Contents

[Question no. 1 jealousy 2](#_Toc188467129)

[Question no. 2 in\_laws 2](#_Toc188467130)

[Question no 3 friendship 4](#_Toc188467131)

[Question no 4 represent graph 4](#_Toc188467132)

[Question 5 factorial 5](#_Toc188467133)

[Linear Regression 6](#_Toc188467134)

[Vacuum cleaner reflex agent 8](#_Toc188467135)

[Perceptron Training for AND gate 9](#_Toc188467136)

[8-puzzle 11](#_Toc188467137)

[Block arrangement problem 13](#_Toc188467138)

# Question no. 1 jealousy

male(ram).

male(rawan).

male(ramesh).

female(sita).

female(geeta).

loves(ram,sita).

loves(rawan,sita).

loves(sita,ramesh).

loves(geeta,ramesh).

loves(geeta,sita).

jealousy(X,Y):-loves(X,Z),loves(Y,Z),X\=Y,opposite\_gender(X,Z),opposite\_gender(Y,Z).

opposite\_gender(X,Y):-male(X),female(Y);male(Y),female(X).

A screenshot of a computer

Description automatically generated

# Question no. 2 in\_laws

male(ram).

male(hari).

male(rajesh).

female(anita).

female(sita).

female(geeta).

parent(gopal,ram).

parent(gopal,anita).

parent(gopal,gita).

parent(shema,hari).

parent(shema,sita).

parent(shema,rajesh).

married(anita,hari).

married(hari,anita).

married(sita,ram).

married(ram,sita).

brother\_in\_law(X,Y):-

(male(X),married(X,Z),sibling(Z,Y));

(male(X),married(Z,Y),sibling(X,Z));

(male(X),married(X,Z),married(Z,W),sibling(W,Y)).

sister\_in\_law(X,Y):-

(female(X),married(X,Z),sibling(Z,Y));

(female(X),married(Z,Y),sibling(X,Z));

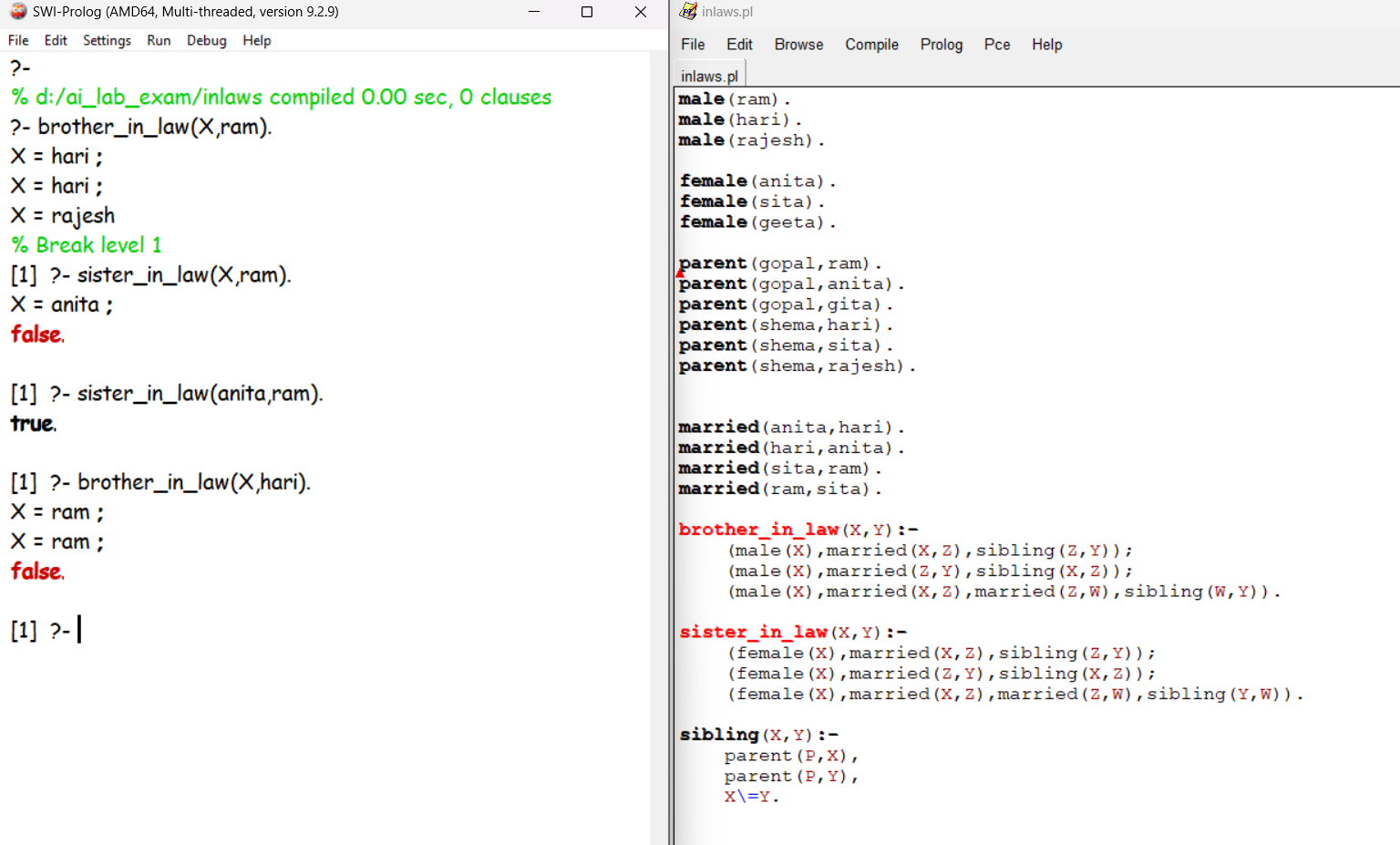
(female(X),married(X,Z),married(Z,W),sibling(Y,W)).

sibling(X,Y):-

parent(P,X),

parent(P,Y),

X\=Y.



# Question no 3 friendship

friend(ram,krishna).

friend(ram,bishnu).

friend(ram,bimal).

friend(bishnu,piyush).

friend(bishnu,ramesh).

friend(piyush,nirmal).

friend(bimal,kamal).

%friend(X,Y):-friend(Y,X).

%this is for if ram is friend fo bishnu , bishnu is also friend of ram

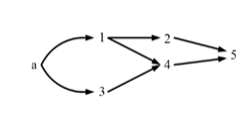
friendoffriend(X,Z):-friend(X,Z).

friendoffriend(X,Z):-friend(X,Y),friendoffriend(Y,Z).

A screenshot of a computer

Description automatically generated

# Question no 4 represent graph



edge(a,1).

edge(a,3).

edge(1,2).

edge(1,4).

edge(3,4).

edge(2,5).

edge(4,5).

route(Vertex1,Vertex2):-

edge(Vertex1,Vertex2),

%print route

write(Vertex1),

write('->'),

write(Vertex2).

route(Vertex1,Vertex2):-

edge(Vertex1,CommonVertex),

write(Vertex1),

write('->'),

route(CommonVertex,Vertex2).

A screenshot of a computer

Description automatically generated

# Question 5 factorial

factorial(1,1):-!.

factorial(N,F):-N>1,N1 is N-1,factorial(N1,F1),F is N\*F1.

A screenshot of a computer

Description automatically generated

# Linear Regression

import numpy as np

import statistics as stats

# Data

#dataX = np.array([6, 12, 14, 6, 9, 13, 15, 9])

#dataY = np.array([300, 400, 560, 250, 290, 650, 630, 520])

# data of some xyz group

dataX = np.array([93.33, 86.67, 86.67, 6.67, 80, 80, 60, 73.33,46.67,46.67,86.67,73.33,66.67,93.33,80,66.67,13.33,80,26.67,46.67])

dataY = np.array([28,25,22,25,24,26,16,21,26,17,27,23,19,25,23,9,18,26,14,7])

X = 80

# Calculate averages

dataXAvg = np.mean(dataX)

dataYAvg = np.mean(dataY)

# Deviation from the mean

x = dataX - dataXAvg

y = dataY - dataYAvg

# Calculate mode

modeX = stats.mode(dataX)  # Accessing the first mode value of dataX

modeY = stats.mode(dataY)  # Accessing the first mode value of dataY

# Calculate median

medianX = np.median(dataX)

medianY = np.median(dataY)

# Calculate variance

varianceX = np.var(dataX)

varianceY = np.var(dataY)

# Calculate standard deviation

stdDevX = np.std(dataX)

stdDevY = np.std(dataY)

# Calculate products and squares

xy = x \* y

xSqr = x \*\* 2

ySqr = y \*\* 2

# Summations

sumxy = np.sum(xy)

sumxSqr = np.sum(xSqr)

sumySqr = np.sum(ySqr)

# Correlation coefficient

rCoefficient = sumxy / np.sqrt(sumxSqr \* sumySqr)

# Regression coefficients

b = sumxy / sumxSqr

a = dataYAvg - b \* dataXAvg

# Predict Y for given X

Y = a + b \* X

# Output results

print("Correlation Coefficient (r):", rCoefficient)

print("Intercept (a):", a)

print("Slope (b):", b)

print("Predicted Y for X =", X, ":", Y)

# Additional statistical measures

print("\nStatistics for dataX:")

print("Mean:", dataXAvg)

print("Mode:", modeX)

print("Median:", medianX)

print("Variance:", varianceX)

print("Standard Deviation:", stdDevX)

print("\nStatistics for dataY:")

print("Mean:", dataYAvg)

print("Mode:", modeY)

print("Median:", medianY)

print("Variance:", varianceY)

print("Standard Deviation:", stdDevY)

# Vacuum cleaner reflex agent

import random  
  
class RoomCleanerAgent:  
    def \_\_init\_\_(self, room\_size=(10, 10)):  
        self.room\_size = room\_size  
        # Initialize the room as a 10x10 grid with random 0 (clean) and 1 (dirty) cells  
        self.grid = [[random.choice([0, 1]) for \_ in range(room\_size[1])] for \_ in range(room\_size[0])]  
        # Initialize the agent's position randomly  
        self.current\_position = (random.randint(0, room\_size[0] - 1), random.randint(0, room\_size[1] - 1))  
  
    def display\_room(self):  
        # Display the current status of the room grid  
        for row in self.grid:  
          for cell in row):  
            print(' '.join(str(cell)))  
          print("\n")  
  
    def perceive(self):  
        # Perceive the cleanliness of the current cell  
        x, y = self.current\_position  
        return self.grid[x][y]  
  
    def act(self):  
        # Perform action based on the perception (clean the cell if dirty)  
        x, y = self.current\_position  
        if self.perceive() == 1:  # If the current cell is dirty (1)  
            print(f"Cell ({x}, {y}) is Dirty. Cleaning...")  
            self.grid[x][y] = 0  # Clean the cell (set to 0)  
            print(f"Cell ({x}, {y}) is now Clean.")  
        else:  
            print(f"Cell ({x}, {y}) is already Clean.")  
  
    def move(self):  
        # Systematic movement to cover the entire grid row by row  
        x, y = self.current\_position  
        if y < self.room\_size[1] - 1:  # Move to the next cell in the same row  
            self.current\_position = (x, y + 1)  
        elif x < self.room\_size[0] - 1:  # Move to the first cell of the next row  
            self.current\_position = (x + 1, 0)  
        else:  
            self.current\_position = None  # All cells have been visited  
  
    def is\_room\_clean(self):  
        # Check if the entire room is clean  
        return all(cell == 0 for row in self.grid for cell in row)  
  
    def run(self):  
        # Display initial status of the room  
        print("Initial Room Status:")  
        self.display\_room()  
  
        steps = 0  
        while not self.is\_room\_clean():  
            print(f"\nStep {steps + 1}:")  
            self.act()  
            self.move()  
            steps += 1  
            if self.current\_position is None:  
                # Restart from the top-left corner if needed to ensure all cells are visited  
                self.current\_position = (0, 0)  
  
        # Display final status of the room  
        print("\nFinal Room Status:")  
        self.display\_room()  
        print(f"Room cleaned in {steps} steps.")  
  
# Create and run the Room Cleaner Agent  
agent = RoomCleanerAgent()  
agent.run()

# Perceptron Training for AND gate

**(sir kei code ho can refer to sir ko mail if it doesn’t work)**

import numpy as np

# import matplotlib.pyplot as plt

# Training data for AND gate

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

y = np.array([0, 0, 0, 1])

def step(x,th):

  if x>=th:

    return(1)

  return(0)

def linearInt(x):

  return (round(x))

def predict(X, y, weights):

  print("X\tActual\tPredicted")

  for x\_input, y\_output in zip(X, y):

    inSum=np.sum(x\_input \* weights)

    y\_pred = step(inSum,th)

    print(x\_input,"\t",y\_output,"\t",y\_pred)

# initialize constants

lr = 0.1

th = 0.5

# Initialize weights array

weights = []

# Loop through each element in X

for i in range(X.shape[1]): #+1 for te bias

    # Initialize w randomly between 0 and 1 using Python's random module & Convert w to have only one digit after the decimal point

    w = round(np.random.rand(),1)

    weights.append(w)

print("Randomly initialized weights for each input:")

for i in range(len(weights)):

    print(f"weights[{i}]:", weights[i])

# initialize constants

lr = 0.1

th = 0.5

# Initialize weights array

weights = []

# Loop through each element in X

for i in range(X.shape[1]): #+1 for te bias

    # Initialize w randomly between 0 and 1 using Python's random module & Convert w to have only one digit after the decimal point

    w = round(np.random.rand(),1)

    weights.append(w)

print("Randomly initialized weights for each input:")

for i in range(len(weights)):

    print(f"weights[{i}]:", weights[i])

#For AND/OR Logic Gates

# print (step(0.5,th))

iterateFlag = True

while (iterateFlag):

  iterateFlag = False

  for x\_input, y\_output in zip(X, y):

    inSum=np.sum(x\_input \* weights)

    y\_pred = step(inSum,th)

    err = y\_output - y\_pred

    if(err!=0):

      iterateFlag = True

      for i in range(len(weights)):

        dw = lr \* x\_input[i] \* err

        weights[i] = weights[i] + dw

      print("input:",x\_input, "actual output:",y\_output,"predicted output: ",y\_pred,"updated weights:", weights)

print("Final weights:", weights)

# 8-puzzle

**(geeksforgeeks ko modified wala ho)**

import heapq

# Class to represent the state of the 8-puzzle

class PuzzleState:

    def \_\_init\_\_(self, board, parent, move, depth, cost):

        self.board = board  # The puzzle board configuration

        self.parent = parent  # Parent state

        self.move = move  # Move to reach this state

        self.depth = depth  # Depth in the search tree

        self.cost = cost  # Cost (depth + heuristic)

    def \_\_lt\_\_(self, other):

        return self.cost < other.cost

# Function to display the board in a visually appealing format

def print\_board(board):

    print("+---+---+---+")

    for row in range(0, 9, 3):

        row\_visual = "|"

        for tile in board[row:row + 3]:

            if tile == 0:  # Blank tile

                row\_visual += f"   |"

            else:

                row\_visual += f" {str(tile)} |"

        print(row\_visual)

        print("+---+---+---+")

# Goal state for the puzzle

goal\_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]

# Possible moves for the blank tile (up, down, left, right)

moves = {

    'U': -3,  # Move up

    'D': 3,   # Move down

    'L': -1,  # Move left

    'R': 1    # Move right

}

# Function to calculate the heuristic (Manhattan distance)

def heuristic(board):

    distance = 0

    for i in range(9):

        if board[i] != 0:

            x1, y1 = divmod(i, 3)

            x2, y2 = divmod(board[i] - 1, 3)

            distance += abs(x1 - x2) + abs(y1 - y2)

    return distance

# Function to get the new state after a move

def move\_tile(board, move, blank\_pos):

    new\_board = board[:]

    new\_blank\_pos = blank\_pos + moves[move]

    new\_board[blank\_pos], new\_board[new\_blank\_pos] = new\_board[new\_blank\_pos], new\_board[blank\_pos]

    return new\_board

# A\* search algorithm

def a\_star(start\_state):

    open\_list = []

    closed\_list = set()

    heapq.heappush(open\_list, PuzzleState(start\_state, None, None, 0, heuristic(start\_state)))

    while open\_list:

        current\_state = heapq.heappop(open\_list)

        if current\_state.board == goal\_state:

            return current\_state

        closed\_list.add(tuple(current\_state.board))

        blank\_pos = current\_state.board.index(0)

        for move in moves:

            if move == 'U' and blank\_pos < 3:  # Invalid move up

                continue

            if move == 'D' and blank\_pos > 5:  # Invalid move down

                continue

            if move == 'L' and blank\_pos % 3 == 0:  # Invalid move left

                continue

            if move == 'R' and blank\_pos % 3 == 2:  # Invalid move right

                continue

            new\_board = move\_tile(current\_state.board, move, blank\_pos)

            if tuple(new\_board) in closed\_list:

                continue

            new\_state = PuzzleState(new\_board, current\_state, move, current\_state.depth + 1, current\_state.depth + 1 + heuristic(new\_board))

            heapq.heappush(open\_list, new\_state)

    return None

# Function to print the solution path

def print\_solution(solution):

    path = []

    current = solution

    while current:

        path.append(current)

        current = current.parent

    path.reverse()

    for step in path:

        print(f"Move: {step.move}")

        print\_board(step.board)

# Initial state of the puzzle

initial\_state = [1, 2, 3, 4, 5, 0, 6, 7, 8]

# Solve the puzzle using A\* algorithm

solution = a\_star(initial\_state)

# Print the solution

if solution:

    print("Solution found:")

    print\_solution(solution)

else:

    print("No solution exists.")

# Block arrangement problem

def calculate\_heuristic(start\_state, goal\_state):

"""

Calculates the heuristic value for the Blocks World problem.

Args:

start\_state (list): A list of blocks from top to bottom in the start state.

goal\_state (list): A list of blocks from top to bottom in the goal state.

Returns:

int: The heuristic value of the state.

"""

heuristic = 0

n = len(start\_state)

for i in range(n):

if start\_state[i:] == goal\_state[i:]:

*# If the support structure matches the goal*

heuristic += 1

break *# Remaining structure is correct*

else:

*# If the support structure does not match*

heuristic -= 1

return heuristic

*# Example: Define the start and goal states*

start\_state = ["A", "D", "C", "B"] *# Top to bottom*

goal\_state = ["D", "C", "B", "A"] *# Top to bottom*

*# Calculate and print the heuristic value*

heuristic\_value = calculate\_heuristic(start\_state, goal\_state)

print(f"Heuristic Value: {heuristic\_value}")