1. Identify the Peas Description And Task Environment for a Given Real World AI Problem.

—>

PEAS Description for Smart Home Assistant

Performance Measure

- User comfort and convenience (temperature, lighting, security)
- Energy efficiency and cost savings
- Response time to user commands
- System reliability and uptime
- User satisfaction scores
- Safety maintenance (detecting hazards like gas leaks, fire)
- Successful task completion rate

Environment

- Home interior spaces (rooms, hallways, etc.)
- Various smart devices (lights, thermostats, locks, cameras, appliances)
- Sensors (motion, temperature, humidity, door/window contacts)
- User presence and activities
- Time of day and schedules
- Weather conditions (for heating/cooling optimization)
- Network connectivity status

Actuators

- Smart lights (on/off, dimming, color control)
- HVAC systems (heating, cooling, fan control)
- Smart locks and door controls
- Window blinds/curtains
- Smart appliances (coffee maker, washing machine, etc.)
- Security cameras and alarm systems
- Smart speakers for audio feedback
- Display screens for visual information

Sensors

- Motion/occupancy sensors
- Temperature and humidity sensors
- Light sensors (ambient brightness)
- Door/window open-close sensors
- Microphones (for voice commands)
- Cameras (for facial recognition, activity detection)

- Smoke and CO detectors
- Water leak sensors
- Smart meters (electricity, gas usage)

Environment Type Classification

The Smart Home Assistant operates in an environment with the following characteristics:

1. **Partially Observable** - Cannot observe all aspects simultaneously (e.g., what's happening in every room at once, user intentions)

2. **Stochastic** - User behavior is unpredictable; sensor readings can vary; external factors (weather) are uncertain

3. **Sequential** - Current actions affect future states (e.g., turning on heating now affects temperature later)

4. **Dynamic** - Environment changes while the agent is deliberating (temperature fluctuates, people move around)

5. **Continuous** - Temperature, lighting levels, and time are continuous variables

6. **Multi-agent** - Multiple household members with potentially conflicting preferences; potentially coordinating with other smart systems

2. Identify suitable Agent Architecture and type for the problem

—>

Suitable Agent Type

Hybrid Agent: Model-Based Reflex + Goal-Based + Learning Agent

The smart home automation system requires a combination of agent types because of its diverse responsibilities:

1. **Model-Based Reflex Agent Component**

    - Maintains internal state of home conditions (temperature, occupancy, device status)
    - Uses if-then rules for immediate responses
    - Example: "IF motion detected AND lights off AND time > 6 PM, THEN turn on lights"
2. **Goal-Based Agent Component**

- Works towards specific objectives (energy savings, comfort optimization)
- Plans sequences of actions to achieve goals
- Example: Achieving target temperature by 7 AM when user wakes up

3. **Utility-Based Agent Component**

    - Makes trade-offs between conflicting goals
    - Maximizes overall satisfaction considering multiple factors
    - Example: Balancing comfort vs. energy cost

4. **Learning Agent Component**

    - Adapts to user preferences over time
    - Learns patterns and routines
    - Improves decision-making based on feedback

3.Implementation of Breadth first search for problem solving.

—>

```
from collections import deque

def bfs(graph, start):
	visited = set()
	queue = deque([start])
	while queue:
		node = queue.popleft()
		if node not in visited:
			print(node, end=" ")
			visited.add(node)
			queue.extend(graph[node] - visited)

graph = {
'A': {'B', 'C'},
'B': {'A', 'D', 'E'},
'C': {'A', 'F'},
'D': {'B'},
'E': {'B', 'F'},
'F': {'C', 'E'}
}

bfs(graph, 'A')
```

4. Implementation of Bidirectional search for problem solving.

—>

```
from collections import deque
```

```python
def bidirectional_search(graph, start, goal):
    if start == goal:
        return [start]

    front_start = deque([start])
    front_goal = deque([goal])
    visited_start = {start}
    visited_goal = {goal}

    while front_start and front_goal:
        node_start = front_start.popleft()
        for neighbor in graph[node_start]:
            if neighbor in visited_goal:
                print(f"Path found between {node_start} and {neighbor}")
                return
            if neighbor not in visited_start:
                visited_start.add(neighbor)
                front_start.append(neighbor)

        node_goal = front_goal.popleft()
        for neighbor in graph[node_goal]:
            if neighbor in visited_start:
                print(f"Path : {start_node} {neighbor} {node_goal}")
                return
            if neighbor not in visited_goal:
                visited_goal.add(neighbor)
                front_goal.append(neighbor)

    print("No path found")

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

start_node = 'A'
goal_node = 'F'
print("Start node : " + start_node)
print("Goal node : " + goal_node)
bidirectional_search(graph, start_node, goal_node)
```

5.Implement Hill Climbing Search for problem solving

—>

```
import random

def hill_climb(function, start, step_size=0.1, max_iterations=100):
    current = start
    for _ in range(max_iterations):
        # Generate a small random neighbor
        neighbor = current + random.uniform(-step_size, step_size)

        # If neighbor is better, move to it
        if function(neighbor) > function(current):
            current = neighbor
    return current

# Example: maximize f(x) = -(x-3)^2 + 9
def f(x):
    return -(x - 3)**2 + 9

result = hill_climb(f, start=random.uniform(0, 6))
print("Best solution found at x =", round(result, 2))
print("Maximum value =", round(f(result), 2))
```

6. Implementation of adversarial search using mini-max algorithm.

—>

```
def minimax(depth, node_index, is_maximizing, scores, height):
    # Base case: leaf node reached
    if depth == height:
        return scores[node_index]

    if is_maximizing:
        return max(
            minimax(depth + 1, node_index * 2, False, scores, height),
            minimax(depth + 1, node_index * 2 + 1, False, scores, height)
        )
    else:
        return min(
            minimax(depth + 1, node_index * 2, True, scores, height),
            minimax(depth + 1, node_index * 2 + 1, True, scores, height)
        )

# Example game tree leaf nodes (possible outcomes)
scores = [3, 5, 2, 9, 12, 5, 23, 23]
```

```python
tree_height = 3  # because 2^3 = 8 leaf nodes

# Start from root (depth=0, node_index=0, maximizing player)
best_value = minimax(0, 0, True, scores, tree_height)

print("The optimal value for the maximizing player is:", best_value)
```

7. Implement knowledge base in Prolog for solving Murder Mistry1) Husband and Alice was not together on the night of murder.

1.The killer and victim were on the beach.

2.On the night of murder, one male and one female was in the bar.

3.The victim was twin and the counterpart was innocent.

4.The killer was younger than the victim.

5.One child was alone at home.

—>

```prolog
lily(child)
person(john).
person(alice).
person(mark).
person(emma).
person(tom).
person(tim).
person(lily).
male(john).
male(mark).
male(tom).
male(tim).
female(alice).
female(emma).
female(lily).
husband(john).
child(lily).
at(john, beach).
at(tom, beach).    % victim was at beach
at(mark, bar).
at(emma, bar).     % one male and one female in the bar: mark (male), emma (female)
at(lily, home).    % one child was alone at home
twin(tom, tim).
innocent(tim) :- twin(tom, tim).  % counterpart innocent
younger(john, tom).   % killer younger than victim (we assert john < tom)
```

```
not_together(john, alice).
victim(tom).
killer(X) :-
    victim(V),
    at(X, beach),
    at(V, beach),
    younger(X, V),
    X \\= V,
    \\+ innocent(X).
```

queries :

% Example queries (in Prolog):

% ?- killer(X).   % expects X = john given the KB above

% ?- at(Person, beach). % list who was on beach

% ?- at(Person, bar).   % who was in bar

8. Implement family tree using prolog programming with different queries

—>

% --- Gender --- male(john). male(peter). male(sam).

female(linda). female(susan). female(anna).

% --- Parent relationships --- parent(john, peter). parent(linda, peter). parent(john, susan). parent(linda, susan). parent(peter, sam). parent(anna, sam).

% --- Rules --- father(X, Y) :- parent(X, Y), male(X). mother(X, Y) :- parent(X, Y), female(X). sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y. grandparent(X, Y) :- parent(X, Z), parent(Z, Y).

queries :-

?- father(john, peter). true.

?- mother(linda, susan). true.

?- sibling(peter, susan). true.

?- grandparent(john, sam). true.

?- parent(anna, sam). true.

9.Implementation of the bayes theorm

—>

P_D = 0.01 # Probability of having disease P_not_D = 1 - P_D # Probability of not having disease P_Pos_given_D = 0.99 # Probability of positive test given disease P_Pos_given_not_D = 0.05 # Probability of positive test given no disease

P_Pos = (P_Pos_given_D * P_D) + (P_Pos_given_not_D * P_not_D) P_D_given_Pos = (P_Pos_given_D * P_D) / P_Pos print("Probability of having disease given positive test: {:.4f}".format(P_D_given_Pos))