BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai – 400058-India
Department of Computer Engineering

Name	Rishabh Santosh Shenoy
UID no.	2023300222
Experiment No.	1
Subject.	AISC Lab

AIM:	1) To select an appropriate case study and define its PEAS (Performance Measure, Environment, Actuators, Sensors) description in detail. 2) To develop a program implementing either the Breadth-First Search (BFS) or Depth-First Search (DFS) technique using any programming language. 3) To implement the 8-puzzle problem solution using a heuristic-based algorithm.
Experiment 1A	
PROBLEM STATEMENT :	To design an AI-powered smart home assistant based on the PEAS framework that accurately interprets user commands, adapts to dynamic environments, and efficiently controls devices using sensors and actuators, ensuring energy efficiency, cost-effectiveness, and high user satisfaction.
THEORY:	The PEAS (Performance Measure, Environment, Actuators, Sensors) framework is used in Artificial Intelligence to define and analyze intelligent agent systems. Performance Measure specifies the criteria for evaluating an agent's success, such as accuracy, efficiency, or user satisfaction. Environment describes the conditions in which the agent operates, including physical, virtual, and dynamic factors. Actuators are the mechanisms through which the agent interacts with and affects its environment, like motors, displays, or APIs. Sensors enable the agent to perceive its environment, such as cameras, microphones, or data feeds. PEAS provides a structured approach to clearly define an AI system's goals, inputs, and outputs.
DOCUMENT:	

BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai – 400058-India
Department of Computer Engineering

PEAS (Performance measure , Environment , Actuator , Sensor) for AI smart assistant system :

Performance Measure :

The smart home assistant's success is measured by the accuracy of responses, user satisfaction, and energy efficiency. It must understand and execute commands correctly, provide helpful and personalized interactions, and optimize device usage to save energy. Balancing quick, relevant answers with sustainable home management ensures the assistant enhances comfort without wasting resources. Poor accuracy, slow response, or high energy use lowers overall performance. Cost must be minimized to build this product as less as possible.

Environment :

The assistant operates in a dynamic home environment with multiple rooms, devices, and users. It must adapt to varying lighting, noise levels, and user preferences, while integrating local appliances and cloud services. The environment includes physical spaces, connected devices, and internet-based information sources. Challenges include understanding commands in noisy settings, handling multiple users, and adapting to changes like new devices or routines within the smart home.

Actuators :

Actuators let the assistant affect the home environment by controlling devices. These include speakers for voice replies, smart plugs to power appliances, light switches for lighting control, and thermostat controllers for temperature adjustment. Through actuators, the assistant automates tasks, enhancing comfort and energy savings. Reliable integration with diverse hardware is essential to avoid errors that could inconvenience users or waste energy.

Sensors :

Sensors enable the assistant to perceive its surroundings and user commands. Microphones capture

voice input, while motion detectors sense occupancy to automate lighting or climate. Internet APIs provide external data like weather or news, enriching responses. Effective sensing ensures accurate understanding and timely actions. Poor sensor performance leads to misinterpretations, missed commands, and reduced automation benefits

Experiment 1B	
PROBLEM STATEMENT :	To design and implement a solution using BFS or DFS to explore all possible states or paths in a given problem domain, ensuring complete traversal and optimal or feasible result generation, with the approach applicable to scenarios such as the Travelling Salesman Problem or similar search-based tasks. To implement a Sudoku solver using the Depth-First Search (DFS) with backtracking approach, ensuring valid placement of numbers in a 9×9 grid according to Sudoku rules in Python.
THEORY:	BFS (Breadth-First Search) and DFS (Depth-First Search) are fundamental graph traversal techniques used to explore nodes and paths in search problems. BFS explores level by level, ensuring the shortest path in unweighted graphs, while DFS dives deep along each branch before backtracking. In heuristic algorithms, such as those solving optimization problems like the Travelling Salesman Problem, these methods can be combined with evaluation functions to guide the search toward promising solutions. Heuristics reduce the search space by estimating the cost to reach the goal, enabling more efficient exploration compared to blind search. This integration balances completeness, speed, and solution quality.

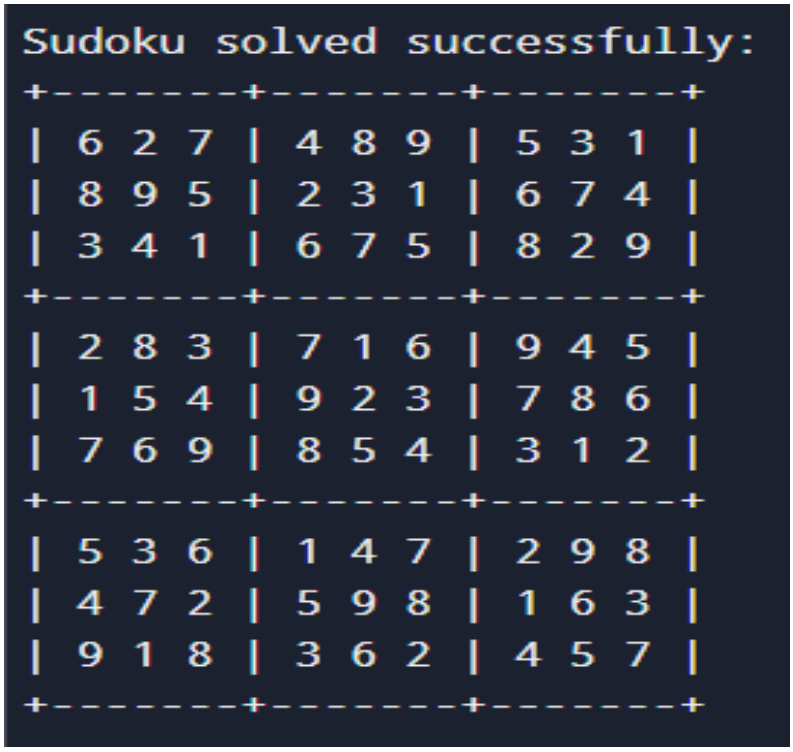
PROGRAM:	def is_safe(board, row, col, num):
	  for x in range(9):
	  if board[row][x] == num:
	  return False
	  for x in range(9):
	  if board[x][col] == num:
	  return False
	
	  start_row = row - row % 3
	  start_col = col - col % 3

```
    for i in range(3):
        for j in range(3):
            if board[start_row + i][start_col + j] == num:
                return False
    return True

def solve_sudoku(board):
    for row in range(9):
        for col in range(9):
            if board[row][col] == 0:
                for num in range(1, 10):
                    if is_safe(board, row, col, num):
                        board[row][col] = num
                        if solve_sudoku(board):
                            return True
                        board[row][col] = 0
                return False
    return True

def print_board(board):
    for i in range(9):
        if i==0:
            print()
            # print("|")
            print("+"+"-"*7+"+"+"-"*7+"+"+"-"*7+"+")
        for j in range(9):
            if j==0:
                print("|",end=" ")
            print(board[i][j], end=" ")
            if j % 3 == 2:
                print("|", end=" ")
        if i % 3 == 2:
            print()
            print("+"+"-"*7+"+"+"-"*7+"+"+"-"*7+"+")
        else:
            print()
        #print("_________")
```

```
board = [
   [0,0,7,0,0,0,5,0,0],
   [0,0,0,0,3,0,0,0,4],
   [0,4,0,0,0,5,8,0,0],
   [0,0,0,0,1,0,9,4,0],
   [0,0,0,9,2,0,0,0,6],
   [0,0,0,8,0,4,0,0,0],
   [5,0,0,0,0,7,0,0,0],
   [0,0,2,0,9,0,1,0,0],
   [9,1,0,0,0,0,0,5,0]
]

if solve_sudoku(board):
   print("Sudoku solved successfully:")
   print_board(board)
else:
   print("No solution exists")
```

RESULT:

```
Input Sudoku:
+-------+-------+-------+
| 0 0 7 | 0 0 0 | 5 0 0 |
| 0 0 0 | 0 3 0 | 0 0 4 |
| 0 4 0 | 0 0 5 | 8 0 0 |
+-------+-------+-------+
| 0 0 0 | 0 1 0 | 9 4 0 |
| 0 0 0 | 9 2 0 | 0 0 6 |
| 0 0 0 | 8 0 4 | 0 0 0 |
+-------+-------+-------+
| 5 0 0 | 0 0 7 | 0 0 0 |
| 0 0 2 | 0 9 0 | 1 0 0 |
| 9 1 0 | 0 0 0 | 0 5 0 |
+-------+-------+-------+
```



Experiment 1C	
PROBLEM STATEMENT:	To implement the 8-puzzle problem solution using a heuristic-based search algorithm, such as A* or Best-First Search, which leverages an evaluation function to efficiently find the shortest sequence of moves from the initial state to the goal state, ensuring optimal performance and reduced search space compared to uninformed methods.
THEORY:	This program solves the 8-puzzle problem using the A* search algorithm with two heuristics: h_1 (Misplaced Tiles) and h_2 (Manhattan Distance). Each puzzle state is represented as a node containing its path cost g(n), heuristic values, and total estimated costs $f_1(n)$ and $f_2(n)$. The algorithm uses a priority queue (min-heap) to explore the most promising states first, expanding nodes with the lowest cost estimate. At each step, it generates valid child states by moving the blank tile in four directions. The search continues until the goal configuration is reached, providing an optimal path while minimizing explored states.
PROGRAM:	#print("Hello") import heapq import copy

```
goal_state = [[1, 2, 3],
              [4, 5, 6],
              [7, 8, 0]]

moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

class PuzzleNode:
    def __init__(self, state, parent=None, g=0):
        self.state = state
        self.parent = parent
        self.g = g
        self.h1 = self.calculate_h1()
        self.h2 = self.calculate_h2()
        self.f1 = self.g + self.h1
        self.f2 = self.g + self.h2

    def __lt__(self, other):
        return self.f2 < other.f2

    def calculate_h1(self):
        misplaced = 0
        for i in range(3):
            for j in range(3):
                if self.state[i][j] != 0 and self.state[i][j] != goal_state[i][j]:
                    misplaced += 1
        return misplaced

    def calculate_h2(self):
        distance = 0
        for i in range(3):
            for j in range(3):
                val = self.state[i][j]
                if val != 0:
                    target_x = (val - 1) // 3
                    target_y = (val - 1) % 3
                    distance += abs(i - target_x) + abs(j - target_y)
        return distance
```

```
    def find_zero(self):
        for i in range(3):
            for j in range(3):
                if self.state[i][j] == 0:
                    return i, j

    def get_children(self):
        children = []
        x, y = self.find_zero()

        for dx, dy in moves:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_state = copy.deepcopy(self.state)
                new_state[x][y], new_state[new_x][new_y] =
new_state[new_x][new_y], new_state[x][y]
                children.append(PuzzleNode(new_state, self, self.g + 1))
        return children

    def is_goal(self):
        return self.state == goal_state

    def print_state(self):
        for row in self.state:
            print(row)
        print()

def a_star_search(start_state):
    print("Welcome to Rishabh's 8 Puzzle Problem Solving !!")
    start_node = PuzzleNode(start_state)
    frontier = []
    heapq.heappush(frontier, start_node)
    explored = set()
    step = 0

    while frontier:
        current_node = heapq.heappop(frontier)
        state_id = str(current_node.state)
```

```
        if state_id in explored:
            continue
        explored.add(state_id)



        print(f"\nStep number {step} is \n")
        current_node.print_state()
        print(f"g(n) = {current_node.g}")
        print(f"h1 (Misplaced Tiles) = {current_node.h1}")
        print(f"h2 (Manhattan Distance) = {current_node.h2}")
        print(f"f1(n) = g(n) + h1 = {current_node.f1}")
        print(f"f2(n) = g(n) + h2 = {current_node.f2}")

        if current_node.is_goal():
            print("\nGoal state reached!")
            print("Final path:")
            print_solution_path(current_node)
            return

        for child in current_node.get_children():
            if str(child.state) not in explored:
                heapq.heappush(frontier, child)

        step += 1

    print("No solution found.")
    print("Please try with other example")

def print_solution_path(node):
    path = []
    while node:
        path.append(node)
        node = node.parent
    path.reverse()

    for idx, step in enumerate(path):
        print(f"\nMove {idx}:")
```

```
        step.print_state()
        print(f"g(n) = {step.g}, h1(n) = {step.h1}, f1(n) = {step.f1}")
        print(f"g(n) = {step.g}, h2(n) = {step.h2}, f2(n) = {step.f2}")

# print("First Problem : \n")
# initial_state = [[1, 2, 3],
#                  [0, 4, 6],
#                  [7, 5, 8]]

# a_star_search(initial_state)

# print("Second Problem : \n")
# initial_state=[[1 ,2 ,3 ],
# [0 ,4 ,6 ],
# [7 ,5 ,8]]

# a_star_search(initial_state)

# print("Third Problem : \n")
# initial_state=[[1 ,2 ,3 ],
# [4 ,0 ,5 ],
# [6 ,7 ,8]]

# a_star_search(initial_state)
```

RESULT:

```
Hello
First Problem :

Welcome to Rishabh's 8 Puzzle Problem Solving !!

Step number 0 is

[1, 2, 3]
[0, 4, 6]
[7, 5, 8]

g(n) = 0
h1 (Misplaced Tiles) = 3
h2 (Manhattan Distance) = 3
f1(n) = g(n) + h1 = 3
f2(n) = g(n) + h2 = 3

Step number 1 is

[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

g(n) = 1
h1 (Misplaced Tiles) = 2
h2 (Manhattan Distance) = 2
f1(n) = g(n) + h1 = 3
f2(n) = g(n) + h2 = 3
```

```
Step number 2 is

[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

g(n) = 2
h1 (Misplaced Tiles) = 1
h2 (Manhattan Distance) = 1
f1(n) = g(n) + h1 = 3
f2(n) = g(n) + h2 = 3

Step number 3 is

[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

g(n) = 3
h1 (Misplaced Tiles) = 0
h2 (Manhattan Distance) = 0
f1(n) = g(n) + h1 = 3
f2(n) = g(n) + h2 = 3

Goal state reached!
Final path:

Move 0:
[1, 2, 3]
```

```
[0, 4, 6]
[7, 5, 8]

g(n) = 0, h1(n) = 3, f1(n) = 3
g(n) = 0, h2(n) = 3, f2(n) = 3

Move 1:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

g(n) = 1, h1(n) = 2, f1(n) = 3
g(n) = 1, h2(n) = 2, f2(n) = 3

Move 2:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

g(n) = 2, h1(n) = 1, f1(n) = 3
g(n) = 2, h2(n) = 1, f2(n) = 3

Move 3:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

g(n) = 3, h1(n) = 0, f1(n) = 3
g(n) = 3, h2(n) = 0, f2(n) = 3

=== Code Execution Successful ===
```

```
Final path:

Move 0:
[1, 2, 3]
[4, 0, 5]
[6, 7, 8]

g(n) = 0, h1(n) = 4, f1(n) = 4
g(n) = 0, h2(n) = 6, f2(n) = 6

Move 1:
[1, 2, 3]
[4, 5, 0]
[6, 7, 8]

g(n) = 1, h1(n) = 3, f1(n) = 4
g(n) = 1, h2(n) = 5, f2(n) = 6

Move 2:
[1, 2, 3]
[4, 5, 8]
[6, 7, 0]

g(n) = 2, h1(n) = 3, f1(n) = 5
g(n) = 2, h2(n) = 6, f2(n) = 8

Move 3:
[1, 2, 3]
[4, 5, 8]
[6, 0, 7]

g(n) = 3, h1(n) = 3, f1(n) = 6
g(n) = 3, h2(n) = 7, f2(n) = 10

Move 4:
[1, 2, 3]
[4, 5, 8]
[0, 6, 7]

g(n) = 4, h1(n) = 3, f1(n) = 7
g(n) = 4, h2(n) = 6, f2(n) = 10
```

```
Move 5:
[1, 2, 3]
[0, 5, 8]
[4, 6, 7]

g(n) = 5, h1(n) = 4, f1(n) = 9
g(n) = 5, h2(n) = 7, f2(n) = 12

Move 6:
[1, 2, 3]
[5, 0, 8]
[4, 6, 7]

g(n) = 6, h1(n) = 5, f1(n) = 11
g(n) = 6, h2(n) = 8, f2(n) = 14

Move 7:
[1, 2, 3]
[5, 6, 8]
[4, 0, 7]

g(n) = 7, h1(n) = 5, f1(n) = 12
g(n) = 7, h2(n) = 7, f2(n) = 14

Move 8:
[1, 2, 3]
[5, 6, 8]
[4, 7, 0]

g(n) = 8, h1(n) = 5, f1(n) = 13
g(n) = 8, h2(n) = 6, f2(n) = 14

Move 9:
[1, 2, 3]
[5, 6, 0]
[4, 7, 8]

g(n) = 9, h1(n) = 5, f1(n) = 14
g(n) = 9, h2(n) = 5, f2(n) = 14
```

```
Move 10:
[1, 2, 3]
[5, 0, 6]
[4, 7, 8]

g(n) = 10, h1(n) = 4, f1(n) = 14
g(n) = 10, h2(n) = 4, f2(n) = 14

Move 11:
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]

g(n) = 11, h1(n) = 3, f1(n) = 14
g(n) = 11, h2(n) = 3, f2(n) = 14

Move 12:
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]

g(n) = 12, h1(n) = 2, f1(n) = 14
g(n) = 12, h2(n) = 2, f2(n) = 14

Move 13:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

g(n) = 13, h1(n) = 1, f1(n) = 14
g(n) = 13, h2(n) = 1, f2(n) = 14

Move 14:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

g(n) = 14, h1(n) = 0, f1(n) = 14
g(n) = 14, h2(n) = 0, f2(n) = 14
psipl@psipl-OptiPlex-SFF-7010:~/Documents/Rishabh_Shenoy_2023300222$
```

CONCLUSION:	I have understood that the 8-puzzle problem can be effectively solved using the A* search algorithm with heuristics like Misplaced Tiles (h_1) and Manhattan Distance (h_2). The algorithm works by combining the actual cost from the start node (g(n)) with the estimated cost to the goal (h(n)) to decide which state to explore next. I learned that the Manhattan Distance heuristic is usually more accurate,

	leading to fewer steps compared to Misplaced Tiles. I also understood the importance of using a priority queue to pick the most promising state and avoiding revisiting already explored states. This approach ensures optimal solutions with reduced search space.