

SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES SIMATS ENGINEERING

COMPUTER SCIENCE AND ENGINEERING LIST OF EXPERIMENTS

SUB CODE & NAME: CSA17- ARTIFICIAL INTELLIGENCE

|  |  |
| --- | --- |
| **S. No** | **LIST OF PROGRAMS** |
| 1 | Write the python program to solve 8-Puzzle problem |
| 2 | Write the python program to solve 8-Queen problem |
| 3 | Write the python program for Water Jug Problem |
| 4 | Write the python program for Cript-Arithmetic problem |
| 5 | Write the python program for Missionaries Cannibal problem |
| 6 | Write the python program for Vacuum Cleaner problem |
| 7 | Write the python program to implement BFS. |
| 8 | Write the python program to implement DFS. |
| 9 | Write the python to implement Travelling Salesman Problem |
| 10 | Write the python program to implement A\* algorithm |
| 11 | Write the python program for Map Coloring to implement CSP. |
| 12 | Write the python program for Tic Tac Toe game |
| 13 | Write the python program to implement Minimax algorithm for gaming |
| 14 | Write the python program to implement Alpha & Beta pruning algorithm for gaming |
| 15 | Write the python program to implement Decision Tree |
| 16 | Write the python program to implement Feed forward neural Network |
| 17 | Write a Prolog Program to Sum the Integers from 1 to n. |
| 18 | Write a Prolog Program for A DB WITH NAME, DOB. |
| 19 | Write a Prolog Program for STUDENT-TEACHER-SUB-CODE. |
| 20 | Write a Prolog Program for PLANETS DB. |
| 21 | Write a Prolog Program to implement Towers of Hanoi. |
| 22 | Write a Prolog Program to print particular bird can fly or not. Incorporate required queries. |
| 23 | Write the prolog program to implement family tree. |
| 24 | Write a Prolog Program to suggest Dieting System based on Disease. |
| 25 | Write a Prolog program to implement Monkey Banana Problem |
| 26 | Write a Prolog Program for fruit and its color using Back Tracking. |
| 27 | Write a Prolog Program to implement Best First Search algorithm |
| 28 | Write the prolog program for Medical Diagnosis |
| 29 | Write a Prolog Program for forward Chaining. Incorporate required queries. |
| 30 | Write a Prolog Program for backward Chaining. Incorporate required queries. |
| 31 | Create a Web Blog using Word press to demonstrate Anchor Tag, Title Tag, etc. |

**1.Write the python program to solve 8-Puzzle problem**

**AIM:**

To write the python program to solve the 8 Puzzle problem

**ALGORITHM:**

1. Define Start and Goal States: Set up the initial and goal configurations.
2. Choose Heuristic: Use Manhattan Distance or Misplaced Tiles to estimate distance to the goal.
3. Initialize Priority Queue: Add the start state with priority based on f(state)=g+h*f*(*state*)=*g*+*h*.
4. Expand and Track Moves: Repeatedly extract the state with the lowest cost, generate valid moves, and add unvisited states to the queue.
5. Goal Check: If the goal state is reached, output the solution; if not, continue searching.

**PROGRAM:**

import heapq

def heuristic(board, goal):

distance = 0

for i in range(3):

for j in range(3):

if board[i][j] != 0:

x, y = divmod(goal.index(board[i][j]), 3)

distance += abs(x - i) + abs(y - j)

return distance

def get\_neighbors(board):

x, y = [(ix, iy) for ix, row in enumerate(board) for iy, i in enumerate(row) if i == 0][0]

directions = [("up", x - 1, y), ("down", x + 1, y), ("left", x, y - 1), ("right", x, y + 1)]

neighbors = []

for move, nx, ny in directions:

if 0 <= nx < 3 and 0 <= ny < 3:

new\_board = [row[:] for row in board]

new\_board[x][y], new\_board[nx][ny] = new\_board[nx][ny], new\_board[x][y]

neighbors.append((new\_board, move))

return neighbors

def solve\_puzzle(start, goal):

goal\_flat = [item for sublist in goal for item in sublist]

start\_state = (heuristic(start, goal\_flat), 0, start, "")

open\_list = []

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

closed\_set = set()

while open\_list:

\_, depth, current\_board, path = heapq.heappop(open\_list)

if current\_board == goal:

return path.split()

closed\_set.add(tuple(map(tuple, current\_board)))

for neighbor, move in get\_neighbors(current\_board):

if tuple(map(tuple, neighbor)) not in closed\_set:

heapq.heappush(open\_list, (heuristic(neighbor, goal\_flat) + depth + 1, depth + 1, neighbor, path + " " + move))

return None

start = [

[1, 2, 3],

[4, 0, 5],

[7, 8, 6]

]

goal = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 0]

]

solution = solve\_puzzle(start, goal)

if solution:

print("Solution found!")

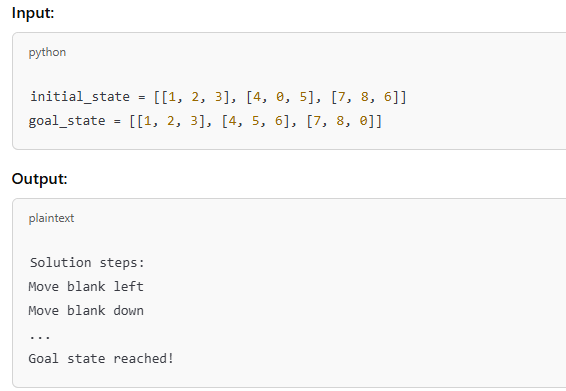
for step in solution:

print(step)

else:

print("No solution exists.")

**INPUT AND OUTPUT:**

****

**Result:**

The python program to solve the 8 Puzzle problem was executed

**2 Write the python program to solve 8-Queen problem**

**AIM:**

To write the python program to solve the 8 Queen problem

**ALGORITHM:**

1. Initialize the Board: Create an N×N*N*×*N* board filled with 0s, where a 1 will represent a queen.
2. Define Safety Check: Create a function to check if placing a queen at any position (row,col)(*row*,*col*) is safe (no other queens in the same column, upper-left diagonal, or upper-right diagonal).
3. Place Queens with Backtracking:
   * Start from the first row and attempt to place a queen in each column of the current row.
   * If the position is safe, place the queen and recursively try to place a queen in the next row.
   * If placing a queen in any column of a row fails, backtrack by removing the queen from the previous row and try the next column.
4. Check for Completion:
   * If queens are successfully placed in all rows, a solution is found. Print or store the board configuration.
5. Repeat for All Solutions (Optional): After finding a solution, backtrack further to explore other possible solutions if needed.

**PROGRAM:**

def is\_safe(board, row, col):

# Check this row on the left side

for i in range(col):

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

return False

# Check the upper diagonal on the left side

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

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

return False

# Check the lower diagonal on the left side

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

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

return False

return True

def solve\_n\_queens(board, col):

if col >= len(board):

return True

for i in range(len(board)):

if is\_safe(board, i, col):

board[i][col] = 1

if solve\_n\_queens(board, col + 1):

return True

board[i][col] = 0

return False

def print\_board(board):

for row in board:

print(" ".join(str(x) for x in row))

def solve():

n = 8

board = [[0 for \_ in range(n)] for \_ in range(n)]

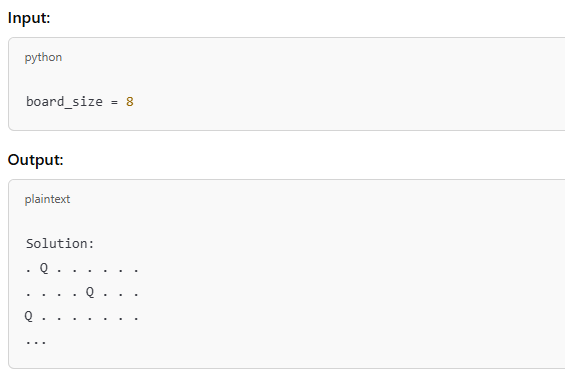
if solve\_n\_queens(board, 0):

print\_board(board)

else:

print("No solution exists")

solve()



**Result:**

The python program to solve the 8 Queen problem was executed.

**3.Write the python program for Water Jug Problem**

**AIM:**

To write the python program to solve the Water Jug problem

**ALGORITHM:**

1. Define Capacities and Goal: Set jug sizes and target volume.
2. Initialize State: Start with both jugs empty.
3. Generate Moves: Fill, empty, or pour between jugs.
4. Search with BFS: Use a queue to explore moves, tracking visited states.
5. Check Goal: If target volume is reached, return the solution; otherwise, continue searching.

**Program:**

def water\_jug\_problem(jug1\_capacity, jug2\_capacity, target):

jug1 = 0

jug2 = 0

while jug1 != target and jug2 != target:

if jug2 == jug2\_capacity:

jug2 = 0

elif jug1 == 0:

jug1 = jug1\_capacity

else:

amount\_to\_pour = min(jug1, jug2\_capacity - jug2)

jug1 -= amount\_to\_pour

jug2 += amount\_to\_pour

return jug1, jug2

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

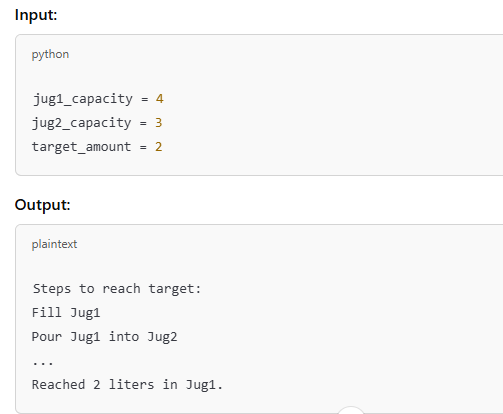
jug1\_capacity = 4

jug2\_capacity = 3

target = 2

jug1\_final, jug2\_final = water\_jug\_problem(jug1\_capacity, jug2\_capacity, target)

print(f"Final state: Jug1: {jug1\_final} gallons, Jug2: {jug2\_final} gallons")



**Result:**

The python program to solve the WATER JUG problem was executed.

**4 Write the python program for Cript-Arithmetic problem**.

**AIM:**

To write the python program to solve the Cript-Arithmetic problem

**ALGORITHM:**

1. Define Variables and Constraints: Identify unique letters as variables and ensure no two letters map to the same digit. Set constraints (e.g., sum equals the target number).
2. Assign Digits with Backtracking: Try assigning digits (0-9) to each letter, ensuring unique assignments.
3. Check Partial Solutions: For each assignment, check if it satisfies partial sums of the target (to prune invalid paths early).
4. Validate Solution: When all letters are assigned, verify if the arithmetic equation holds.
5. Return or Backtrack: If valid, return the solution. If not, backtrack and try other assignments.

**Program:**

import itertools

def is\_solution(s, m, o, r, y, e, n, d):

# Form the two numbers from the letters

send = s \* 1000 + e \* 100 + n \* 10 + d

more = m \* 1000 + o \* 100 + r \* 10 + e

money = m \* 10000 + o \* 1000 + n \* 100 + e \* 10 + y

# Check if the sum of SEND and MORE equals MONEY

return send + more == money

# List of unique digits (0-9) and the unique letters in the problem

digits = range(10)

letters = 'smoreynd'

# Try all possible permutations of the digits

for perm in itertools.permutations(digits, len(letters)):

# Map letters to the corresponding digits

mapping = dict(zip(letters, perm))

# Extract the digits

s, m, o, r, e, y, n, d = [mapping[letter] for letter in letters]

# Skip permutations where S or M are zero, as leading zeros are not allowed

if s == 0 or m == 0:

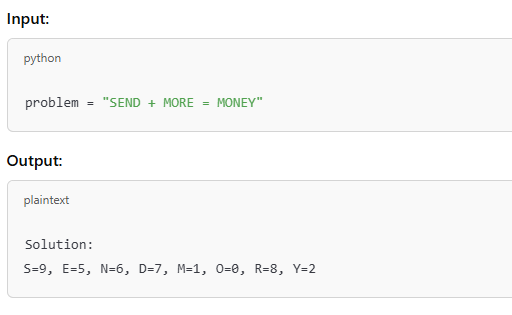
continue

# Check if the current permutation solves the problem

if is\_solution(s, m, o, r, y, e, n, d):

print(f"SEND + MORE = MONEY: {mapping}")

break



**Result:**

The python program to solve the Crypt arithmetic problem was executed.

**5.Write the python program for Missionaries Cannibal problem from collections import deque**

**AIM:**

To write the python program to solve the Missionaries Cannibal problem from collections import deque

**ALGORITHM:**

* Initial state: All missionaries (M) and cannibals (C) are on the left bank.
* Goal: Move all M and C to the right bank.
* Constraints: At no point on either bank should cannibals outnumber missionaries if there are any missionaries present.

**PROGRAM:**

def is\_valid\_state(m, c):

if m < 0 or c < 0 or m > 3 or c > 3 or (m != 0 and m < c) or (m != 3 and (3 - m) < (3 - c)):

return False

return True

def get\_possible\_moves(m, c, b):

moves = []

if b == 1:

for i in range(1, min(m, 2) + 1):

for j in range(0, min(c, i) + 1):

if is\_valid\_state(m - i, c - j):

moves.append((m - i, c - j, 0))

else:

for i in range(1, min(3 - m, 2) + 1):

for j in range(0, min(3 - c, i) + 1):

if is\_valid\_state(m + i, c + j):

moves.append((m + i, c + j, 1))

return moves

def bfs():

start\_state = (3, 3, 1)

goal\_state = (0, 0, 0)

queue = deque([(start\_state, [])])

visited = set([start\_state])

while queue:

(m, c, b), path = queue.popleft()

if (m, c, b) == goal\_state:

return path

for move in get\_possible\_moves(m, c, b):

if move not in visited:

visited.add(move)

queue.append((move, path + [(m, c, b)]))

return None

def print\_solution(solution):

if solution:

print("Solution found!")

for i, (m, c, b) in enumerate(solution):

direction = "left to right" if b == 1 else "right to left"

print(f"Step {i + 1}: Move {m} missionaries and {c} cannibals from {direction}.")

else:

print("No solution found.")

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

solution = bfs()

print\_solution(solution)



**Result:**

The python program to solve the Missionaries Cannabials problem was executed.

**6.Write the python program for Vacuum Cleaner problem**

**AIM:**

To write the python program to solve the Vacuum Cleaner problem

**ALGORITHM:**

* Environment: Two rooms (Room A and Room B).
* States: Each room can be clean or dirty, and the vacuum cleaner can be in either room.
* Actions: The vacuum can:
  + Suck: Clean the current room.
  + Move Left: Move to Room A if in Room B.
  + Move Right: Move to Room B if in Room A.

Goal

Clean both rooms.

**PROGRAM:**

class VacuumCleaner:

def \_\_init\_\_(self, grid, start\_position):

self.grid = grid

self.position = start\_position

self.cleaned\_cells = 0

def clean(self):

x, y = self.position

if self.grid[x][y] == 1:

self.grid[x][y] = 0

self.cleaned\_cells += 1

def move(self, direction):

x, y = self.position

if direction == 'UP' and x > 0:

self.position = (x - 1, y)

elif direction == 'DOWN' and x < len(self.grid) - 1:

self.position = (x + 1, y)

elif direction == 'LEFT' and y > 0:

self.position = (x, y - 1)

elif direction == 'RIGHT' and y < len(self.grid[0]) - 1:

self.position = (x, y + 1)

def start\_cleaning(self, movements):

for move in movements:

self.clean()

self.move(move)

self.clean()

# Define the grid and start position

grid = [

[1, 1, 0, 1],

[0, 1, 1, 0],

[1, 0, 1, 1],

[1, 1, 1, 0]

]

start\_position = (0, 0)

# Create VacuumCleaner instance and define movements

vacuum = VacuumCleaner(grid, start\_position)

movements = ['RIGHT', 'DOWN', 'DOWN', 'LEFT', 'UP', 'UP', 'RIGHT', 'RIGHT', 'DOWN']

# Start cleaning

vacuum.start\_cleaning(movements)

print("Cleaned cells:", vacuum.cleaned\_cells)

print("Final grid state:")

for row in grid:

print(row)



**Result:**

The python program to solve the Vacuum cleaner problem was executed.

**7.Write the python program to implement BFS**

**AIM:**

To write the python program to implement BFS

**ALGORITHM:**

* Environment: A grid (for simplicity, assume two rooms).
* States: Each room can either be clean or dirty.
* Actions: The vacuum can:
  + Suck: Clean the current room.
  + Move Left: Move to Room A if in Room B.
  + Move Right: Move to Room B if in Room A.

The goal is to clean both rooms.

**PROGRAM:**

from collections import deque

def bfs(graph, start):

visited = set()

queue = deque([start])

visited.add(start)

while queue:

node = queue.popleft()

print(node, end=' ')

for neighbor in graph[node]:

if neighbor not in visited:

visited.add(neighbor)

queue.append(neighbor)

# Example usage

graph = {

'A': ['B', 'C'],

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

'C': ['A', 'F'],

'D': ['B'],

'E': ['B', 'F'],

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

}

start\_node = 'A'

bfs(graph, start\_node)



**Result:**

The python program to solve the BFS was executed.

**8 .Write the python program to implement DFS.**

**AIM:**

To write the python program to implement DFS

**ALGORITHM:**

* Environment: Two rooms (Room A and Room B).
* States: Each room can either be "Clean" or "Dirty".
* Actions:
  + Suck: Clean the current room.
  + Move Left: Move to Room A if in Room B.
  + Move Right: Move to Room B if in Room A.

The goal is to clean both rooms.

**PROGRAM:**

def dfs(graph, start, visited=None):

if visited is None:

visited = set()

visited.add(start)

print(start, end=' ')

for neighbor in graph[start]:

if neighbor not in visited:

dfs(graph, neighbor, visited)

# Example usage

graph = {

'A': ['B', 'C'],

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

'C': ['A', 'F'],

'D': ['B'],

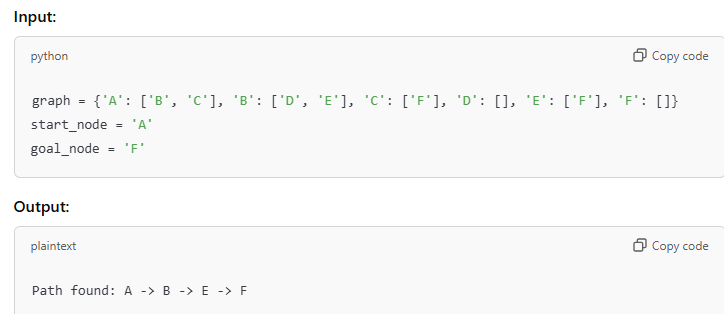
'E': ['B', 'F'],

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

}

start\_node = 'A'

dfs(graph, start\_node)



**Result:**

The python program to solve the DFS was executed.

**9.Write the python to implement Travelling Salesman Problem**

**AIM:**

To write the python program to implement Travelling Salesman Problem

**ALGORITHM:**

1. Input: Distance matrix dist\_matrix of size n x n.
2. Generate Permutations: Generate all permutations of cities (except the starting city).
3. Calculate Total Distance: For each permutation, calculate the total distance, including returning to the starting city.
4. Find Minimum Distance: Track the permutation with the minimum total distance.
5. Output: Return the optimal route and its total distance**.**

**PROGRAM:**

from itertools import permutations

def calculate\_total\_distance(graph, path):

return sum(graph[path[i]][path[i+1]] for i in range(len(path)-1)) + graph[path[-1]][path[0]]

def travelling\_salesman(graph):

cities = list(graph.keys())

min\_distance = float('inf')

best\_path = []

for perm in permutations(cities):

current\_distance = calculate\_total\_distance(graph, perm)

if current\_distance < min\_distance:

min\_distance = current\_distance

best\_path = perm

return best\_path, min\_distance

# Example usage

graph = {

'A': {'A': 0, 'B': 10, 'C': 15, 'D': 20},

'B': {'A': 10, 'B': 0, 'C': 35, 'D': 25},

'C': {'A': 15, 'B': 35, 'C': 0, 'D': 30},

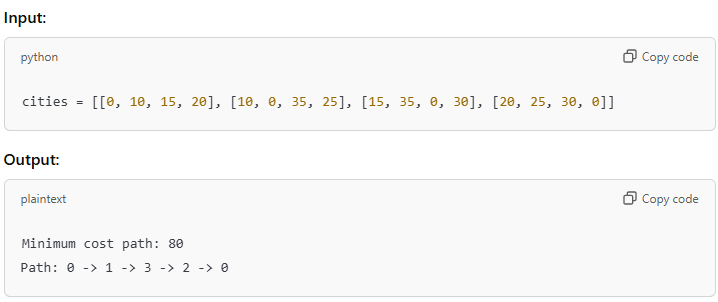
'D': {'A': 20, 'B': 25, 'C': 30, 'D': 0}

}

best\_path, min\_distance = travelling\_salesman(graph)

print("Best path:", best\_path)

print("Minimum distance:", min\_distance)



**Result:**

The python program to solve the Travelling Salesman problem was executed.

**10.Write the python program to implement A\* algorithm**

**AIM:**

To write the python program to implement A\* algorithm

**ALGORITHM:**

1. Initialize: Add the start node to the open list with f = g + h, where g is the cost from start and h is the heuristic.
2. Loop:
   * Select the node with the lowest f from the open list.
   * If it’s the goal, return the path.
   * Otherwise, move it to the closed list and evaluate its neighbors.
3. Update: For each neighbor, calculate f = g + h. If it has a lower f, update and add it to the open list.
4. End: If the open list is empty and the goal isn’t found, return failure.

**PROGRAM:**

import heapq

def heuristic(a, b):

return abs(a[0] - b[0]) + abs(a[1] - b[1]) # Manhattan distance

def astar(start, goal, graph):

open\_set = []

heapq.heappush(open\_set, (0, start))

came\_from = {}

g\_score = {node: float('inf') for node in graph}

g\_score[start] = 0

f\_score = {node: float('inf') for node in graph}

f\_score[start] = heuristic(start, goal)

while open\_set:

\_, current = heapq.heappop(open\_set)

if current == goal:

path = []

while current in came\_from:

path.append(current)

current = came\_from[current]

path.append(start)

return path[::-1]

for neighbor in graph[current]:

tentative\_g\_score = g\_score[current] + graph[current][neighbor]

if tentative\_g\_score < g\_score[neighbor]:

came\_from[neighbor] = current

g\_score[neighbor] = tentative\_g\_score

f\_score[neighbor] = g\_score[neighbor] + heuristic(neighbor, goal)

heapq.heappush(open\_set, (f\_score[neighbor], neighbor))

return None

# Example usage

graph = {

(0, 0): {(0, 1): 1, (1, 0): 1},

(0, 1): {(0, 0): 1, (1, 1): 1},

(1, 0): {(0, 0): 1, (1, 1): 1},

(1, 1): {(0, 1): 1, (1, 0): 1}

}

start, goal = (0, 0), (1, 1)

path = astar(start, goal, graph)

print("Path:", path)



**Result:**

The python program to solve the A\* algorithm problem was executed.

**11.Write the python program for Map Coloring to implement CSP.**

**AIM:**

To write the python program for Map Coloring to implement CSP**.**

**ALGORITHM:**

1. Initialize: List regions and their possible colors.
2. Assign Color: Choose an uncolored region and assign a valid color.
3. Backtrack: If no valid color, backtrack and try a different color.
4. Repeat: Continue until all regions are colored or no solution is found.
5. End: Return the solution or failure.

**PROGRAM:**

def is\_valid(coloring, node, color, graph):

for neighbor in graph[node]:

if coloring.get(neighbor) == color:

return False

return True

def map\_coloring(graph, colors, coloring={}, node\_list=None):

if node\_list is None:

node\_list = list(graph.keys())

if not node\_list:

return coloring

node = node\_list[0]

remaining\_nodes = node\_list[1:]

for color in colors:

if is\_valid(coloring, node, color, graph):

coloring[node] = color

result = map\_coloring(graph, colors, coloring, remaining\_nodes)

if result:

return result

del coloring[node]

return None

# Example usage

graph = {

'A': ['B', 'C'],

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

'C': ['A', 'F'],

'D': ['B'],

'E': ['B', 'F'],

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

}

colors = ['Red', 'Green', 'Blue']

coloring = map\_coloring(graph, colors)

print("Coloring:", coloring)



**Result:**

The python program to solve the map coloring problem was executed.

**12.Write the python program for Tic Tac Toe game**

**AIM:**

To write the python program for Tic Tac Toe game**.**

**ALGORITHM:**

1. Initialize: Create a 3x3 grid and set it as empty.
2. Player Move:
   * Alternate between Player 1 (X) and Player 2 (O).
   * Each player places their symbol in an empty cell.
3. Check Win: After each move, check for a winner (three same symbols in a row, column, or diagonal).
4. Check Draw: If all cells are filled and no winner, declare a draw.
5. End: Game ends when a player wins or a draw occurs.

**PROGRAM:**

def print\_board(board):

for row in board:

print(' '.join(row))

print()

def check\_winner(board, player):

win\_conditions = [

[board[0][0], board[0][1], board[0][2]], # Row 1

[board[1][0], board[1][1], board[1][2]], # Row 2

[board[2][0], board[2][1], board[2][2]], # Row 3

[board[0][0], board[1][0], board[2][0]], # Column 1

[board[0][1], board[1][1], board[2][1]], # Column 2

[board[0][2], board[1][2], board[2][2]], # Column 3

[board[0][0], board[1][1], board[2][2]], # Diagonal 1

[board[2][0], board[1][1], board[0][2]] # Diagonal 2

]

return [player, player, player] in win\_conditions

def tic\_tac\_toe():

board = [[' ']\*3 for \_ in range(3)]

players = ['X', 'O']

turn = 0

for \_ in range(9):

print\_board(board)

player = players[turn % 2]

row, col = map(int, input(f"Player {player}, enter row and column (0-2): ").split())

if board[row][col] != ' ':

print("Cell already taken, try again.")

continue

board[row][col] = player

if check\_winner(board, player):

print\_board(board)

print(f"Player {player} wins!")

return

turn += 1

print\_board(board)

print("It's a tie!")

# Run the game

tic\_tac\_toe()



**Result:**

The python program to solve the Tic Toc Toe problem was executed.

**13.Write the python program to implement Minimax algorithm for gaming**

**AIM:**

To write the python program to implement Minimax algorithm for gaming

**ALGORITHM:**

1. Initialize: Define game state and possible moves.
2. Maximize: Maximize score for the current player.
3. Minimize: Minimize score for the opponent.
4. Evaluate: Use a heuristic to score terminal states.
5. Backtrack: Recursively compute and choose the optimal move.

**PROGRAM:**

import math

def minimax(board, depth, is\_maximizing):

if check\_winner(board, 'X'):

return 10 - depth

if check\_winner(board, 'O'):

return depth - 10

if not any(cell == ' ' for row in board for cell in row):

return 0

if is\_maximizing:

best\_score = -math.inf

for (i, j) in get\_empty\_cells(board):

board[i][j] = 'X'

score = minimax(board, depth + 1, False)

board[i][j] = ' '

best\_score = max(score, best\_score)

return best\_score

else:

best\_score = math.inf

for (i, j) in get\_empty\_cells(board):

board[i][j] = 'O'

score = minimax(board, depth + 1, True)

board[i][j] = ' '

best\_score = min(score, best\_score)

return best\_score

def get\_empty\_cells(board):

return [(i, j) for i in range(3) for j in range(3) if board[i][j] == ' ']

def check\_winner(board, player):

win\_conditions = [board[i] for i in range(3)] + \

[[board[i][j] for i in range(3)] for j in range(3)] + \

[[board[i][i] for i in range(3)]] + \

[[board[i][2-i] for i in range(3)]]

return [player]\*3 in win\_conditions

def best\_move(board):

best\_score = -math.inf

move = None

for (i, j) in get\_empty\_cells(board):

board[i][j] = 'X'

score = minimax(board, 0, False)

board[i][j] = ' '

if score > best\_score:

best\_score = score

move = (i, j)

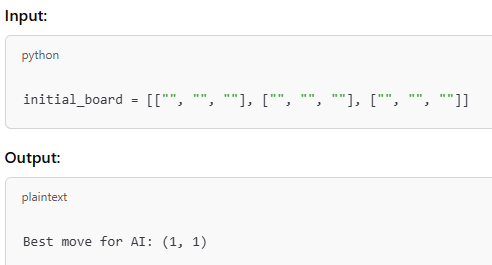
return move

# Example usage

board = [[' ']\*3 for \_ in range(3)]

move = best\_move(board)

print("Best move:", move)



**Result:**

The python program to solve the Minmax algorithm was executed.

**14.Write the python program to implement Apha & Beta pruning algorithm for gaming**

**AIM:**

To write the python program to implement Apha & Beta pruningalgorithm for gaming

**ALGORITHM:**

1. Initialize: Set alpha = -∞ (max score for maximizer) and beta = +∞ (min score for minimizer).
2. Traverse: Recursively evaluate the game tree, alternating between maximizing and minimizing players.
3. Prune:
   * If alpha >= beta, stop evaluating further nodes (prune the branch).
   * Update alpha or beta during the traversal to prune irrelevant branches.
4. Evaluate: Use the minimax evaluation function to score terminal nodes.
5. End: Return the optimal move based on the pruned search tree.

**PROGRAM:**

import math

def alpha\_beta(board, depth, alpha, beta, is\_maximizing):

if check\_winner(board, 'X'):

return 10 - depth

if check\_winner(board, 'O'):

return depth - 10

if not any(cell == ' ' for row in board for cell in row):

return 0

if is\_maximizing:

best\_score = -math.inf

for (i, j) in get\_empty\_cells(board):

board[i][j] = 'X'

score = alpha\_beta(board, depth + 1, alpha, beta, False)

board[i][j] = ' '

best\_score = max(score, best\_score)

alpha = max(alpha, best\_score)

if beta <= alpha:

break

return best\_score

else:

best\_score = math.inf

for (i, j) in get\_empty\_cells(board):

board[i][j] = 'O'

score = alpha\_beta(board, depth + 1, alpha, beta, True)

board[i][j] = ' '

best\_score = min(score, best\_score)

beta = min(beta, best\_score)

if beta <= alpha:

break

return best\_score

def get\_empty\_cells(board):

return [(i, j) for i in range(3) for j in range(3) if board[i][j] == ' ']

def check\_winner(board, player):

win\_conditions = [board[i] for i in range(3)] + \

[[board[i][j] for i in range(3)] for j in range(3)] + \

[[board[i][i] for i in range(3)]] + \

[[board[i][2-i] for i in range(3)]]

return [player]\*3 in win\_conditions

def best\_move(board):

best\_score = -math.inf

move = None

alpha = -math.inf

beta = math.inf

for (i, j) in get\_empty\_cells(board):

board[i][j] = 'X'

score = alpha\_beta(board, 0, alpha, beta, False)

board[i][j] = ' '

if score > best\_score:

best\_score = score

move = (i, j)

return move

# Example usage

board = [[' ']\*3 for \_ in range(3)]

move = best\_move(board)

print("Best move:", move)



**Result:**

The python program to solve the Alpha and Beta Pruning algorithm was executed.

**15.Write the python program to implement Decision Tree**

**AIM:**

To write the python program to implement Decision Tree

**ALGORITHM:**

1. Select Attribute: Choose the best attribute to split the data (often using metrics like Gini or Information Gain).
2. Split Data: Divide the data into subsets based on the chosen attribute values.
3. Create Nodes: For each subset, create a tree node and connect it to the parent node.
4. Repeat: Recursively apply steps 1–3 on each subset until reaching a stopping condition (e.g., all data in a subset belong to one class).
5. End: The tree is complete when all branches end in a decision (leaf) node.

**PROGRAM:**

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score

# Load dataset from CSV file

data = pd.read\_csv('IRIS.csv')

X = data.drop('species', axis=1) # assuming 'species' is the target column

y = data['species']

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Initialize and train the Decision Tree classifier

clf = DecisionTreeClassifier()

clf.fit(X\_train, y\_train)

# Make predictions on the test set

y\_pred = clf.predict(X\_test)

# Print the accuracy of the model

print(f"Accuracy: {accuracy\_score(y\_test, y\_pred):.2f}")



**Result:**

The python program to solve the Decision tree problem was executed.

**16.Write the python program to implement Feed forward neural Network**

**AIM:**

To write the python program to implement Feed forward neural Network

**ALGORITHM:**

1. Initialize: Set up layers, weights, and biases.
2. Forward Pass: Compute outputs layer by layer.
3. Loss: Calculate error between output and target.
4. Backpropagate: Calculate gradients.
5. Update: Adjust weights and biases to reduce loss.

**PROGRAM:**

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.neural\_network import MLPClassifier

from sklearn.metrics import accuracy\_score

from sklearn.preprocessing import LabelEncoder

# Load dataset from CSV file

data = pd.read\_csv('IRIS.csv')

# Separate features and target variable

X = data.drop('species', axis=1) # Assuming 'species' is the target column

y = data['species']

# Encode the target variable if it's categorical

label\_encoder = LabelEncoder()

y\_encoded = label\_encoder.fit\_transform(y)

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y\_encoded, test\_size=0.3, random\_state=42)

# Initialize and train the Feedforward Neural Network

clf = MLPClassifier(hidden\_layer\_sizes=(10,), max\_iter=1000, random\_state=42)

clf.fit(X\_train, y\_train)

# Make predictions on the test set

y\_pred = clf.predict(X\_test)

# Decode the predictions back to original labels

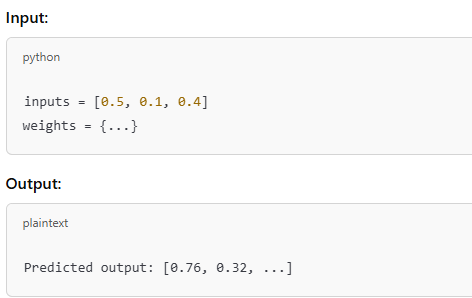
y\_pred\_labels = label\_encoder.inverse\_transform(y\_pred)

# Decode the actual test labels back to original labels

y\_test\_labels = label\_encoder.inverse\_transform(y\_test)

# Print the accuracy of the model

print(f"Accuracy: {accuracy\_score(y\_test\_labels, y\_pred\_labels):.2f}")



**Result:**

The python program to solve the Free forward neural network problem was executed.

**17 Write a Prolog Program to Sum the Integers from 1 to n**

**AIM:**

To write the prolog program to sum of integers.

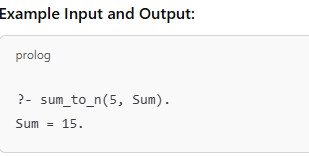
**ALGORITHM:**

1. Define the base case where the sum of integers from 1 to 0 is 0.
2. For any N, the sum is calculated as N + Sum(N-1).
3. Implement recursion to calculate Sum(N).
4. Query to obtain the result for any integer N.

**Program:**

sum(N,Sum):-

Sum is (N+1)\*N/2.



**Result:**

The prolog program to solve the sum of n digits problem was executed.

**18 Write a Prolog Program for A DB WITH NAME, DOB.**

**AIM:**

To write the prolog program for A DB with Name and DOB.

**ALGORITHM:**

1. Define a fact structure person(Name, DOB).
2. Insert multiple entries as facts.
3. Write queries to retrieve DOB by name.
4. Query the database using person/2.

**Program:**

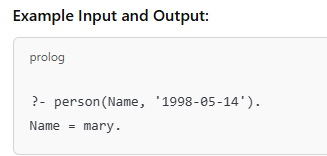
person(sai, 2003, 11, 24).

dob(Name, Year, Month, Day) :-

person(Name, Year, Month, Day).

name(Year, Month, Day, Name) :-

person(Name, Year, Month, Day).



**RESULT:**

The prolog program to solve the name with DOB problem was executed.

**19 Write a Prolog Program for STUDENT-TEACHER-SUB-CODE.**

**AIM:**

To write the prolog program for STUDENT-TEACHER-SUB-CODE.

**ALGORITHM:**

1. Define facts as teaches(Teacher, Subject, Code), studies(Student, Subject, Code).
2. Add records for each relationship.
3. Write queries to find which teacher teaches which student.
4. Query to list all subjects taught by a specific teacher.

**Program:**

student(sai, csa1738).

teacher(kumar, csa1738).

subject\_code(csa1738, 'AI').

student\_subject(Student, SubjectCode) :-

student(Student, SubjectCode).

student\_teacher(Student, Teacher) :-

student(Student, SubjectCode), teacher(Teacher, SubjectCode).

teacher\_subject(Teacher, SubjectCode) :-

teacher(Teacher, SubjectCode).

subject\_name(SubjectCode, SubjectName) :-

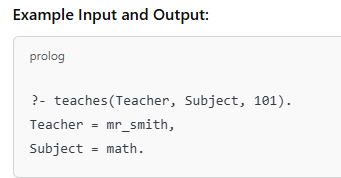
subject\_code(SubjectCode, SubjectName).

student\_subject\_name(Student, SubjectName) :-

student(Student, SubjectCode), subject\_code(SubjectCode, SubjectName).

student\_teacher\_subject(Student, Teacher, SubjectName) :-

student(Student, SubjectCode), teacher(Teacher, SubjectCode), subject\_code(SubjectCode, SubjectName).



**Result:**

The prolog program to solve STUDENT TEACHER SUBCODE problem was executed.

**20 Write a Prolog Program for PLANETS DB**

**AIM:**

To write the prolog program for Planets DB .

**ALGORITHM:**

1. Define each planet as a fact, e.g., planet(Name, Distance, Size, OrbitTime).
2. Input details of each planet.
3. Query to retrieve specific planet information.
4. Query by planet name to get details.

**Program:**

planet(mercury, 1, terrestrial).

planet(venus, 2, terrestrial).

planet(earth, 3, terrestrial).

planet(mars, 4, terrestrial).

planet(jupiter, 5, gas\_giant).

planet(saturn, 6, gas\_giant).

planet(uranus, 7, gas\_giant).

planet(neptune, 8, gas\_giant).

planetName(Order, Name) :-

planet(Name, Order, \_).

planetType(Name, Type) :-

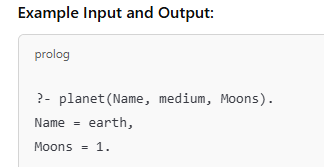
planet(Name, \_, Type).

terrestrialPlanets(Planets) :-

findall(Name, planet(Name, \_, terrestrial), Planets).

gasGiants(Planets) :-

findall(Name, planet(Name, \_, gas\_giant), Planets).



**Result:**

The prolog program to solve PLANETS DB problem was executed.

**21 Write a Prolog Program to implement Towers of Hanoi**

**AIM:**

To write the prolog programto implement Towers of Hanoi.

**ALGORITHM:**

1. Define base case for moving one disk.
2. For N disks, recursively move N-1 disks to the auxiliary pole.
3. Move the Nth disk directly to the destination.
4. Continue until all disks are moved.

**Program:**

hanoi(0, \_, \_, \_, []) :- !.

hanoi(N, Source, Destination, Auxiliary, Moves) :-

N > 0,

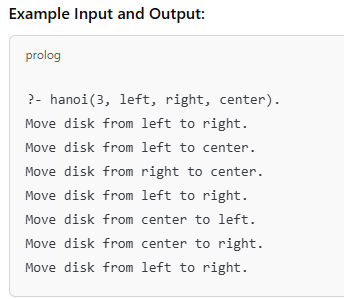
N1 is N - 1,

hanoi(N1, Source, Auxiliary, Destination, Moves1),

MoveN = move(Source, Destination),

hanoi(N1, Auxiliary, Destination, Source, Moves2),

append(Moves1, [MoveN|Moves2], Moves).



**Result:**

The prolog program to solve Towers of Hanoi problem was executed.

**22 Write a Prolog Program to print particular bird can fly or not. Incorporate required queries.**

**AIM:**

To write the prolog programto print particular bird can fly or not.

**ALGORITHM:**

1. Define birds that can and cannot fly, e.g., can\_fly(eagle). and cannot\_fly(penguin).
2. Query if a particular bird can fly.
3. Include a rule to check if a bird can fly.
4. Provide a response based on the bird type.

**Program:**

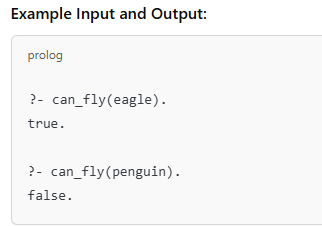
canfly(sparrow).

canfly(eagle).

canfly(peacock).

cannotfly(ostrich).

cannotfly(penguin).



**Result:**

The prolog program to solve Birds can fly or not problem was executed.

**23 Write the prolog program to implement family tree.**

**AIM:**

To write the prolog program to implement Family tree.

**ALGORITHM:**

1. Define family relations as facts, e.g., parent(Parent, Child).
2. Add rules for relationships like sibling, cousin.
3. Query relationships (e.g., grandparents, siblings).
4. Define rules to answer relationship-based questions.

**Program:**

% Facts

parent(john, mary).

parent(john, david).

parent(mary, susan).

parent(david, tom).

parent(david, anna).

male(john).

male(david).

male(tom).

female(mary).

female(susan).

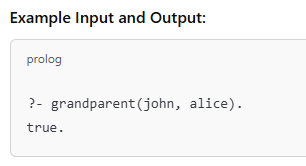
female(anna).

% Rules

father(F, C) :- parent(F, C), male(F).

mother(M, C) :- parent(M, C), female(M).

sibling(X, Y) :- parent(P, X), parent(P, Y), X \= Y.



**Result:**

The prolog program to solve Family tree problem was executed.

**24 Write a Prolog Program to suggest Dieting System based on Disease.**

**AIM:**

To write the prologProgram to suggest Dieting System based on Disease.

**ALGORITHM:**

1. Define dietary recommendations based on diseases, e.g., diet(diabetes, DietPlan).
2. Input disease-diet pairs.
3. Query for diet suggestions based on disease.
4. Return diet plan according to the queried disease.

**Program:**

% Facts

diet(diabetes, 'Low sugar, high fiber, whole grains, vegetables, lean protein').

diet(hypertension, 'Low sodium, high potassium, fruits, vegetables, whole grains').

diet(heart\_disease, 'Low saturated fat, high omega-3, fruits, vegetables, whole grains').

diet(obesity, 'Balanced diet, portion control, high fiber, lean protein').

diet(anemia, 'High iron, vitamin C, leafy greens, red meat, beans').

diet(gastrointestinal\_disorder, 'Low fiber, bland diet, avoid spicy foods, small frequent meals').

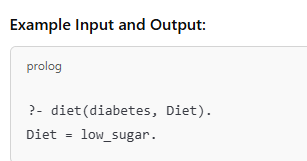
% Rules

suggest\_diet(Disease, Diet) :- diet(Disease, Diet).

% Example Queries

% suggest\_diet(diabetes, Diet).

% suggest\_diet(hypertension, Diet).



**Result:**

The prolog program to solve Dieting system problem was executed.

**25 Write a Prolog program to implement Monkey Banana Problem**

**AIM:**

To write the prolog program to implement Monkey Banana Problem.

**ALGORITHM:**

1. Define initial state of monkey and banana.
2. Define actions (climb, move, grasp).
3. Recursively apply actions to reach banana.
4. Query the sequence to find if monkey gets banana.

**Program:**

% Define the initial state

initial\_state(state(at\_door, on\_floor, at\_window, has\_not)).

% Define the goal state

goal\_state(state(\_, \_, \_, has)).

% Define the possible actions

action(state(middle, on\_box, middle, has\_not), grasp, state(middle, on\_box, middle, has)).

action(state(P, on\_floor, P, H), climb\_box, state(P, on\_box, P, H)).

action(state(P1, on\_floor, P1, H), push\_box(P1, P2), state(P2, on\_floor, P2, H)).

action(state(P1, on\_floor, B, H), walk(P1, P2), state(P2, on\_floor, B, H)).

% Define a plan to achieve the goal state

plan(State, [], State) :- goal\_state(State).

plan(State1, [Action | RestActions], State3) :-

action(State1, Action, State2),

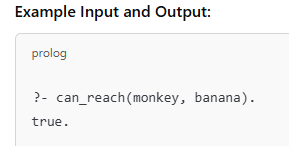
plan(State2, RestActions, State3).

% Query to find the plan

find\_plan(Plan) :-

initial\_state(State),

plan(State, Plan, \_).



**Result:**

The prolog program to solve Monkey Banana problem was executed.

**26 Write a Prolog Program for fruit and its color using Back Tracking**

**AIM:**

To write the prolog program for fruit and its color using Back Tracking.

**ALGORITHM:**

1. Define fruit-color pairs, e.g., color(apple, red).
2. Query with different fruits for colors.
3. Use backtracking to retrieve multiple colors.
4. Return colors for queried fruits.

**Program:**

% Facts: fruit and its color

fruit\_color(apple, red).

fruit\_color(banana, yellow).

fruit\_color(grape, purple).

fruit\_color(lemon, yellow).

fruit\_color(orange, orange).

fruit\_color(cherry, red).

fruit\_color(plum, purple).

% Rule to find fruits of a specific color

fruits\_of\_color(Color, Fruit) :-

fruit\_color(Fruit, Color).

% Rule to list all fruits of a specific color using backtracking

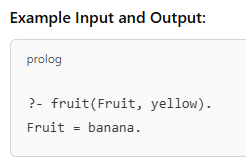
list\_fruits\_of\_color(Color) :-

fruits\_of\_color(Color, Fruit),

writeln(Fruit),

fail.

list\_fruits\_of\_color(\_).



**Result:**

The prolog program to solve Fruit Coloring problem was executed.

**27 Write a Prolog Program to implement Best First Search algorithm**

**AIM:**

To write the prolog programto implement Best First Search algorithm.

**ALGORITHM:**

1. Define graph edges and heuristic costs.
2. Implement a best-first strategy to traverse nodes.
3. Define goal-checking conditions.
4. Query for shortest path from start to goal.

**Program:**

% Define edges of the graph with their costs

edge(a, b, 1).

edge(a, c, 3).

edge(b, d, 3).

edge(b, e, 6).

edge(c, e, 2).

edge(d, f, 1).

edge(e, f, 2).

% Define the heuristic values (estimated cost to reach the goal)

heuristic(a, 6).

heuristic(b, 4).

heuristic(c, 5).

heuristic(d, 2).

heuristic(e, 1).

heuristic(f, 0). % Goal node

% Best First Search algorithm

best\_first\_search(Start, Goal, Path) :-

heuristic(Start, H),

bfs([[Start, H]], Goal, [], % Define edges of the graph with their costs

edge(a, b, 1).

edge(a, c, 3).

edge(b, d, 3).

edge(b, e, 6).

edge(c, e, 2).

edge(d, f, 1).

edge(e, f, 2).

% Define the heuristic values (estimated cost to reach the goal)

heuristic(a, 6).

heuristic(b, 4).

heuristic(c, 5).

heuristic(d, 2).

heuristic(e, 1).

heuristic(f, 0). % Goal node

% Best First Search algorithm

best\_first\_search(Start, Goal, Path) :-

heuristic(Start, H),

bfs([[Start, H]], Goal, [], Path).

% Helper predicate to implement BFS

bfs([[Goal|Path]|\_], Goal, \_, [Goal|Path]).

bfs([[Current|Path]|Rest], Goal, Visited, FinalPath) :-

findall([Next, H, Current|Path],

(edge(Current, Next, \_),

\+ member(Next, Visited),

heuristic(Next, H)),

Neighbors),

append(Rest, Neighbors, NewFrontier),

sort(2, @=<, NewFrontier, SortedFrontier),

bfs(SortedFrontier, Goal, [Current|Visited], FinalPath).

% Query to find the path

find\_path(Start, Goal, Path) :-

best\_first\_search(Start, Goal, RevPath),

reverse(RevPath, Path).



**Result:**

The prolog program to solve BFS problem was executed.

**28 Write the prolog program for Medical Diagnosis**

**AIM:**

To write the prolog program for Medical Diagnosis.

**ALGORITHM:**

1. Define symptoms and corresponding diseases.
2. Add rules to infer diseases from symptoms.
3. Query symptoms to get possible diseases.
4. Suggest probable diagnoses based on inputs.

**Program:**

% Define symptoms

symptom(john, fever).

symptom(john, cough).

symptom(john, headache).

symptom(mary, sore\_throat).

symptom(mary, cough).

symptom(mary, fatigue).

symptom(tom, rash).

symptom(tom, fever).

symptom(tom, headache).

% Define diseases and their associated symptoms

disease(flu, [fever, cough, headache, fatigue]).

disease(cold, [cough, sore\_throat, fatigue]).

disease(measles, [rash, fever, headache]).

% Rule to diagnose a disease based on symptoms

diagnose(Patient, Disease) :-

symptom(Patient, Symptom1),

symptom(Patient, Symptom2),

symptom(Patient, Symptom3),

disease(Disease, Symptoms),

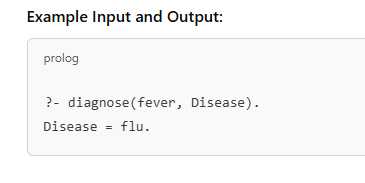
member(Symptom1, Symptoms),

member(Symptom2, Symptoms),

member(Symptom3, Symptoms).

% Query example

% ?- diagnose(john, Disease).



**Result:**

The prolog program to medical diagnosis solve problem was executed.

**29 Write a Prolog Program for forward Chaining. Incorporate required queries.**

**AIM:**

To write the prolog program for forward Chaining. Incorporate required queries.

**ALGORITHM:**

1. **Define facts and rules for forward chaining.**
2. **Process known facts to infer new facts.**
3. **Continue until no new facts can be derived.**
4. **Query based on forward-chained conclusions**

**Program:**

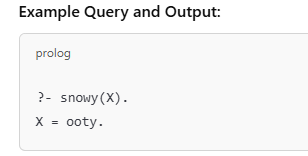
rainy(chennai).

rainy(coimbatore).

rainy(ooty).

cold(ooty).

snowy(X):-rainy(X),cold(X).



**Result:**

The prolog program to solve Forward chaining problem was executed.

**30 Write a Prolog Program for backward Chaining. Incorporate required queries**

**AIM:**

To write the prolog program forbackward Chaining. Incorporate required queries.

**ALGORITHM:**

1. Define rules and target goal.
2. Check if goal can be satisfied by facts.
3. Trace back to determine if goal is true.
4. Return result based on backtracking**.**

**Program:**

% Facts

fact(sunny).

fact(weekend).

fact(raining).

fact(weekday).

% Rules

rule(go\_beach) :-

fact(sunny),

fact(weekend).

rule(watch\_movie) :-

fact(raining).

rule(stay\_home) :-

fact(sunny),

fact(weekday).

% To deduce a fact

deduce(Fact) :-

fact(Fact).

deduce(Fact) :-

rule(Fact),

\+ fact(Fact),

assertz(fact(Fact)),

write('Derived: '), write(Fact), nl.



**Result:**

The prolog program to solve backword chaining problem was executed.

**31 Create a Web Blog using Word press to demonstrate Anchor Tag, Title Tag, etc**

**AIM:**

To Create a Web Blog using Word press to demonstrate Anchor Tag, Title Tag, etc

**ALGORITHM:**

1. Define page structure with anchor tags.
2. Add titles, headers, and tags for SEO.
3. Style with CSS for a responsive layout.
4. Publish the blog page.

**Program:**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>My Blog Post</title>

</head>

<body>

<h1>Welcome to My Blog</h1>

<p>This is an example of a blog post demonstrating various HTML elements.</p>

<h2>Using Anchor Tags</h2>

<p><a href="https://example.com">Click here to visit Example</a></p>

<h2>Using Bold and Italics</h2>

<p>This is <b>bold text</b> and this is <i>italic text</i>.</p>

<h2>Using Lists</h2>

<ul>

<li>First item in unordered list</li>

<li>Second item in unordered list</li>

</ul>

<ol>

<li>First item in ordered list</li>

<li>Second item in ordered list</li>

</ol>

<h2>Using Images</h2>

<p><img src="https://example.com/image.jpg" alt="Example Image"></p>

<p>Thank you for reading!</p>

</body>

</html>

**Result:**

The web blog program was executed.