VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Harshitha H G (1BM23CS108)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in

COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Aug 2025 to Dec 2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Harshitha H G (1BM23CS108),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Anusha S Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/Harshithahrgopal/AI_Lab

Program 1
Implement Tic –Tac –Toe Game Implement
vacuum cleaner agent

Algorithm:

Thursday

Date 21/8/25

1) Algorithm for TIC-TAC-TOE game :-

Step-1:- User will block or enter their pawn at any random position initially.

Step-2:- System will block at position which is along the row or coloumn or diagonal to the user entry.

Step-3:- System has to block at position where two concurrent user entries are there in a row/column/diagonal pos^n.

Step-4:- If two opponents entries are not in same row/column/diagonal position then it need to check all the future possible deadlock that user may cause, to that crucial vacant box system should enter its pawn.

Step-5:- If both Step-3 and Step-4 is not possible then system should enter in the position where its winning possibility is more.

Step-6:- If winning possibility is less (i.e., only two concurrent system entries) then it should search for vacant space where system should cause deadlock to user (i.e., 2 available spaces causes system to win).

2) All moving directionality implementation of vaccum cleaner

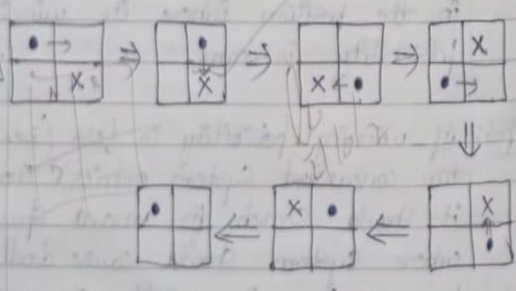
Step-1:- From the position of vacuum cleaner check the distance of dirt in all four directions.

Step-2:- Among the distance calculated, find the minimum/short distance and move in ~~that~~ direction.

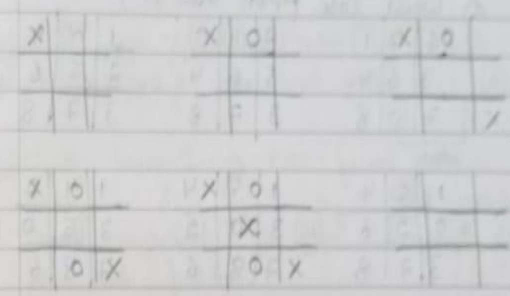
Step-3:- At the arrived position it has to suck the dirt, if it is not there then again from that position to ~~that~~ desired dirt again it should check the minimum distance

Step-4:- with the help of recurssion it should check ~~for~~ each entered block's distance from the dirt and approach to the dirt along minimum distance path.

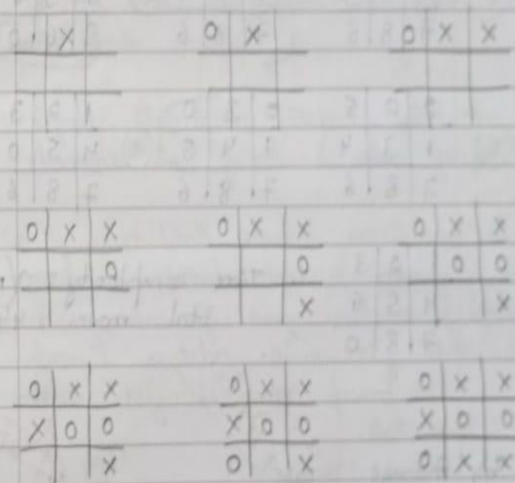
Eg



Example output of Tic-Tac-Toe:



user wins!



System wins!

Code:

Tic –Tac –Toe Game

```
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)


def check_winner(board, player):
    # Check rows, columns and diagonals
    for i in range(3):
        if all([cell == player for cell in board[i]]) or \
           all([board[j][i] == player for j in range(3)]):
            return True

    if all([board[i][i] == player for i in range(3)]) or \
       all([board[i][2 - i] == player for i in range(3)]):
        return True

    return False


def is_full(board):
    return all(cell in ['X', 'O'] for row in board for cell in row)


def get_move(player):
    while True:
        try:
            move = input(f"Player {player}, enter your move (row and column: 1 1): ")
            row, col = map(int, move.split())
            if row in [1, 2, 3] and col in [1, 2, 3]:
                return row - 1, col - 1
            else:
                print("Invalid input. Enter numbers between 1 and 3.")
        except ValueError:
            print("Invalid input. Enter two numbers separated by space.")


def play_game():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = "X"

    while True:
        print_board(board)
        row, col = get_move(current_player)
```

```
        if board[row][col] != " ":
            print("That spot is taken. Try again.")
            continue

        board[row][col] = current_player

        if check_winner(board, current_player):
            print_board(board)
            print(f"Player {current_player} wins!")
            break

        if is_full(board):
            print_board(board)
            print("It's a draw!")
            break

        current_player = "O" if current_player == "X" else "X"


if __name__ == "__main__":
    play_game()
```

Output:

```
  |   |
---------
  |   |
---------
  |   |
---------
Player X, enter your move (row and column: 1 1): 1 1
X |   |
---------
  |   |
---------
  |   |
---------
Player O, enter your move (row and column: 1 1): 1 2
X | O |
---------
  |   |
---------
  |   |
---------
Player X, enter your move (row and column: 1 1): 1 3
X | O | X
---------
  |   |
---------
  |   |
---------
Player O, enter your move (row and column: 1 1): 2 2
X | O | X
---------
  | O |
---------
  |   |
---------
Player X, enter your move (row and column: 1 1): 3 3
X | O | X
---------
  | O |
---------
  |   | X
---------
Player O, enter your move (row and column: 1 1): 3 1
X | O | X
---------
  | O |
---------
O |   | X
---------
Player X, enter your move (row and column: 1 1): 2 3
X | O | X
---------
  | O | X
---------
O |   | X
---------
Player X wins!
```

Vacuum Cleaner

```
def vacuum_simulation():
    cost = 0

    # Get initial states and location
    state_A = int(input("Enter state of A (0 for clean, 1 for dirty): "))
    state_B = int(input("Enter state of B (0 for clean, 1 for dirty): "))
    location = input("Enter location (A or B): ").upper()

    # Vacuum operation loop
    while True:
        if location == 'A':
            if state_A == 1:
                print("Cleaning A.")
                state_A = 0
                cost += 1
            elif state_B == 1:
                print("Moving vacuum right")
                location = 'B'
                cost += 1
            else:
                print("Turning vacuum off")
                break

        elif location == 'B':
            if state_B == 1:
                print("Cleaning B.")
                state_B = 0
                cost += 1
```

```
        elif state_A == 1:
            print("Moving vacuum left")
            location = 'A'
            cost += 1
        else:
            print("Turning vacuum off")
            break
    print(f"Cost: {cost}")
    print(f"{{'A': {state_A}, 'B': {state_B}}}")


vacuum_simulation()
```

OUTPUT

```
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A
Cleaning A.
Turning vacuum off
Cost: 1
{'A': 0, 'B': 0}
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)

Implement Iterative deepening search algorithm

Algorithm:

3) 8-puzzle game:-

a) Worst case [more than 31 moves] initial

(5)

5	2	1
3	6	4
7	0	8

2	1	0
5	6	4
3	7	8

(11)

1	0	4
2	5	6
3	7	8

rotate

(13)

1	5	4
0	2	6
3	7	8

(18)

1	5	4
3	0	2
7	8	6

(21)

1	4	2
3	5	0
7	8	6

rotate

(24)

1	2	0
3	4	5
7	8	6

(27)

1	2	5
0	3	4
7	8	6

(30)

1	2	5
7	3	4
8	6	0

(35)

2	0	5
1	3	4
7	8	6

(38)

2	3	0
1	4	5
7	8	6

(43)

1	2	3
4	5	0
7	8	6

(44)

1	2	3
4	5	6
7	8	0

Time complexity: O(n^m)

Total moves: 44

b) Average case

1	2	3
4	5	6
0	7	8

1	2	3
4	5	6
7	8	0

⇒ O(n^m)

⇒ 2 moves

c) Best case:- (Goal state)

1	2	3
4	5	6
7	8	9

⇒ O(1)

⇒ 0 moves

Thursday 11.9.2025

Algorithm for IDDFS:-

Step-1 → Set dept-limit = 0 initially, which checks only the root node.

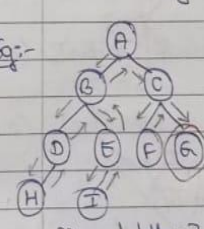
Step-2 → If element is not found then increase dept-limit by 1 and again start searching for the element in DFS fasion.

Step-3:- After every iteration start searching for the element from root to dept limit.

Step-4:- Repeat Step-2 and Step-3 until the element is found in the graph.

Step-5:- If dept limit reaches maximum value that is total dept of given graph then also if any entries of graph doesn't matches given key then return that key is not found.

Eg:-



Root Goal

i) Depth = 0 : A ≠ G.

ii) Depth = 1 : AB & AC

iii) Depth = 2 : ABD & ABE

iv) Depth = 3 : ABDH & ABEI

Backtrack.

ACF ; G not found

ACG.

Max-depth = 3

| HI | | DE FG | | ABG | | A |

Final: Path: ABDHDBEIEBACFCG.

⇒ ABDHEICFG

ABDECFG.

A

ABC

ABDEC

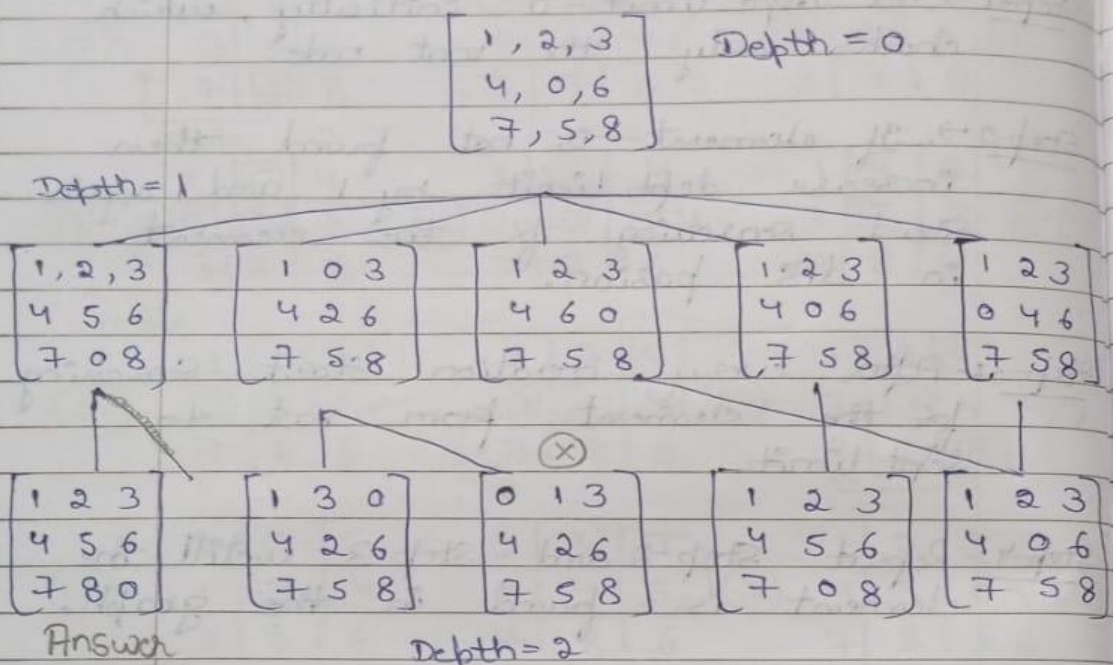
ABDHEC

ABDHEIC

ABDHEICF

ABDHEICFG

output



Answer

Solution found at depth 2 in 2 moves!
Soution path:

1 2 3
4 0 6
7 5 8

1 2 3
4 5 6
7 0 8

1 2 3
4 5 6
7 8 0

Code:

Usig DFS 8 puzzel without heuristic

```
# Goal state
goal = ((1, 2, 3),
        (8, 0, 4),
        (7, 6, 5))

# Moves: Up, Down, Left, Right
moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

def get_neighbors(state):
    # Find the empty tile (0)
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                x, y = i, j
                break

    neighbors = []
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            # Swap empty tile with adjacent tile
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))
    return neighbors

def dfs_limited(start, depth_limit):
    stack = [(start, [start])]
    visited = set([start])

    while stack:
        current, path = stack.pop()

        if current == goal:
            return path

        if len(path) - 1 >= depth_limit:  # already reached depth limit
            continue

        for neighbor in get_neighbors(current):
            if neighbor not in visited:
                visited.add(neighbor)
                stack.append((neighbor, path + [neighbor]))

    return None
```

```
# Example start state
start = ((2, 8, 3),
         (1, 6, 4),
         (7, 0, 5))

solution_path = dfs_limited(start, depth_limit=5)

if solution_path:
    print(f"Solution found in {len(solution_path) - 1} moves:")
    for state in solution_path:
        for row in state:
            print(row)
        print()
else:
    print("No solution found within 5 moves.")
```

OUTPUT

```
Solution found in 5 moves:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

(2, 8, 3)
(1, 0, 4)
(7, 6, 5)

(2, 0, 3)
(1, 8, 4)
(7, 6, 5)

(0, 2, 3)
(1, 8, 4)
(7, 6, 5)

(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

(1, 2, 3)
(8, 0, 4)
(7, 6, 5)
```

Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS)

```
from copy import deepcopy
GOAL_STATE = [
    [1, 2, 3],
    [4, 5, 0],
    [6, 7, 8]
]

# Possible moves of the blank (0) tile: up, down, left, right
MOVES = [(-1, 0), (1, 0), (0, -1), (0, 1)]

def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def is_goal(state):
    return state == GOAL_STATE

def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)

    for dx, dy in MOVES:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = deepcopy(state)
            # Swap blank with neighbor
```

```
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(new_state)
    return neighbors

def dfs(state, depth, limit, path, visited):
    if is_goal(state):
        return path + [state]
    if depth == limit:
        return None

    for neighbor in get_neighbors(state):
        # To avoid cycles, do not revisit states in current path
        if neighbor not in visited:
            result = dfs(neighbor, depth + 1, limit, path + [state], visited + [neighbor])
            if result is not None:
                return result
    return None

def iterative_deepening_search(initial_state, max_depth=50):
    for depth_limit in range(max_depth):
        print(f"Searching with depth limit = {depth_limit}")
        result = dfs(initial_state, 0, depth_limit, [], [initial_state])
        if result is not None:
            return result
    return None

def print_state(state):
    for row in state:
        print(' '.join(str(x) for x in row))
    print()
```

```python
if __name__ == "__main__":
    initial_state = [
        [1, 2, 3],
        [4, 0, 5],
        [6, 7, 8]
    ]

    solution = iterative_deepening_search(initial_state)

    if solution:
        print(f"Solution found in {len(solution)-1} moves!")
        for step, state in enumerate(solution):
            print(f"Step {step}:")
            print_state(state)
    else:
        print("No solution found.")
```

OUTPUT

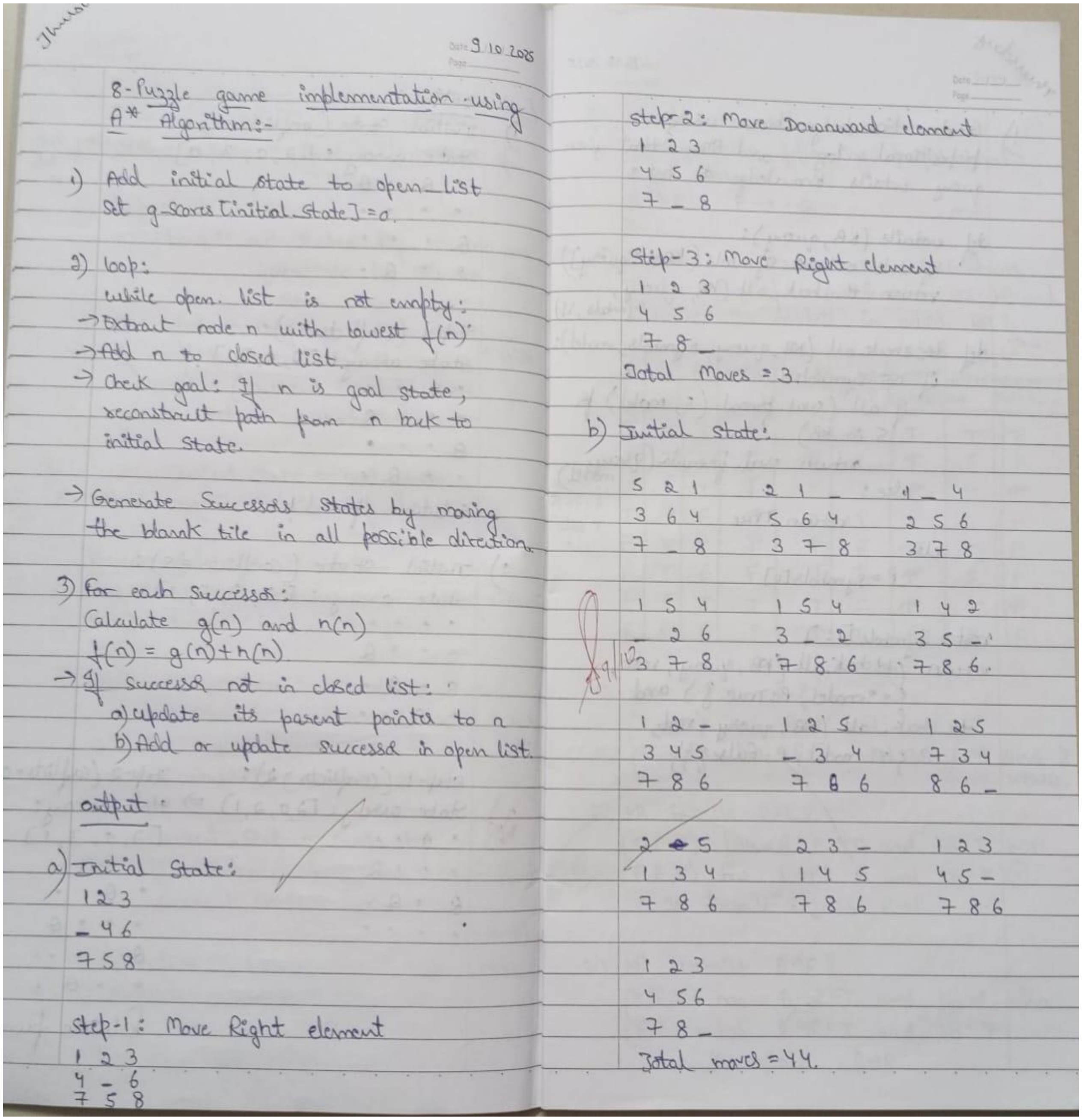
```
Searching with depth limit = 0
Searching with depth limit = 1
Solution found in 1 moves!
Step 0:
1 2 3
4 0 5
6 7 8

Step 1:
1 2 3
4 5 0
6 7 8
```

Program 3

Implement A* search algorithm

Algorithm:



Date: 9/10/2025

8-Puzzle game implementation using A* Algorithm:-

1) Add initial state to open_list
   Set g-Scores[initial_state] = 0.

2) loop:
   while open list is not empty:
   -> Extract node n with lowest f(n)
   -> Add n to closed list.
   -> Check goal: If n is goal state, reconstruct path from n back to initial state.

   -> Generate Successors states by moving the blank tile in all possible direction.

3) For each successor:
   Calculate g(n) and h(n)
   f(n) = g(n) + h(n).
   -> If successor not in closed list:
   a) update its parent pointer to n
   b) Add or update successor in open list.

output:

a) Initial State:

1 2 3
_ 4 6
7 5 8

Step-1: Move Right element

1 2 3
4 _ 6
7 5 8

Step-2: Move Downward element

1 2 3
4 5 6
7 _ 8

Step-3: Move Right element

1 2 3
4 5 6
7 8 _

Total Moves = 3.

b) Initial state:

5 2 1 3 6 4 7 _ 8	2 1 _ 5 6 4 3 7 8	1 _ 4 2 5 6 3 7 8
1 5 4 _ 2 6 3 7 8	1 5 4 3 _ 2 7 8 6	1 4 2 3 5 _ 7 8 6
1 2 _ 3 4 5 7 8 6	1 2 5 _ 3 4 7 8 6	1 2 5 7 3 4 8 6 _
2 _ 5 1 3 4 7 8 6	2 3 _ 1 4 5 7 8 6	1 2 3 4 5 _ 7 8 6

1 2 3
4 5 6
7 8 _

Total moves = 44.

Code:
Misplace Tiles

```
import heapq


# Goal state

goal = ((1, 2, 3),

        (8, 0, 4),

        (7, 6, 5))


# Moves: Up, Down, Left, Right

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


# Heuristic: Misplaced tiles

def misplaced_tiles(state):

    count = 0

    for i in range(3):

        for j in range(3):

            if state[i][j] != 0 and state[i][j] != goal[i][j]:

                count += 1

    return count


# Find blank position (0)

def find_blank(state):

    for i in range(3):

        for j in range(3):

            if state[i][j] == 0:

                return i, j

# Generate neighbors

def get_neighbors(state):

    neighbors = []
```

```
    x, y = find_blank(state)

    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state)
    return neighbors
# A* Search
def astar(start):
    pq = []
    heapq.heappush(pq, (misplaced_tiles(start), 0, start, []))
    visited = set()

    while pq:
        f, g, state, path = heapq.heappop(pq
        if state == goal:
            return path + [state]
        if state in visited:
            continue
        visited.add(state)
        for neighbor in get_neighbors(state):
            if neighbor not in visited:
                new_g = g + 1
                new_f = new_g + misplaced_tiles(neighbor)
                heapq.heappush(pq, (new_f, new_g, neighbor, path + [state]))

    return None
```

```
# Example usage
start_state = ((2, 8, 3),
               (1, 6, 4),
               (7, 0, 5))

solution = astar(start_state)

# Print solution path
for step in solution:
    for row in step:
        print(row)
    print("------")
```

OUTPUT

```
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)
------
(2, 8, 3)
(1, 0, 4)
(7, 6, 5)
------
(2, 0, 3)
(1, 8, 4)
(7, 6, 5)
------
(0, 2, 3)
(1, 8, 4)
(7, 6, 5)
------
(1, 2, 3)
(0, 8, 4)
(7, 6, 5)
------
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)
------
```

Manhatten:

```
import heapq
goal = ((1, 2, 3),
        (8, 0, 4),
        (7, 6, 5))

moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:
                # goal position of this tile
                goal_x = (value - 1) // 3
                goal_y = (value - 1) % 3
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance
def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
```

```
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))
    return neighbors
def astar(start):
    pq = []
    heapq.heappush(pq, (manhattan_distance(start), 0, start, []))  # (f, g, state, path)
    visited = set()
    while pq:
        f, g, state, path = heapq.heappop(pq)
        if state == goal:
            return path + [state]
        if state in visited:
            continue
        visited.add(state)
        for neighbor in get_neighbors(state):
            if neighbor not in visited:
                new_g = g + 1
                new_f = new_g + manhattan_distance(neighbor)
                heapq.heappush(pq, (new_f, new_g, neighbor, path + [state]))

    return None
start_state = ((2, 8, 3),
               (1, 6, 4),
               (7, 0, 5))
solution = astar(start_state)

if solution is None:
    print("No solution found.")
else:
```

```
print("Solution path:")
for step in solution:
    for row in step:
        print(row)
    print("------")
```

OUTPUT:

```
Solution path:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)
------
(2, 8, 3)
(1, 0, 4)
(7, 6, 5)
------
(2, 0, 3)
(1, 8, 4)
(7, 6, 5)
------
(0, 2, 3)
(1, 8, 4)
(7, 6, 5)
------
(1, 2, 3)
(0, 8, 4)
(7, 6, 5)
------
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)
------
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

Thursday.

Date 09 10 2025

4-Queens problem using Hill climbing

Place 4 Queens in 4X4 matrix (Grid) such that
→ Each column has only one Queen
→ Each row has exactly 1 Queen
→ No two Queens are placed diagonally.

Hill climbing Algorithm:-

1) Initial State: 4X4 matrix is empty.
2) Place the first queen at random position in row-1. [current state]
3). Loop do: [until all queens are placed].
    next ← highest among all the neighbouring states of current state
    if current.state ≥ next.state:
        return current_state.
4) Exit.

Simulation ~~Stream~~ Annealing:-

1) Initial State: Set the value for Temperature T.
   $\Delta E \leftarrow 0$.
2) loop do:
    when $\Delta E > T$: [Threshold temperature].
        $\Delta E -= T$
    elif $\Delta E < T$:
        $\Delta E += T$
    else: if $\Delta E == T$:
        return ΔE.

Pseudo Code for N-Queen: [Hill climbing]

1) current_state ← Random placement of 4 Queen
2) loop:
   neighbour ← all states by moving one queen in its column to another row.
3) next_state ← neighbor with minimum conflicts
4) if conflicts (next_state) ≥ conflicts (current_state) then
   return current_state
   else
   current_state ← next state
5) Repeat until Solution found.

Pseudo Code for Simulation Anncaling:-

1) current_state ← random placement of N-Queen
2) temperature ← initial_temperature
3) while temperature > min_temperature;
   next_state ← random neighbor of current state
   Δ ← conflict (next state) - conflict (current_state)
4) If Δ < 0:
   current_state ← next_state
   else if random(0,1) < exp($-\Delta$/temperature):
   current_State ← next state
   (↓ temp)
5) temperature ← temperature * cooling rate
6) if conflicts (current_State) == 0:
   return current_state
7) return current state

output:-

1) Initial State (conflicts = 1):
State array: [2, 0, 3, 0] row along each col

```
. Q . Q
. . . .
Q . . .
. . Q .
```

Step 1 (conflicts = 0):
State array: [2, 0, 3, 1]

```
. Q . .
. . . Q
Q . . .
. . Q .
```

Solution found!

2) Initial state (conflicts = 5):
State array: [2, 3, 2, 1]

```
. . . .
. . . Q
Q . Q .
. Q . .
```

step-1 (conflicts = 2):
State array: [2, 0, 2, 1] ⇒

```
. Q . .
. . . Q
Q . Q .
. . . .
```

step-2 (conflicts = 0)
state array: [2, 0, 3, 1]

```
. Q . .
. . . Q
Q . . .
. . Q .
```

Solution found

Code:

```
import random


def compute_cost(state):
    n = len(state)
    cost = 0
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j]:            # same row
                cost += 1
            elif abs(state[i] - state[j]) == abs(i - j):  # same diagonal
                cost += 1
    return cost



def get_neighbors(state):
    neighbors = []
    n = len(state)
    for col in range(n):        # pick a column
        for row in range(n):    # try moving queen in this column to another row
            if row != state[col]:
                new_state = state.copy()
                new_state[col] = row
                neighbors.append(new_state)
    return neighbors
```

```
def print_board(state):
    n = len(state)
    for r in range(n):
        line = ""
        for c in range(n):
            line += "Q " if state[c] == r else ". "
        print(line)
    print("")


def hill_climb(initial_state, max_sideways=50):
    current = initial_state
    current_cost = compute_cost(current)
    steps = 0
    sideways_moves = 0

    print("Initial State (cost={}):".format(current_cost))
    print_board(current)

    while True:
        neighbors = get_neighbors(current)
        costs = [compute_cost(n) for n in neighbors]
        min_cost = min(costs)

        if min_cost > current_cost:
            # no better neighbor -> stop
            break
```

```
        # pick one of the best neighbors randomly
        best_neighbors = [n for n, c in zip(neighbors, costs) if c == min_cost]
        next_state = random.choice(best_neighbors)
        next_cost = compute_cost(next_state)

        # handle sideways moves
        if next_cost == current_cost:
            if sideways_moves >= max_sideways:
                break
            else:
                sideways_moves += 1
        else:
            sideways_moves = 0

        current = next_state
        current_cost = next_cost
        steps += 1

        print("Step {} (cost={}):".format(steps, current_cost))
        print_board(current)

        if current_cost == 0:
            print("Solution found in {} steps ✅".format(steps))
            return current

    print("Local minimum reached (cost={}) ❌".format(current_cost))
    return current

initial_state = [3, 1, 2, 0]
```

final = hill_climb(initial_state, max_sideways=10)

OUTPUT:

```
Initial State (cost=2):
. . . Q
. Q . .
. . Q .
Q . . .

Step 1 (cost=2):
. . . Q
Q Q . .
. . Q .
. . . .

Step 2 (cost=1):
. . . Q
Q . . .
. . Q .
. Q . .

Step 3 (cost=1):
. . Q Q
Q . . .
. . . .
. Q . .

Step 4 (cost=0):
. . Q .
Q . . .
. . . Q
. Q . .

Solution found in 4 steps ✅
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:

Pseudo Code for N-Queen: [Hill climbing]

```
1) current_state ← Random placement of
                   4 Queen
2) loop:
   neighbour ← all states by moving one
               queen in its column to
               another row.
3) next_state ← neighbor with minimum
                                  conflicts
4) if conflicts (next_state) ≥
      conflicts (current_state) then
          return current_state
   else
          current_state ← next state
5) Repeat until Solution found
```

Pseudo Code for Simulation Annealing:-

```
1) current state ← random placement of N-Queen
2) temperature ← initial_temperature

3) while temperature > min_temperature;
       next_state ← random neighbor of current
                                        State
       Δ ← conflict (next state) -
           conflict (current_state)
4) If Δ<0:
       current_State ← next_state
   else if random(0,1) < exp(-Δ/temperature):
       current_State ← next state
                                    (↓ temp)
5) temperature ← temperature * cooling rate
6) if conflicts (current_State) == 0:
       return current_state
7) return current_state
```

output :-

1) Initial State (conflicts = 1):

State array : [2, 0, 3, 0] row along each col

```
. Q . Q
. . . .
Q . . .
. . Q .
```

Step 1 (conflicts = 0):

State array : [2, 0, 3, 1]

```
. Q . .
. . . Q
Q . . .
. . Q .
```

solution found!

2) Initial state (conflicts = 5):

State array : [2, 3, 2, 1]

```
. . . .
. . . Q
Q . Q .
. Q . .
```

step-1 (conflicts = 2):

State array: [2, 0, 2, 1] ⇒

```
. Q . .
. . . Q
Q . Q .
. . . .
```

step-2 (conflicts = 0)

state array: [2, 0, 3, 1]

```
. Q . .
. . . Q
Q . . .
. . Q .
```

Solution found

Code:

```
from scipy.optimize import dual_annealing
import numpy as np


def queens_max(x):
    cols = np.round(x).astype(int)
    n = len(cols)


    if len(set(cols)) < n:
        return 1e6
    attacks = 0
    for i in range(n):
        for j in range(i + 1, n):
            if abs(i - j) == abs(cols[i] - cols[j]):
                attacks += 1
    return attacks
n = 8
bounds = [(0, n - 1)] * n
result = dual_annealing(queens_max, bounds)


best_cols = np.round(result.x).astype(int).tolist()
not_attacking = n


print(f"The best position found is: {best_cols}")
print(f"The number of queens that are not attacking each other is: {not_attacking}")
```

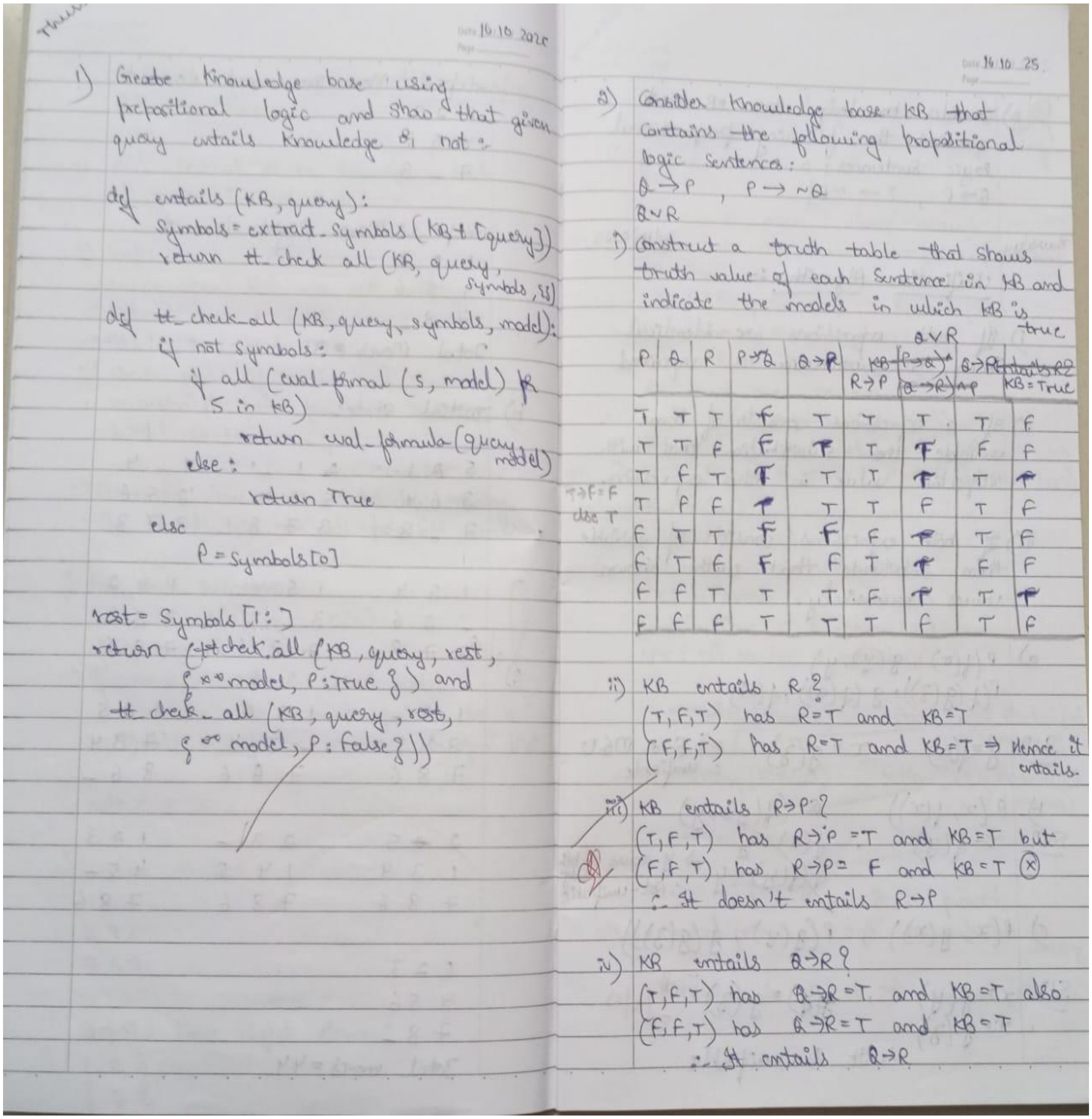
OUTPUT:

```
The best position found is: [7, 4, 6, 1, 3, 5, 0, 2]
The number of queens that are not attacking each other is: 8
```

Program 6

Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

Algorithm:



16.10.2025

1) Create knowledge base using prepositional logic and shao that given query entails knowledge or not :-

```
def entails (KB, query):
    Symbols = extract_symbols (KB + [query])
    return tt_check all (KB, query,
                                      symbols, {})
def tt_check_all (KB, query, symbols, model):
    if not symbols :
        if all (eval_formal (s, model) for
         s in KB)
            return eval_formula (query,
                                      model)
        else :
            return True
    else
        P = symbols[0]

rest = Symbols [1:]
return (tt check all (KB, query, rest,
        {** model, P: True }) and
  tt_check_all (KB, query, rest,
        {** model, P: False}))
```

16.10.25

2) Consider knowledge base KB that contains the following propositional logic sentences:

$Q \to P$, $P \to \sim Q$

$Q \vee R$

i) Construct a truth table that shows truth value of each sentence in KB and indicate the models in which KB is true

P	Q	R	P→~Q	Q→R	~~KB~~ R→P	Q∨R (P→Q)∧(Q→R)	Q→R∧P	~~entails R2~~ KB = True
T	T	T	F	T	T	T	T	F
T	T	F	F	T	T	T	F	F
T	F	T	T	T	T	T	T	T
T	F	F	T	T	T	F	T	F
F	T	T	F	F	F	T	T	F
F	T	F	F	F	T	T	F	F
F	F	T	T	T	F	T	T	T
F	F	F	T	T	T	F	T	F

T→F = F
else T

ii) KB entails R ?

(T, F, T) has R = T and KB = T

(F, F, T) has R = T and KB = T ⇒ Hence it entails.

iii) KB entails R→P ?

(T, F, T) has R→P = T and KB = T but

(F, F, T) has R→P = F and KB = T (x)

∴ It doesn't entails R→P

iv) KB entails Q→R ?

(T, F, T) has Q→R = T and KB = T also

(F, F, T) has Q→R = T and KB = T

∴ It entails Q→R

Code:

```
import itertools
from sympy import symbols, sympify

A, B, C = symbols('A B C')

alpha_input = input("Enter alpha (example: A | B): ")
kb_input = input("Enter KB (example: (A | C) & (B | ~C)): ")

alpha = sympify(alpha_input, evaluate=False)
kb = sympify(kb_input, evaluate=False)

GREEN = "\033[92m"
RESET = "\033[0m"

print(f"\nTruth Table for α = {alpha_input}, KB = {kb_input}\n")
print(f"{'A':<6}{'B':<6}{'C':<6}{'α':<10}{'KB':<10}")

entailed = True

for values in itertools.product([False, True], repeat=3):
    subs = {A: values[0], B: values[1], C: values[2]}
    alpha_val = alpha.subs(subs)
    kb_val = kb.subs(subs)

    alpha_str = f"{GREEN}{alpha_val}{RESET}" if kb_val else str(alpha_val)
    kb_str = f"{GREEN}{kb_val}{RESET}" if kb_val else str(kb_val)

    print(f"{str(values[0]):<6}{str(values[1]):<6}{str(values[2]):<6}"
          f"{alpha_str:<10}{kb_str:<10}")
```

```
    if kb_val and not alpha_val:
        entailed = False

if entailed:
    print(f"\n KB |= α holds (KB entails α)\n")
else:
    print(f"\n KB does NOT entail α\n")
```

OUTPUT:

```
Enter alpha (example: A | B): A|B
Enter KB (example: (A | C) & (B | ~C)): (A | C) & (B | ~C)

Truth Table for α = A|B, KB = (A | C) & (B | ~C)

A     B     C     α          KB
False False False False      False
False False True  False      False
False True  False True       False
False True  True  TrueTrue
True  False False TrueTrue
True  False True  True       False
True  True  False TrueTrue
True  True  True  TrueTrue

 KB |= α holds (KB entails α)
```

Program 7
Implement unification in first order logic
Algorithm:

③ [a) Consider knowledge Base KB that contains the following propositional logic sentences:
$Q \rightarrow P$, $P \rightarrow \sim Q$]

Thursday 30/10/2025

Unification Algorithm:-

i) If both expressions are identical then no substitution is required.

ii) If a expressions consists of some variable then substitution that with respective value in other expression.

iii) If both expressions consists of variable then substitute that with known values recursively.

a) $P(f(x), g(y), y)$
$P(f(g(z)), g(f(a)), f(a))$

Soln:- $y = f(a)$, $x = g(z)$ $\Rightarrow \theta$ or MGU
∴ unifiable

b) $Q(x, f(x))$ $Q(\cancel{f(y)}, y)$
Soln: $x = f(y)$, $f(x) = y$
$f(f(y)) = y$ ⇒ y occurs on both sides
∴ Not-unifiable

c) $P(x, g(x))$, $P(g(y), g(g(z)))$

Soln: $x = g(y)$, $\cancel{g(y)} = g(z)$ 29/10/25
$x = g(z)$
∴ It is unifiable

Code:

```
def occurs_check(var, expr):
    if var == expr:
        return True
    if isinstance(expr, tuple):
        return any(occurs_check(var, sub) for sub in expr[1:])  # Skip function symbol
    return False
def substitute(expr, subst):
    if isinstance(expr, str):
        # Follow substitution chain until fully resolved
        while expr in subst:
            expr = subst[expr]
        return expr
    # If it's a function term: (f, arg1, arg2, ...)
    return (expr[0],) + tuple(substitute(sub, subst) for sub in expr[1:])
def unify(Y1, Y2, subst=None):
    if subst is None:
        subst = {}
    Y1 = substitute(Y1, subst)
    Y2 = substitute(Y2, subst)

    # Case 1: identical
    if Y1 == Y2:
        return subst

    # Case 2: Y1 is variable
    if isinstance(Y1, str):
        if occurs_check(Y1, Y2):
            return "FAILURE"
        subst[Y1] = Y2
        return subst
```

```
    # Case 3: Y2 is variable
    if isinstance(Y2, str):
        if occurs_check(Y2, Y1):
            return "FAILURE"
        subst[Y2] = Y1
        return subst

    # Case 4: function mismatch
    if Y1[0] != Y2[0] or len(Y1) != len(Y2):
        return "FAILURE"

    # Case 5: unify arguments
    for a, b in zip(Y1[1:], Y2[1:]):
        subst = unify(a, b, subst)
        if subst == "FAILURE":
            return "FAILURE"

    return subst
expr1 = ("p", "X", ("f", "Y"))
expr2 = ("p", "a", ("f", "b"))

output = unify(expr1, expr2)
print(output)
```

OUTPUT:

```
{'X': 'a', 'Y': 'b'}
```

Program 8

Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

Algorithm:

Thursday

6.11.2025

11. Create a Knowledge base consisting of FOL & prove the given query using fworard reasoning.

Marcus is man : Man(Marcus)

Marcus is Pompeian : Pompeian(Marcus)

All Pompeians are Roman :

$\forall x$ (Pompeian(x) $\rightarrow$ Roman(x))

All Romans are loyal :

$\forall x$ (Roman(x) $\rightarrow$ loyal(x))

All men are persons

$\forall x$ (man(x) $\rightarrow$ Person(x))

All persons are mortal : $\forall x$ (Person(x) $\rightarrow$ Mortal(x))

To prove : Marcus is mortal ? Mortal(Marcus)

Soln: Man(Marcus)

Pompeian(Marcus)

$\forall x$: ~Pompeian(x) ∨ Roman(x)

$\forall x$: ~Roman(x) ∨ loyal(x)

$\forall x$: ~Man(x) ∨ Person(x)

$\forall x$: ~Person(x) ∨ mortal(x)

~Mortal(Marcus)   ~Person(x) ∨ Mortal(x)

x | {Marcus}

~Person(Marcus)   ~Man(x) ∨ Person(x)

x | {Marcus}

~Man(Marcus)   Man(Marcus)

{ } Hence it is proved

$\Rightarrow$ ~Mortal(Marcus) our assumption is wrong.

∴ Marcus is Mortal $\Rightarrow$ Mortal(Marcus)

Code:

```
from collections import deque
class KnowledgeBase:
    def __init__(self):
        self.facts = set()
        self.rules = []
        self.inferred = set()

    def add_fact(self, fact):
        if fact not in self.facts:
            print(f"Adding fact: {fact}")
            self.facts.add(fact)
            return True
        return False

    def add_rule(self, premises, conclusion):
        self.rules.append((premises, conclusion))

    def forward_chain(self):
        agenda = deque(self.facts)

        while agenda:
            fact = agenda.popleft()
            if fact in self.inferred:
                continue
            self.inferred.add(fact)

            for (premises, conclusion) in self.rules:
                if all(p in self.inferred for p in premises):
```

```
            if conclusion not in self.facts:
                print(f"Inferred new fact: {conclusion} from {premises} => {conclusion}")
                self.facts.add(conclusion)
                agenda.append(conclusion)

                if conclusion == 'Criminal(West)':
                    print("\n✅ Goal Reached: West is Criminal")
                    return True
        return False
kb = KnowledgeBase()

kb.add_fact('American(West)')
kb.add_fact('Enemy(Nono, America)')
kb.add_fact('Missile(M1)')
kb.add_fact('Owns(Nono, M1)')

kb.add_rule(premises=['Missile(M1)'], conclusion='Weapon(M1)')

kb.add_rule(premises=['Missile(M1)', 'Owns(Nono, M1)'], conclusion='Sells(West, M1, Nono)')

kb.add_rule(premises=['Enemy(Nono, America)'], conclusion='Hostile(Nono)')
kb.add_rule(premises=['American(West)', 'Weapon(M1)', 'Sells(West, M1, Nono)', 'Hostile(Nono)'],
conclusion='Criminal(West)')
kb.forward_chain()
```

OUTPUT:

```
Adding fact: American(West)
Adding fact: Enemy(Nono, America)
Adding fact: Missile(M1)
Adding fact: Owns(Nono, M1)
Inferred new fact: Weapon(M1) from ['Missile(M1)'] => Weapon(M1)
Inferred new fact: Hostile(Nono) from ['Enemy(Nono, America)'] => Hostile(Nono)
Inferred new fact: Sells(West, M1, Nono) from ['Missile(M1)', 'Owns(Nono, M1)'] => Sells(West, M1, Nono)
Inferred new fact: Criminal(West) from ['American(West)', 'Weapon(M1)', 'Sells(West, M1, Nono)', 'Hostile(Nono)'] => Criminal(West)

✅ Goal Reached: West is Criminal
True
```

Program 9

Create a knowledge base consisting of first order logic statements and prove the given query using Resolution

Algorithm:

12. Convert given FOL into CNF:-

$\forall x\, [\sim\forall y \sim(Animal(y) \vee loves(x,y))] \vee [\exists y\, Loves(y,x)]$

[Remove ~]

$\rightarrow \forall x\, [\exists y \sim\sim(Animal(y) \vee loves(x,y))] \vee [\exists y\, Loves(y,x)$

$\rightarrow \forall x\, [\exists y\, (Animal(y) \vee loves(x,y))] \vee [\exists y\, Loves(y,x)]$

[Standardized] $\rightarrow \forall x\, [\exists y\, (Animal(y) \vee Loves(x,y))] \vee [\exists w\, Loves(w,z)]$

[Skolemize]

$\rightarrow \forall x\, [Animal(A) \vee loves(x,A))] \vee [loves(B,z)]$

[Drop $\forall$]

$\rightarrow Animal(A) \vee loves(x,A) \vee loves(B,z)$

Code:

```
from itertools import combinations


def get_clauses():
    n = int(input("Enter number of clauses in Knowledge Base: "))
    clauses = []
    for i in range(n):
        clause = input(f"Enter clause {i+1}: ")
        clause_set = set(clause.replace(" ", "").split("v"))
        clauses.append(clause_set)
    return clauses


def resolve(ci, cj):
    resolvents = []
    for di in ci:
        for dj in cj:
            if di == ('~' + dj) or dj == ('~' + di):
                new_clause = (ci - {di}) | (cj - {dj})
                resolvents.append(new_clause)
    return resolvents


def resolution_algorithm(kb, query):
    kb.append(set(['~' + query]))
    derived = []
    clause_id = {frozenset(c): f"C{i+1}" for i, c in enumerate(kb)}

    step = 1
    while True:
        new = []
        for (ci, cj) in combinations(kb, 2):
```

```
            resolvents = resolve(ci, cj)
            for res in resolvents:
                if res not in kb and res not in new:
                    cid_i, cid_j = clause_id[frozenset(ci)], clause_id[frozenset(cj)]
                    clause_name = f"R{step}"
                    derived.append((clause_name, res, cid_i, cid_j))
                    clause_id[frozenset(res)] = clause_name
                    new.append(res)
                    print(f"[Step {step}] {clause_name} = Resolve({cid_i}, {cid_j}) → {res or '{}'}")
                    step += 1

                    # If empty clause found → proof complete
                    if res == set():
                        print("\n✅ Query is proved by resolution (empty clause found).")
                        print("\n--- Proof Tree ---")
                        print_tree(derived, clause_name)
                        return True
        if not new:
            print("\n❌ Query cannot be proved by resolution.")
            return False
        kb.extend(new)


def print_tree(derived, goal):
    tree = {name: (parents, clause) for name, clause, *parents in [(r[0], r[1], r[2:][0], r[2:][1]) for r in
derived]}


    def show(node, indent=0):
        if node not in tree:
            print(" " * indent + node)
            return
```

```
        parents, clause = tree[node]
        print(" " * indent + f"{node}: {set(clause) or '{}'}")
        for p in parents:
            show(p, indent + 4)

    show(goal)
```

OUPUT:

```
=== FOL Resolution Demo with Proof Tree ===
Enter number of clauses in Knowledge Base: 3
Enter clause 1: P
Enter clause 2: ~P v Q
Enter clause 3: ~Q
Enter query to prove: Q
[Step 1] R1 = Resolve(C1, C2) → {'Q'}
[Step 2] R2 = Resolve(C2, C4) → {'~P'}
[Step 3] R3 = Resolve(C1, R2) → {}

✅ Query is proved by resolution (empty clause found).

--- Proof Tree ---
R3: {}
    C1
    R2: {'~P'}
        C2
        C4
True
```

Program 10

Implement Alpha-Beta Pruning.

Algorithm:

```
function DLS(node, goal, limit):
    if node = goal then return found
    if limit = 0 then return not found
    for each child of node DO
        If DLS(child, goal, limit-1) = found then
            return found
    End for
    Return not found.
END

4) Implement vaccum cleaner Agent:-
function simple reflex_agent(location, status)
                              suck dirt
    if status == 'Dirty': return "Clean"
    # if clean, move in fixed pattern to cover all
    if location == (1,1): return "Right"
    if loc == (1,2): return "Down"
    if loc == (2,2): return ("Left")
    if loc == (2,1): return ("Up")

5) Unification in FOL:- [2 expression]
Unify (x, y):
-> If x = y: return {}
-> If x is variable: return Substitute (x, y)
-> If y is variable: return Substitute (y, x)
-> If x & y are functions:
     if func names & no of argum differ:
         return Fail
     else:
         return unify each corresponding argu
-> otherwise: return Fail

6) Alpha-Beta Pruning:- (α -> MAX, β -> min)
technique that skips parts of game tree
that never effects final result.

function AlphaBeta(node, depth, α, β, maximizing-Player):
If node is terminal & depth = 0:
    return heuristic_value (node)
If maximizing-Player:
    value = -∞
    for each child of node:
        value = max (value, AlphaBeta(child, depth-1, α, β, False))
        α = max (α, value)
        if α ≥ β: break // prune
    return value
else:
    value = +∞
    for each child of node:
        value = min ( -u-  True))
        β = min (β, value)
        If α ≥ β: break // prune
    return value
```

Code:

```
class Node:
    def __init__(self, name):
        self.name = name
        self.children = []
        self.value = None
        self.pruned = False

def alpha_beta(node, depth, maximizing, values, alpha, beta, index):
    # Terminal node
    if depth == 3:
        node.value = values[index[0]]
        index[0] += 1
        return node.value

    if maximizing:
        best = float('-inf')
        for i in range(2): # 2 children
            child = Node(f"{node.name}{i}")
            node.children.append(child)
            val = alpha_beta(child, depth + 1, False, values, alpha, beta, index)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                node.pruned = True
                break
        node.value = best
        return best
    else:
        best = float('inf')
```

```
        for i in range(2):
            child = Node(f"{node.name}{i}")
            node.children.append(child)
            val = alpha_beta(child, depth + 1, True, values, alpha, beta, index)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                node.pruned = True
                break
        node.value = best
        return best

def print_tree(node, indent=0):
    prune_mark = " [PRUNED]" if node.pruned else ""
    val = f" = {node.value}" if node.value is not None else ""
    print(" " * indent + f"{node.name}{val}{prune_mark}")
    for child in node.children:
        print_tree(child, indent + 4)

# --- main ---
print("=== Alpha-Beta Pruning with Tree ===")
values = list(map(int, input("Enter 8 leaf node values separated by spaces: ").split()))

root = Node("R")
alpha_beta(root, 0, True, values, float('-inf'), float('inf'), [0])

print("\n--- Game Tree ---")
print_tree(root)

print("\nOptimal Value at Root:", root.value)
```

OUTPUT:

```
=== Alpha-Beta Pruning with Tree ===
Enter 8 leaf node values separated by spaces: 3 5 6 9 1 2 0 7

--- Game Tree ---
R = 5
    R0 = 5
        R00 = 5
            R000 = 3
            R001 = 5
        R01 = 6 [PRUNED]
            R010 = 6
    R1 = 2 [PRUNED]
        R10 = 9
            R100 = 9
            R101 = 1
        R11 = 2
            R110 = 2
            R111 = 0

Optimal Value at Root: 5
```