

# JOURNAL

IN THE MAJOR COURSE

# ARTIFICIAL INTELLIGENCE

SUBMITTED BY

# ROBIN DUBEY

# OF

# SY BSc. IT

**SDIT-009A**

SEMESTER IV

UNDER THE GUIDANCE OF

Asst. Professor Prashant Chaubey

ACADEMIC YEAR

# 2023 – 2024

# 

**CERTIFICATE**

This is to certify that **ROBIN DUBEY** of **SECOND** year of **INFORMATION TECHNOLOGY.** Div.: **A** Roll No. **SDIT009A** of Semester **IV** (2023 - 2024) has successfully completed the Journal for the Major course **ARTIFICIAL INTELLIGENCE** as per the guidelines of KES’ Shroff College of Arts and Commerce, Kandivali(W), Mumbai- 400067.

Teacher In-charge: Principal:

**Asst. Prof. Prashant Chaubey** **Dr. Lily Bhushan**

**INDEX PAGE**

|  |  |  |  |
| --- | --- | --- | --- |
| **S no.** | **Aim** | **Date** | **Faculty Sign** |
| **1** | **Write a program to implement depth first search algorithm.** |  |  |
| **2** | **Write a program to implement breadth first search algorithm.** |  |  |
| **3** | **Write a program to simulate 4-Queen / N-Queen problem.** |  |  |
| **4** | **Write a program to solve tower of Hanoi problem.** |  |  |
| **5** | **Write a program to implement mini-max algorithm.** |  |  |
| **6** | **Write a program to solve water jug problem.** |  |  |
| **7** | **Design the simulation of tic – tac – toe game using alpha-beta algorithm.** |  |  |
| **8** | **Design an application to simulate number puzzle problem.** |  |  |
| **9** | **Solve constraint satisfaction problem.** |  |  |
| **10** | **Write a program to derive the predicate. (fore.g.: Sachin is batsman, batsman is cricketer) - >Sachin is Cricketer** |  |  |
| **11** | **Write a program which contains three predicates: male, female, parent. Make rules for following family relations: father, mother, grandfather,grandmother,brother, sister, uncle, aunt, nephew and niece, cousin. Question: i). Draw Family Tree. ii. Define: Clauses, Facts, Predicates and Rules with conjunction and disjunction.** |  |  |

**PRACTICAL 1:**

**Write a program to implement depth first search algorithm.**

**CODE:**

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

'C':['A','D','E'],'D':['A','C'],'E':['B','C']}

visited=set()

def dfs (visited,graph,root):

if root not in visited:

print(root)

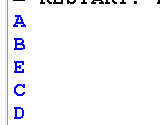
visited.add(root)

for neighbour in graph[root]:

dfs(visited,graph,neighbour)

dfs(visited,graph,'A')

**OUTPUT:**



**PRACTICAL 2:**

**Write a program to implement breadth first search algorithm.**

**CODE:**

graph={

'A':['B','S'],

'B':['A'],

'C':['S','F','E','D'],

'D':['C'],

'E':['C','H'],

'F':['C','H'],

'G':['S','F','H'],

'H':['E','G'],

'S':['A','G','C']

}

visited=[]

queue=[]

def bfs(visited,graph,node):

visited.append(node)

queue.append(node)

while queue:

m=queue.pop(0)

print(m,end=" ")

for neighbour in graph[m]:

if neighbour not in visited:

visited.append(neighbour)

queue.append(neighbour)

bfs(visited,graph,'A')

**OUTPUT:**



**PRACTICAL 3:**

**Write a program to simulate 4-Queen / N-Queen problem.**

**CODE:**

n=int(input("Enter the value"))

board=[[ 0 for i in range(n)]for i in range(n)]

def check\_col (board,row,col):

for i in range (row,-1,-1):

if board[i][col]==1:

return False

return True

def check\_digonal(board,row,col):

for i,j in zip (range(row,-1,-1),range(col,-1,-1)):

if board [i][j]==1:

return False

for i,j in zip (range(row,-1,-1),range(col,n)):

if board[i][j]==1:

return False

return True

def nqn(board,row):

if row==n:

return True

for i in range(n):

if (check\_col(board,row,i)==True and check\_digonal(board,row,i)==True):

board[row][i]=1

if nqn(board,row+1):

return True

board[row][i]=0

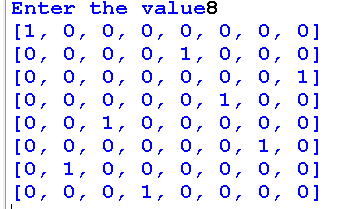
return False

nqn(board,0)

for row in board:

print (row)

**OUTPUT:**



**PRACTICAL 4:**

**Write a program to solve tower of Hanoi problem.**

**CODE:**

#pole and disc ## Tower of Hanoi

def towerOfHanoi(n,from\_rod,to\_rod,aux\_rod):

if n==0:

return

towerOfHanoi(n-1,from\_rod,to\_rod,aux\_rod)

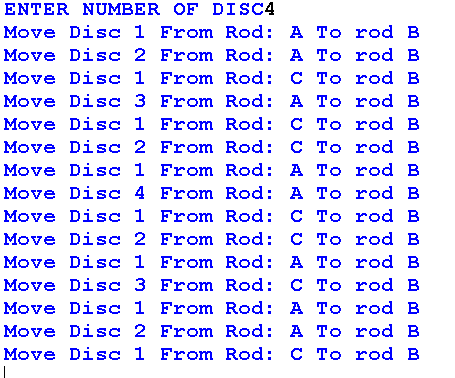
print("Move Disc",n,"From Rod:",from\_rod,"To rod",to\_rod)

towerOfHanoi(n-1,aux\_rod,to\_rod,from\_rod,)

n=int(input("ENTER NUMBER OF DISC"))

towerOfHanoi(n,'A','B','C')

**OUTPUT:**



**PRACTICAL 5:**

**Write a program to implement mini-max algorithm.**

**CODE:**

MAX,MIN=1000,-1000

def minimax(depth,nodeIndex,maximizingPlayer,values,alpha,beta):

if depth==3:

return values[nodeIndex]

if maximizingPlayer:

best=MIN

for i in range(0,2):

val=minimax(depth+1,nodeIndex\*2+i,False,values,alpha,beta)

best=max(best,val)

alpha=max(alpha,best)

if beta<=alpha:

break

return best

else:

best=MAX

for i in range(0,2):

val=minimax(depth+1,nodeIndex\*2+i,False,values,alpha,beta)

best=min(best,val)

beta=min(beta,best)

if beta<=alpha:

break

return best

if \_\_name\_\_=="\_\_main\_\_":

values=[3,5,6,9,1,2,0,-1,3,4]

print("The Optimal Values is:",minimax(0,0,True,values,MIN,MAX))

**OUTPUT:**



**PRACTICAL 6:**

**Write a program to solve water jug problem.**

**CODE:**

from collections import deque

def Solution(a, b, target):

m = {}

isSolvable = False

path = []

q = deque()

#Initializing with jugs being empty

q.append((0, 0))

while (len(q) > 0):

# Current state

u = q.popleft()

if ((u[0], u[1]) in m):

continue

if ((u[0] > a or u[1] > b or

u[0] < 0 or u[1] < 0)):

continue

path.append([u[0], u[1]])

m[(u[0], u[1])] = 1

if (u[0] == target or u[1] == target):

isSolvable = True

if (u[0] == target):

if (u[1] != 0):

path.append([u[0], 0])

else:

if (u[0] != 0):

path.append([0, u[1]])

sz = len(path)

for i in range(sz):

print("(", path[i][0], ",",

path[i][1], ")")

break

q.append([u[0], b]) # Fill Jug2

q.append([a, u[1]]) # Fill Jug1

for ap in range(max(a, b) + 1):

c = u[0] + ap

d = u[1] - ap

if (c == a or (d == 0 and d >= 0)):

q.append([c, d])

c = u[0] - ap

d = u[1] + ap

if ((c == 0 and c >= 0) or d == b):

q.append([c, d])

q.append([a, 0])

q.append([0, b])

if (not isSolvable):

print("Solution not possible")

if \_\_name\_\_ == '\_\_main\_\_':

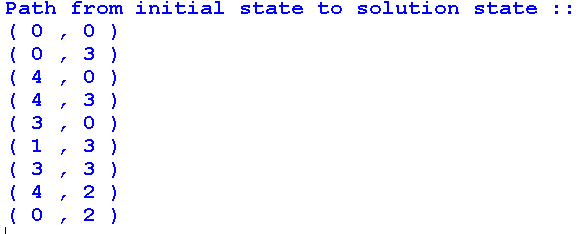
Jug1, Jug2, target = 4, 3, 2

print("Path from initial state "

"to solution state ::")

Solution(Jug1, Jug2, target)

**OUTPUT:**



**7). Design the simulation of tic – tac – toe game using alpha-beta algorithm.**

**CODE:**

from random import choice

from math import inf

board = [[0, 0, 0],

[0, 0, 0],

[0, 0, 0]]

def Gameboard(board):

chars = {1: 'X', -1: 'O', 0: ' '}

for x in board:

for y in x:

ch = chars[y]

print(f'| {ch} |', end='')

print('\n' + '---------------')

print('===============')

def Clearboard(board):

for x, row in enumerate(board):

for y, col in enumerate(row):

board[x][y] = 0

def winningPlayer(board, player):

conditions = [[board[0][0], board[0][1], board[0][2]],

[board[1][0], board[1][1], board[1][2]],

[board[2][0], board[2][1], board[2][2]],

[board[0][0], board[1][0], board[2][0]],

[board[0][1], board[1][1], board[2][1]],

[board[0][2], board[1][2], board[2][2]],

[board[0][0], board[1][1], board[2][2]],

[board[0][2], board[1][1], board[2][0]]]

if [player, player, player] in conditions:

return True

return False

def gameWon(board):

return winningPlayer(board, 1) or winningPlayer(board, -1)

def printResult(board):

if winningPlayer(board, 1):

print('X has won! ' + '\n')

elif winningPlayer(board, -1):

print('O\'s have won! ' + '\n')

else:

print('Draw' + '\n')

def blanks(board):

blank = []

for x, row in enumerate(board):

for y, col in enumerate(row):

if board[x][y] == 0:

blank.append([x, y])

return blank

def boardFull(board):

if len(blanks(board)) == 0:

return True

return False

def setMove(board, x, y, player):

board[x][y] = player

def playerMove(board):

e = True

moves = {1: [0, 0], 2: [0, 1], 3: [0, 2],

4: [1, 0], 5: [1, 1], 6: [1, 2],

7: [2, 0], 8: [2, 1], 9: [2, 2]}

while e:

try:

move = int(input('Enter a number between 1-9: '))

if move < 1 or move > 9:

print('Invalid Move! Try again!')

elif not (moves[move] in blanks(board)):

print('Invalid Move! Try again!')

else:

setMove(board, moves[move][0], moves[move][1], 1)

Gameboard(board)

e = False

except(KeyError, ValueError):

print('Enter a number!')

def getScore(board):

if winningPlayer(board, 1):

return 10

elif winningPlayer(board, -1):

return -10

else:

return 0

def abminimax(board, depth, alpha, beta, player):

row = -1

col = -1

if depth == 0 or gameWon(board):

return [row, col, getScore(board)]

else:

for cell in blanks(board):

setMove(board, cell[0], cell[1], player)

score = abminimax(board, depth - 1, alpha, beta, -player)

if player == 1:

# X is always the max player

if score[2] > alpha:

alpha = score[2]

row = cell[0]

col = cell[1]

else:

if score[2] < beta:

beta = score[2]

row = cell[0]

col = cell[1]

setMove(board, cell[0], cell[1], 0)

if alpha >= beta:

break

if player == 1:

return [row, col, alpha]

else:

return [row, col, beta]

def o\_comp(board):

if len(blanks(board)) == 9:

x = choice([0, 1, 2])

y = choice([0, 1, 2])

setMove(board, x, y, -1)

Gameboard(board)

else:

result = abminimax(board, len(blanks(board)), -inf, inf, -1)

setMove(board, result[0], result[1], -1)

Gameboard(board)

def x\_comp(board):

if len(blanks(board)) == 9:

x = choice([0, 1, 2])

y = choice([0, 1, 2])

setMove(board, x, y, 1)

Gameboard(board)

else:

result = abminimax(board, len(blanks(board)), -inf, inf, 1)

setMove(board, result[0], result[1], 1)

Gameboard(board)

def makeMove(board, player, mode):

if mode == 1:

if player == 1:

playerMove(board)

else:

o\_comp(board)

else:

if player == 1:

o\_comp(board)

else:

x\_comp(board)

def pvc():

while True:

try:

order = int(input('Enter to play 1st or 2nd: '))

if not (order == 1 or order == 2):

print('Please pick 1 or 2')

else:

break

except(KeyError, ValueError):

print('Enter a number')

Clearboard(board)

if order == 2:

currentPlayer = -1

else:

currentPlayer = 1

while not (boardFull(board) or gameWon(board)):

makeMove(board, currentPlayer, 1)

currentPlayer \*= -1

printResult(board)

# Driver Code

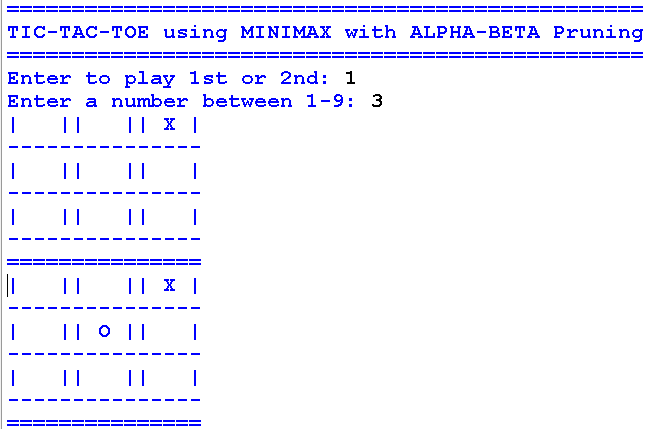
print("=================================================")

print("TIC-TAC-TOE using MINIMAX with ALPHA-BETA Pruning")

print("=================================================")

pvc()

**OUTPUT:**



**OR**



**PRACTICAL 8:**

**Design an application to simulate number puzzle problem.**

**CODE:**

class Node:

def \_\_init\_\_(self,data,level,fval):

""" Initialize the node with the data, level of the node and the calculated fvalue """

self.data = data

self.level = level

self.fval = fval

def generate\_child(self):

""" Generate child nodes from the given node by moving the blank space

either in the four directions {up,down,left,right} """

x,y = self.find(self.data,'\_')

""" val\_list contains position values for moving the blank space in either of

the 4 directions [up,down,left,right] respectively. """

val\_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]

children = []

for i in val\_list:

child = self.shuffle(self.data,x,y,i[0],i[1])

if child is not None:

child\_node = Node(child,self.level+1,0)

children.append(child\_node)

return children

def shuffle(self,puz,x1,y1,x2,y2):

""" Move the blank space in the given direction and if the position value are out

of limits the return None """

if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):

temp\_puz = []

temp\_puz = self.copy(puz)

temp = temp\_puz[x2][y2]

temp\_puz[x2][y2] = temp\_puz[x1][y1]

temp\_puz[x1][y1] = temp

return temp\_puz

else:

return None

def copy(self,root):

""" Copy function to create a similar matrix of the given node"""

temp = []

for i in root:

t = []

for j in i:

t.append(j)

temp.append(t)

return temp

def find(self,puz,x):

""" Specifically used to find the position of the blank space """

for i in range(0,len(self.data)):

for j in range(0,len(self.data)):

if puz[i][j] == x:

return i,j

class Puzzle:

def \_\_init\_\_(self,size):

""" Initialize the puzzle size by the specified size,open and closed lists to empty """

self.n = size

self.open = []

self.closed = []

def accept(self):

""" Accepts the puzzle from the user """

puz = []

for i in range(0,self.n):

temp = input().split(" ")

puz.append(temp)

return puz

def f(self,start,goal):

""" Heuristic Function to calculate hueristic value f(x) = h(x) + g(x) """

return self.h(start.data,goal)+start.level

def h(self,start,goal):

""" Calculates the different between the given puzzles """

temp = 0

for i in range(0,self.n):

for j in range(0,self.n):

if start[i][j] != goal[i][j] and start[i][j] != '\_':

temp += 1

return temp

def process(self):

""" Accept Start and Goal Puzzle state"""

print("Enter the start state matrix \n")

start = self.accept()

print("Enter the goal state matrix \n")

goal = self.accept()

start = Node(start,0,0)

start.fval = self.f(start,goal)

""" Put the start node in the open list"""

self.open.append(start)

print("\n\n")

while True:

cur = self.open[0]

print("")

print(" | ")

print(" | ")

print(" \\\'/ \n")

for i in cur.data:

for j in i:

print(j,end=" ")

print("")

""" If the difference between current and goal node is 0 we have reached the goal node"""

if(self.h(cur.data,goal) == 0):

break

for i in cur.generate\_child():

i.fval = self.f(i,goal)

self.open.append(i)

self.closed.append(cur)

del self.open[0]

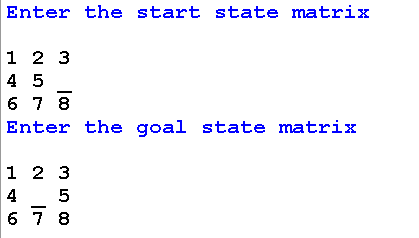
""" sort the opne list based on f value """

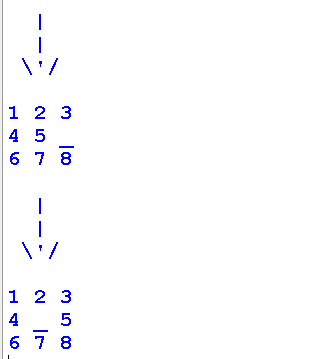
self.open.sort(key = lambda x:x.fval,reverse=False)

puz = Puzzle(3)

puz.process()

**OUTPUT:**





**PRACTICAL 9:**

**Solve constraint satisfaction problem.**

**CODE:**

def solve\_sudoku(board):

# Find the first empty cell in the board

empty\_cell = find\_empty\_cell(board)

# If there are no empty cells, the puzzle is solved

if not empty\_cell:

return True

row, col = empty\_cell

# Try filling in a digit from 1 to 9

for num in range(1, 10):

if is\_valid\_move(board, row, col, num):

# If the move is valid, set the cell to the chosen number

board[row][col] = num

# Recursively try to solve the rest of the puzzle

if solve\_sudoku(board):

return True

# If the puzzle cannot be solved with this choice, backtrack

board[row][col] = 0

# If no valid number can be placed, backtrack to the previous cell

return False

def find\_empty\_cell(board):

# Find the first empty cell in the board

for row in range(9):

for col in range(9):

if board[row][col] == 0:

return (row, col)

return None

def is\_valid\_move(board, row, col, num):

# Check if the chosen number is valid for the given cell

return (

not used\_in\_row(board, row, num) and

not used\_in\_col(board, col, num) and

not used\_in\_box(board, row - row % 3, col - col % 3, num)

)

def used\_in\_row(board, row, num):

# Check if the number is used in the same row

return num in board[row]

def used\_in\_col(board, col, num):

# Check if the number is used in the same column

return num in [board[i][col] for i in range(9)]

def used\_in\_box(board, box\_start\_row, box\_start\_col, num):

# Check if the number is used in the 3x3 box

for i in range(3):

for j in range(3):

if board[i + box\_start\_row][j + box\_start\_col] == num:

return True

return False

# Example Sudoku board

sudoku\_board = [

[0, 0, 3, 0, 2, 0, 6, 0, 0],

[9, 0, 0, 3, 0, 5, 0, 0, 1],

[0, 0, 1, 8, 0, 6, 4, 0, 0],

[0, 0, 8, 1, 0, 2, 9, 0, 0],

[7, 0, 0, 0, 0, 0, 0, 0, 8],

[0, 0, 6, 7, 0, 8, 2, 0, 0],

[0, 0, 2, 6, 0, 9, 5, 0, 0],

[8, 0, 0, 2, 0, 3, 0, 0, 9],

[0, 0, 5, 0, 1, 0, 3, 0, 0]

]

if solve\_sudoku(sudoku\_board):

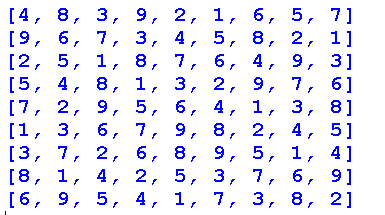
for row in sudoku\_board:

print(row)

else:

print("No solution exists.")

**OUTPUT:**



**PRACTICAL 10:**

**Write a program to derive the predicate. (for e.g.: Sachin is batsman, batsman is cricketer) - >Sachin is Cricketer**

**CODE:**

class PredicateDerivation:

def \_\_init\_\_(self):

self.predicates = {} # Dictionary to store predicates

def add\_predicate(self, subject, predicate):

"""

Add a predicate to the dictionary

"""

if subject not in self.predicates:

self.predicates[subject] = set() # Initialize set if subject not found

self.predicates[subject].add(predicate)

def derive\_predicate(self, subject):

"""

Derive predicate using transitive inference

"""

if subject not in self.predicates:

return None # Subject not found

derived\_predicates = set() # Initialize set for derived predicates

stack = list(self.predicates[subject]) # Stack for DFS traversal

while stack:

current\_predicate = stack.pop()

derived\_predicates.add(current\_predicate)

if current\_predicate in self.predicates:

# Add predicates related to current\_predicate to the stack

stack.extend(self.predicates[current\_predicate])

return derived\_predicates

# Example usage:

if \_\_name\_\_== "\_\_main\_\_":

# Create an instance of PredicateDerivation

predicate\_derivation = PredicateDerivation()

# Add predicates

predicate\_derivation.add\_predicate("Sachin", "batsman")

predicate\_derivation.add\_predicate("batsman", "cricketer")

# Derive predicate for Sachin

derived\_predicates = predicate\_derivation.derive\_predicate("Sachin")

# Print derived predicates

print("Derived predicates for Sachin:", derived\_predicates)

**OUTPUT:**



**PRACTICAL 11:**

**Write a program which contains three predicates: male, female, parent. Make rules for following family relations: father, mother, grandfather, grandmother, brother, sister, uncle, aunt, nephew and niece, cousin. Question:**

1. **Draw Family Tree. ii. Define: Clauses, Facts, Predicates and Rules with conjunction and disjunction.**

**CODE:**

# Define predicates

males = ['Rahul', 'Amit', 'Ravi', 'Suresh', 'Vikram', 'Rajesh', 'Anil'] # List of males

females = ['Meera', 'Priya', 'Neha', 'Sita', 'Deepa', 'Rani', 'Geeta'] # List of females

# Define parent relationships as facts (clauses)

parents = [('Rahul', 'Meera'), ('Rahul', 'Priya'), ('Meera', 'Neha'), ('Meera', 'Ravi'),

('Amit', 'Priya'), ('Amit', 'Neha'), ('Priya', 'Sita'), ('Priya', 'Suresh'),

('Suresh', 'Vikram'), ('Suresh', 'Neha'), ('Ravi', 'Deepa'), ('Sita', 'Anil')]

# Define rules for family relationships using predicates and facts

# Rules for father, mother, grandfather, grandmother, brother, sister, uncle, aunt, nephew, niece, cousin

def father(x, y):

return (x in males) and ((x, y) in parents)

def mother(x, y):

return (x in females) and ((x, y) in parents)

def grandfather(x, y):

for z in males:

if father(x, z) and father(z, y):

return True

elif father(x, z) and mother(z, y):

return True

return False

def grandmother(x, y):

for z in females:

if mother(x, z) and father(z, y):

return True

elif mother(x, z) and mother(z, y):

return True

return False

def brother(x, y):

for z in males:

if (father(z, x) and father(z, y)) or (mother(z, x) and mother(z, y)):

return True

return False

def sister(x, y):

for z in females:

if (father(z, x) and father(z, y)) or (mother(z, x) and mother(z, y)):

return True

return False

def uncle(x, y):

return (brother(x, z) and (father(z, y) or mother(z, y))) or (husband(x, z) and sister(z, w) and (father(w, y) or mother(w, y)))

def aunt(x, y):

return (sister(x, z) and (father(z, y) or mother(z, y))) or (wife(x, z) and brother(z, w) and (father(w, y) or mother(w, y)))

def nephew(x, y):

return (male(y) and (uncle(z, y) or aunt(z, y)))

def niece(x, y):

return (female(y) and (uncle(z, y) or aunt(z, y)))

def cousin(x, y):

return (parent(z, x) and parent(w, y) and (brother(z, w) or sister(z, w)))

# Main code for testing the predicates and rules

if \_\_name\_\_ == "\_\_main\_\_":

# Test the predicates and rules

print(f"Rahul is a father of Neha: {father('Rahul', 'Neha')}")

print(f"Meera is a mother of Ravi: {mother('Meera', 'Ravi')}")

print(f"Suresh is a grandfather of Deepa: {grandfather('Suresh', 'Deepa')}")

print(f"Meera is a grandmother of Anil: {grandmother('Meera', 'Anil')}")

print(f"Ravi is a brother of Sita: {brother('Ravi', 'Sita')}")

print(f"Priya is a sister of Deepa: {sister('Priya', 'Deepa')}")

# print(f"Amit is an uncle of Deepa: {uncle('Amit', 'Deepa')}")

# print(f"Meera is an aunt of Anil: {aunt('Meera', 'Anil')}")

# print(f"Vikram is a nephew of Priya: {nephew('Vikram', 'Priya')}")

# print(f"Sita is a niece of Neha: {niece('Sita', 'Neha')}")

# print(f"Neha and Anil are cousins: {cousin('Neha', 'Anil')}")

**OUTPUT:**

