using Mamdani Method

Aim

Write a program to simulate a fuzzy control system..

Theory

A fuzzy control system uses fuzzy logic to make decisions based on approximate inputs rather than only true or false values.

The Mamdani method is a fuzzy inference technique that uses fuzzy sets and IF–THEN rules to produce an output.

Main steps:

1. Fuzzification: Converts crisp input values into fuzzy membership values.
2. Rule evaluation: Applies fuzzy IF–THEN rules.
3. Aggregation: Combines the outputs of all rules.
4. Defuzzification: Converts the final fuzzy output into a single crisp value.

Example: Fan Speed Control

Input: Room temperature (0–40 °C). Output: Fan speed (0–100%).

- IF temperature is low, THEN fan speed is slow.
- IF temperature is medium, THEN fan speed is moderate.
- IF temperature is high, THEN fan speed is fast.

Python Program (Jupyter Notebook)

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

# Triangular membership function
def trimf(x, a, b, c):
    left = np.ones_like(x, dtype=float) if a == b else (x-a)/(b-a)
    right = np.ones_like(x, dtype=float) if b == c else (c-x)/(c-b)
    return np.maximum(0, np.minimum(left, right))

# Input: Temperature (0 to 40)
temp = np.arange(0, 41, 1)
low = trimf(temp, 0, 0, 20)
medium = trimf(temp, 10, 20, 30)
high = trimf(temp, 20, 40, 40)

# Output: Fan speed (0 to 100)
speed = np.arange(0, 101, 1)
slow = trimf(speed, 0, 0, 50)
moderate = trimf(speed, 25, 50, 75)
fast = trimf(speed, 50, 100, 100)

# Given room temperature
T = 30

# Fuzzification
mu_low = np.interp(T, temp, low)
mu_medium = np.interp(T, temp, medium)
mu_high = np.interp(T, temp, high)
```

```

# Mamdani rule evaluation (minimum)
r1 = np.minimum(mu_low, slow)
r2 = np.minimum(mu_medium, moderate)
r3 = np.minimum(mu_high, fast)

# Aggregation (maximum)
output = np.maximum(r1, np.maximum(r2, r3))

# Defuzzification (centroid method)
fan_speed = np.sum(speed * output) / np.sum(output)

print("Room Temperature:", T, "°C")
print("Low Membership:", round(mu_low, 2))
print("Medium Membership:", round(mu_medium, 2))
print("High Membership:", round(mu_high, 2))
print("Fan Speed:", round(fan_speed, 2), "%")

# Plot temperature membership functions
plt.figure(figsize=(10, 4))
plt.plot(temp, low, label="Low")
plt.plot(temp, medium, label="Medium")
plt.plot(temp, high, label="High")
plt.xlabel("Temperature (°C)")
plt.ylabel("Membership Degree")
plt.title("Temperature Membership Functions")
plt.legend()
plt.grid()
plt.show()

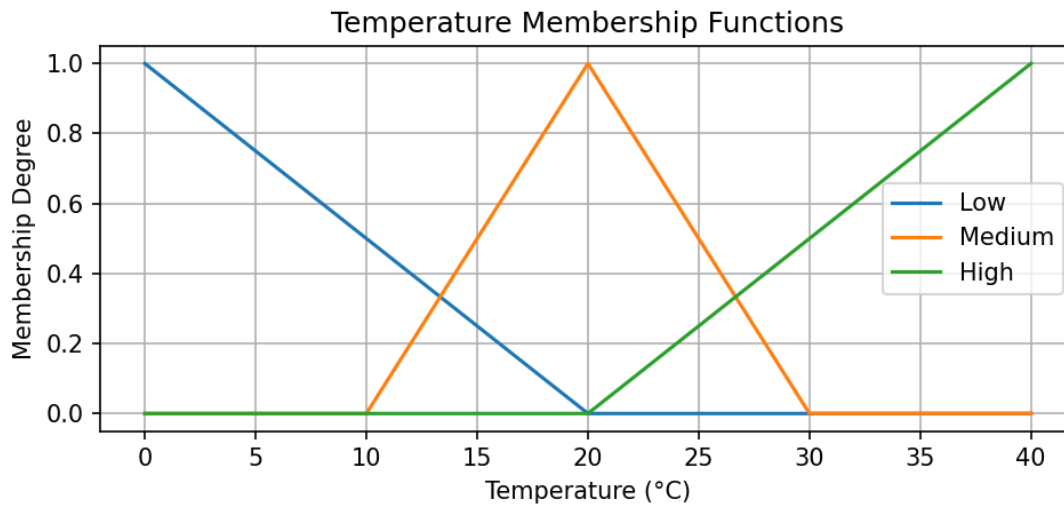
# Plot fuzzy output
plt.figure(figsize=(10, 4))
plt.plot(speed, output, label="Aggregated Output")
plt.axvline(fan_speed, linestyle="--",
            label="Defuzzified Fan Speed")
plt.xlabel("Fan Speed (%)")
plt.ylabel("Membership Degree")
plt.title("Mamdani Fuzzy Control Output")
plt.legend()
plt.grid()
plt.show()

```

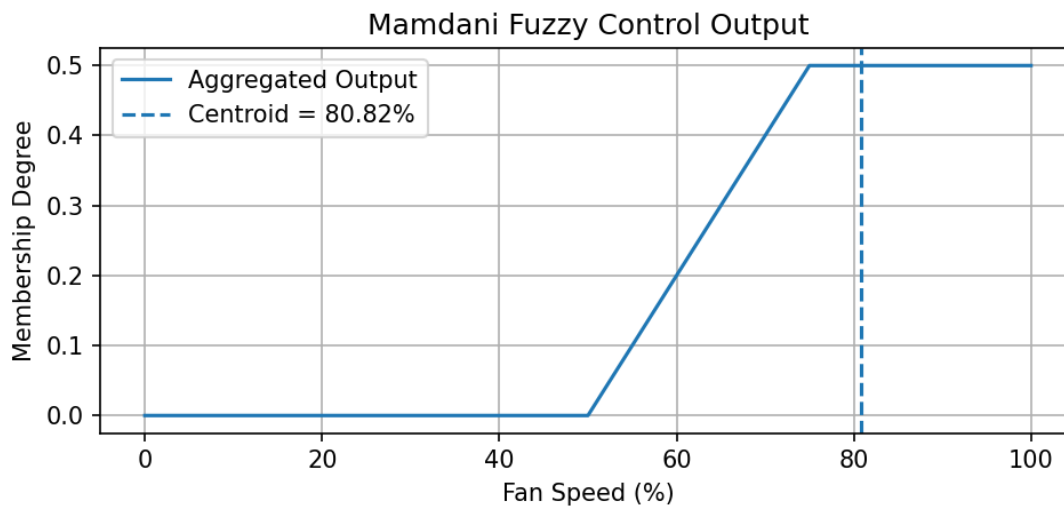
Output

Room Temperature: 30 °C
 Low Membership: 0.0
 Medium Membership: 0.0
 High Membership: 0.5
 Fan Speed: 80.82 %

Temperature Membership Function Diagram



Mamdani Fuzzy Control Output Diagram



Conclusion

The fuzzy control system was implemented using the Mamdani method. The program performs fuzzification, rule evaluation, aggregation, and defuzzification to determine fan speed based on room temperature.

EXPERIMENT NO. 9

IMPLEMENTATION OF GENETIC ALGORITHM

Aim

Implementation of Genetic Algorithm.

Theory

A Genetic Algorithm is an optimization technique inspired by natural selection. It finds a good solution by improving a population of possible solutions over several generations.

1. **Initialization:** Create a population of random solutions.
2. **Fitness evaluation:** Calculate the fitness of each solution.
3. **Selection:** Select the best solutions as parents.
4. **Crossover:** Combine parents to create new solutions.
5. **Mutation:** Make small random changes to maintain diversity.
6. **Termination:** Stop after a fixed number of generations and display the best solution.

Python Program (Jupyter Notebook)

This example uses a Genetic Algorithm to maximize $f(x) = x^2$, where x is an 8-bit binary number.

```
import random

# Genetic Algorithm to maximize  $f(x) = x^2$ 
POP_SIZE = 6
GEN = 10
BITS = 8

# Create random population
population = [
    ''.join(random.choice('01') for _ in range(BITS))
    for _ in range(POP_SIZE)
]

# Convert chromosome to decimal
def decode(chromosome):
    return int(chromosome, 2)

# Fitness function
def fitness(chromosome):
    x = decode(chromosome)
    return x * x

# Genetic Algorithm
for g in range(GEN):
    population.sort(key=fitness, reverse=True)
```

```

best = population[0]
print("Generation", g + 1,
      "Best x =", decode(best),
      "Fitness =", fitness(best))

# Selection: choose best half
parents = population[:POP_SIZE // 2]
new_population = parents.copy()

# Crossover and mutation
while len(new_population) < POP_SIZE:
    p1, p2 = random.sample(parents, 2)
    point = random.randint(1, BITS - 1)

    child = p1[:point] + p2[point:]

    # Mutation
    child = list(child)
    i = random.randint(0, BITS - 1)
    child[i] = '1' if child[i] == '0' else '0'

    new_population.append("".join(child))

population = new_population

# Final best solution
best = max(population, key=fitness)
print("\nBest Solution:", decode(best))
print("Maximum Fitness:", fitness(best))

```

Sample Output

```

Generation 1 Best x = 237 Fitness = 56169
Generation 2 Best x = 237 Fitness = 56169
Generation 3 Best x = 237 Fitness = 56169
Generation 4 Best x = 248 Fitness = 61504
Generation 5 Best x = 249 Fitness = 62001
Generation 6 Best x = 249 Fitness = 62001
Generation 7 Best x = 249 Fitness = 62001
Generation 8 Best x = 249 Fitness = 62001
Generation 9 Best x = 249 Fitness = 62001
Generation 10 Best x = 251 Fitness = 63001

```

```

Best Solution: 251
Maximum Fitness: 63001

```

Conclusion

The Genetic Algorithm was implemented using a population, fitness evaluation, selection, crossover, and mutation to search for the maximum value of a function.

DOP	DOS	REMARK	SIGN

Aim:

To design and implement an AI-powered Scrambled Word Puzzle Game using n8n, by creating an AI Agent and integrating it with workflow nodes to generate scrambled words, provide hints, process user answers, and calculate the score.

Theory:

n8n is a workflow automation platform used to connect different tasks and automate processes. In this experiment, an AI Agent is created in n8n to provide intelligent interaction with the user. The AI Agent can understand the user's input, process the request, and generate an appropriate response.

The workflow uses nodes such as a Trigger node, AI Agent node, Code node, and output node. The Code node handles the Scrambled Word Puzzle logic, including word selection, scrambling, hints, rounds, and score calculation. The AI Agent can interact with the user and assist with the game by understanding answers and providing suitable responses.

Working:

1. The workflow is started using a Trigger node.
2. The AI Agent receives and understands the user's input.
3. The Code node generates a random scrambled word and hint.
4. The user provides an answer, which is processed by the workflow.
5. The AI Agent can evaluate or respond to the user's input.
6. The workflow calculates the score and displays the result through the output node.

Thus, the experiment demonstrates how AI Agents and custom Code nodes can be combined in n8n to create an interactive and automated application.

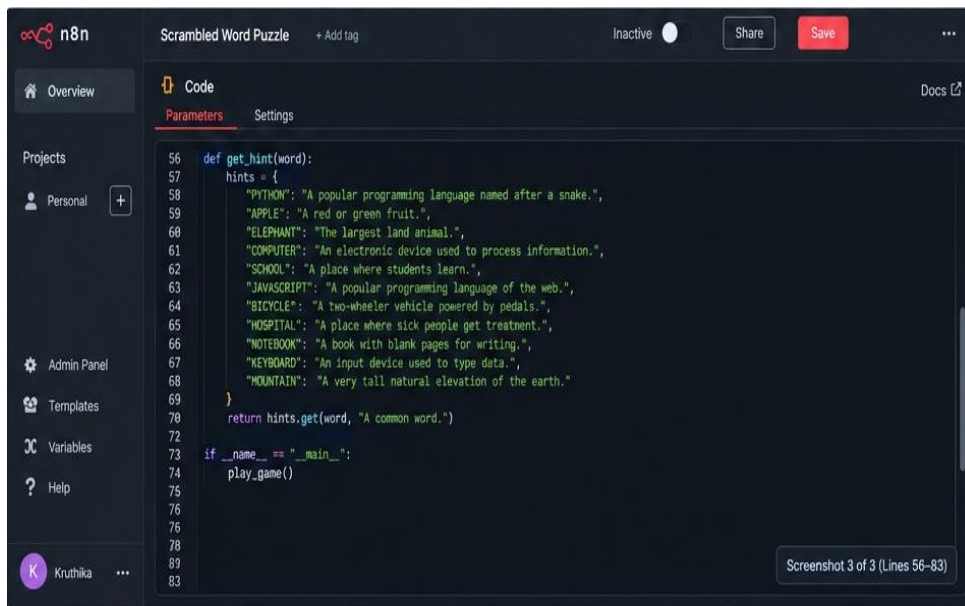
Code:

This screenshot shows the first part of the Python code for the 'Scrambled Word Puzzle' workflow in n8n. The interface includes a sidebar with navigation options like Overview, Projects, Admin Panel, and Templates. The main editor area displays the code for the 'Code' node, which is set to 'Run Once for All Items' mode and 'Python' language. The code defines a word bank, sets the number of rounds and points per round, and includes a function to scramble a word.

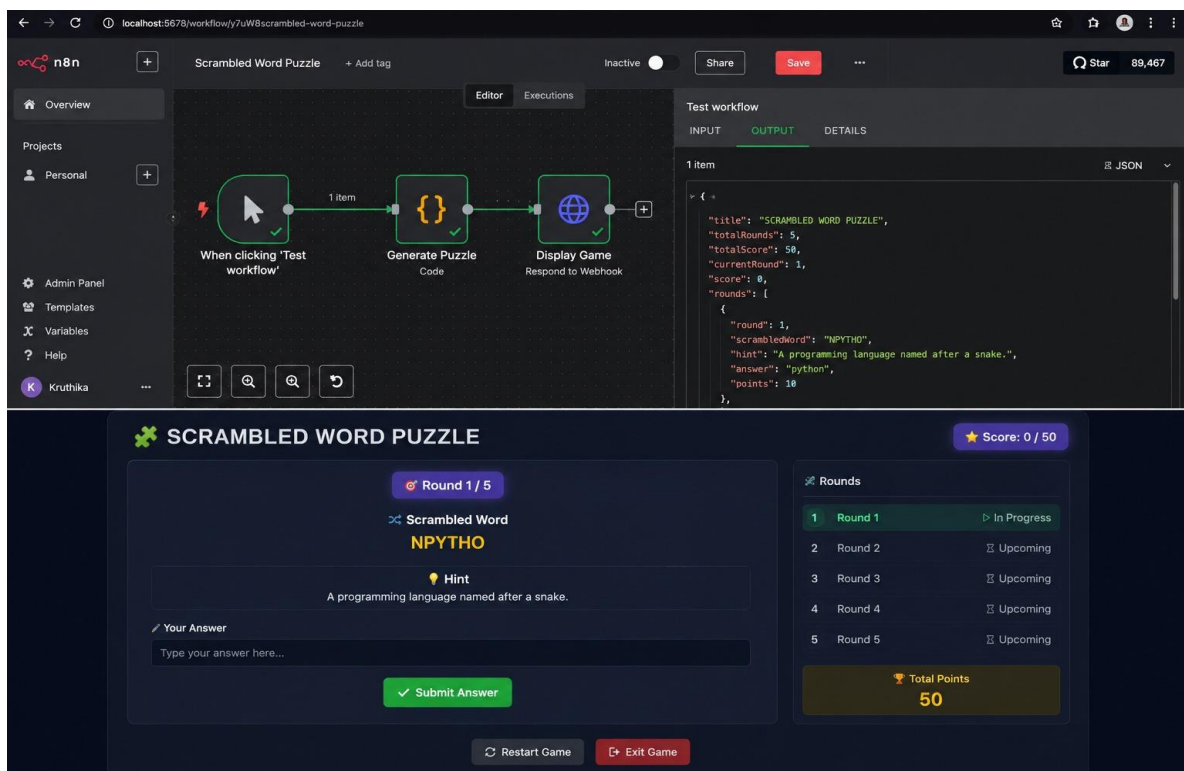
```
1 # Scrambled Word Puzzle Game - n8n Code Node
2 import random
3
4 # Word bank with 11 words
5 WORD_BANK = [
6     "PYTHON", "APPLE", "ELEPHANT", "COMPUTER", "SCHOOL",
7     "JAVASCRIPT", "BICYCLE", "HOSPITAL", "NOTEBOOK",
8     "KEYBOARD", "MOUNTAIN"
9 ]
10
11 ROUNDS = 5 # Total number of rounds
12 POINTS_PER_ROUND = 10 # Points for each correct answer
13
14 def scramble_word(word):
15     letters = list(word)
16     random.shuffle(letters)
17     return ''.join(letters)
18
19 # Function to play the game
20 def play_game():
21     score = 0
22     print("\n 🎮 SCRAMBLED WORD PUZZLE 🎮")
23     print("=====")
24     print(f"Total Rounds: {ROUNDS}")
25     print(f"Points per Round: {POINTS_PER_ROUND}")
26     print("\nLet's start the game!\n")
27
28
29
30
```

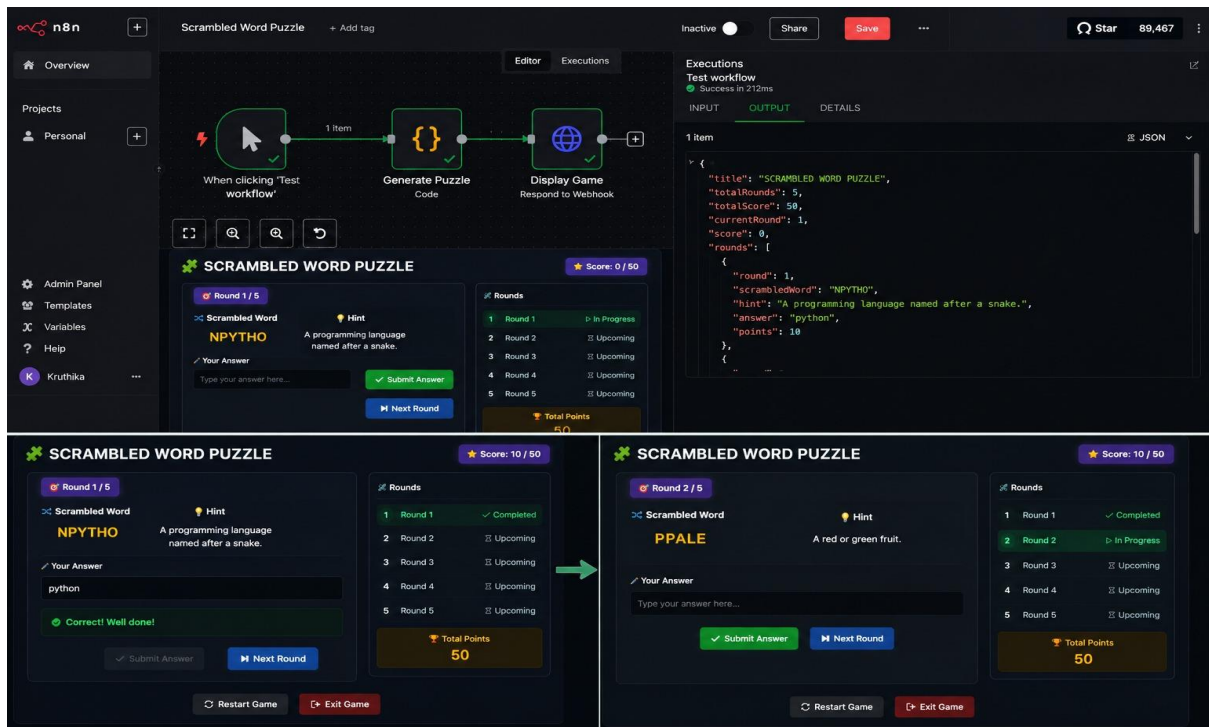
This screenshot shows the second part of the Python code, which contains the main game loop. It iterates through the rounds, selects a random word from the bank, scrambles it, and prompts the user for an answer. The code checks if the answer is correct and updates the score accordingly. It also includes a final result display and congratulatory messages based on the user's performance.

```
31 for i in range(1, ROUNDS + 1):
32     print(f"\nRound {i} of {ROUNDS}")
33     word = random.choice(WORD_BANK)
34     scrambled = scramble_word(word)
35     hint = get_hint(word)
36     print(f"Scrambled Word: {scrambled}")
37     print(f"Hint: {hint}")
38     answer = input("Your Answer: ").strip().upper()
39     if answer == word:
40         score += POINTS_PER_ROUND
41         print(" 🎉 Correct! Well done!")
42     else:
43         print(f" ❌ Wrong! Correct answer is: {word}")
44     print(f"Current Score: {score}/{ROUNDS * POINTS_PER_ROUND}")
45     print("-----")
46
47 # Show final result
48 print(f"\nGame Over! Your final score: {score}/{ROUNDS * POINTS_PER_ROUND}")
49 if score == ROUNDS * POINTS_PER_ROUND:
50     print(" 🏆 Perfect! Outstanding!")
51 elif score >= ROUNDS * POINTS_PER_ROUND / 2:
52     print(" 🌟 Great job! Keep it up!")
53 else:
54     print(" 💡 Keep Practicing! You can do better!")
55
```



Output:





CONCLUSION:

Thus, the AI-powered Scrambled Word Puzzle Game was successfully implemented using n8n. The AI Agent interacts with the user, while the Code node handles the puzzle logic, word scrambling, hints, rounds, and scoring. This experiment demonstrates the use of AI Agents and workflow automation in n8n to create an interactive application.