VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Deepthi M (1BM23CS088)

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 **Deepthi M(1BM23CS088),** 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.

Sheetal V A Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

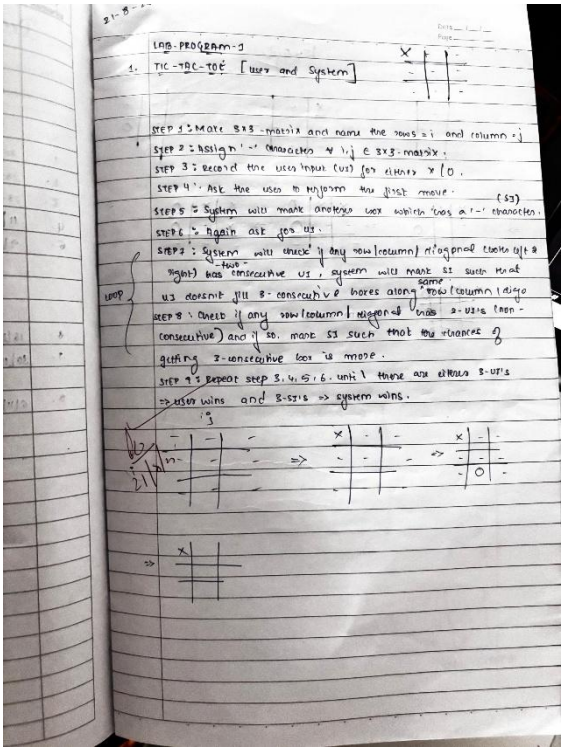
Github Link:
https://github.com/Deepthi-1534/AI-LAB---2025

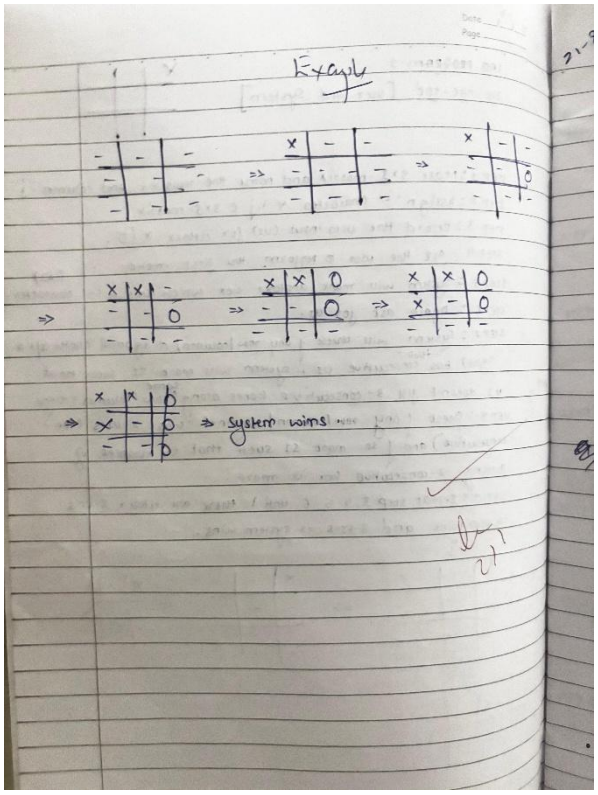
Program 1

Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

1.TIC TAC TOE GAME

Algorithm:





Code:

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


def check_winner(board):
    for row in board:
        if row[0] == row[1] == row[2] != " ":
            return True
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] != " ":
            return True
    if board[0][0] == board[1][1] == board[2][2] != " ":
```

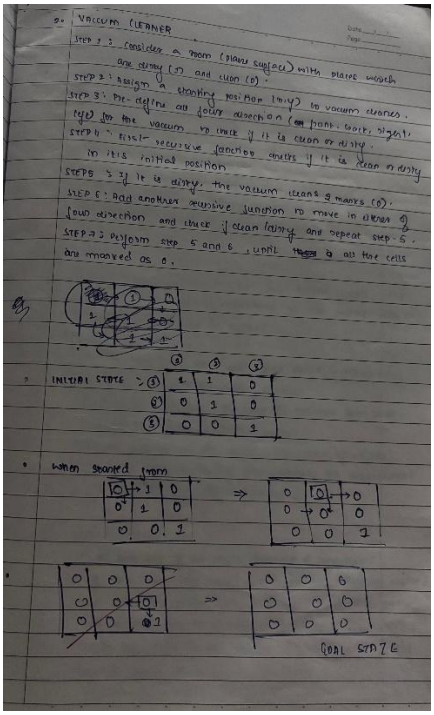
```
        return True
    if board[0][2] == board[1][1] == board[2][0] != " ":
        return True
    return False


def is_full(board):
    for row in board:
        if " " in row:
            return False
    return True


def tic_tac_toe():
    print("Welcome to Tic Tac Toe!")
    board = [[" " for _ in range(3)] for _ in range(3)]
    print_board(board)
    current_player = "X"
    while True:
        try:
            row, col = map(int, input(f"Player {current_player}, enter your move (row and column: 1 1
for top-left): ").split())
            if row < 1 or row > 3 or col < 1 or col > 3:
                print("Invalid position! Enter numbers between 1 and 3.")
                continue
            if board[row - 1][col - 1] != " ":
                print("Cell already taken! Try again.")
                continue
            board[row - 1][col - 1] = current_player
            print_board(board)
            if check_winner(board):
                print(f"🎉 Player {current_player} wins!")
                break
            if is_full(board):
                print("It's a tie!")
                break
            current_player = "O" if current_player == "X" else "X"
        except ValueError:
            print("Invalid input! Enter row and column numbers separated by a space.")


tic_tac_toe()
```

Output:

```
PS C:\Users\BMSCECSE\Desktop\1BF24CS211\AI> python -u "c:\Users\BMSCECSE\Desktop\1BF24CS211\AI\CODE.PY"
_ | _ | _
---------
_ | _ | _
---------
_ | _ | _
---------
AI X moves to (0, 0)
X | _ | _
---------
_ | _ | _
---------
_ | _ | _
---------
Enter your move (row and column, 0-based, separated by space): 1 1
X | _ | _
---------
_ | O | _
---------
_ | _ | _
---------
AI X moves to (0, 1)
X | X | _
---------
_ | O | _
---------
_ | _ | _
---------
Enter your move (row and column, 0-based, separated by space): 2 5
Invalid move. Try again.
Enter your move (row and column, 0-based, separated by space): 2 3
Invalid move. Try again.
Enter your move (row and column, 0-based, separated by space): 2 2
X | X | _
---------
_ | O | _
---------
_ | _ | O
---------
AI X moves to (0, 2)
X | X | X
---------
_ | O | _
---------
_ | _ | O
---------
X wins!
PS C:\Users\BMSCECSE\Desktop\1BF24CS211\AI>
```

2.VACCUM CLEANER

Algorithm:



Code:

```
from typing import Set, Tuple

class Robot:
    def move(self) -> bool:
        """Moves the robot forward by one unit in the current direction."""
        raise NotImplementedError

    def turnLeft(self) -> None:
        """Turns the robot 90 degrees to the left."""
        raise NotImplementedError

    def turnRight(self) -> None:
        """Turns the robot 90 degrees to the right."""
        raise NotImplementedError

    def clean(self) -> None:
        """Cleans the current cell."""
        raise NotImplementedError
```

```
class RobotCleaner:
    # Directions: up, right, down, left
    directions = [(-1, 0), (0, 1), (1, 0), (0, -1)]

    def __init__(self):
        self.visited: Set[Tuple[int, int]] = set()
        self.robot = None

    def cleanRoom(self, robot: Robot):
        self.robot = robot
        self.dfs(0, 0, 0)

    def dfs(self, row: int, col: int, direction: int):
        self.robot.clean()
        self.visited.add((row, col))

        for k in range(4):
            new_direction = (direction + k) % 4
            dr, dc = self.directions[new_direction]
            new_row, new_col = row + dr, col + dc

            if (new_row, new_col) not in self.visited and self.robot.move():
                self.dfs(new_row, new_col, new_direction)

                # Backtrack: turn 180 degrees, move back, then turn 180 degrees again
                self.robot.turnRight()
                self.robot.turnRight()
                self.robot.move()
                self.robot.turnRight()
                self.robot.turnRight()

            self.robot.turnRight()


class MockRobot(Robot):
    def __init__(self):
        # 1 = open space, 0 = obstacle
        self.room = [
            [1,1,1,1,0],
            [1,0,1,0,1],
            [1,1,1,1,1],
            [0,1,0,1,0],
```

```
            [1,1,1,1,1]
        ]
        self.row = 0
        self.col = 0
        self.direction = 0  # 0=up,1=right,2=down,3=left

        # Directions match the RobotCleaner directions order
        self.directions = [(-1,0),(0,1),(1,0),(0,-1)]

    def move(self) -> bool:
        dr, dc = self.directions[self.direction]
        new_row, new_col = self.row + dr, self.col + dc

        if 0 <= new_row < len(self.room) and 0 <= new_col < len(self.room[0]) and
self.room[new_row][new_col] == 1:
            self.row, self.col = new_row, new_col
            # Reduced output: no print here
            return True
        return False

    def turnLeft(self) -> None:
        self.direction = (self.direction - 1) % 4
        # Reduced output: no print here

    def turnRight(self) -> None:
        self.direction = (self.direction + 1) % 4
        # Reduced output: no print here

    def clean(self) -> None:
        print(f"Cleaned cell ({self.row}, {self.col})")


if __name__ == "__main__":
    robot = MockRobot()
    cleaner = RobotCleaner()
    cleaner.cleanRoom(robot)
```

Output:

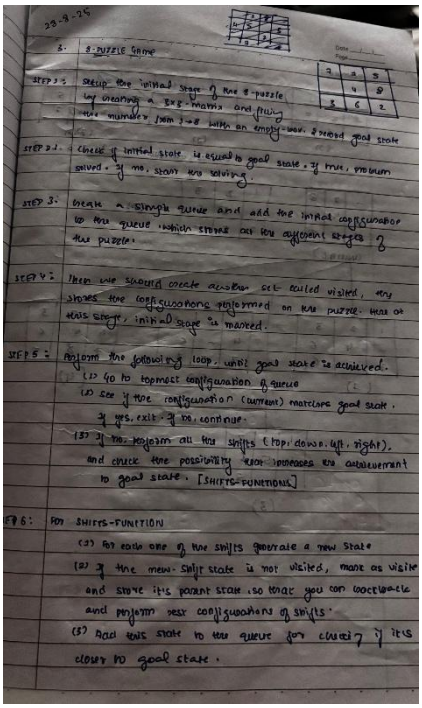
```
PS C:\Users\BMSCECSE\AI> python -u "c:\Users\BMSCECSE\AI\RobotCleaner.py"
Cleaned cell (0, 0)
Cleaned cell (0, 1)
Cleaned cell (0, 2)
Cleaned cell (0, 3)
Cleaned cell (1, 2)
Cleaned cell (2, 2)
Cleaned cell (2, 1)
Cleaned cell (2, 0)
Cleaned cell (1, 0)
Cleaned cell (3, 1)
Cleaned cell (4, 1)
Cleaned cell (4, 0)
Cleaned cell (4, 2)
Cleaned cell (4, 3)
Cleaned cell (4, 4)
Cleaned cell (3, 3)
Cleaned cell (2, 3)
Cleaned cell (2, 4)
Cleaned cell (1, 4)
PS C:\Users\BMSCECSE\AI>
```

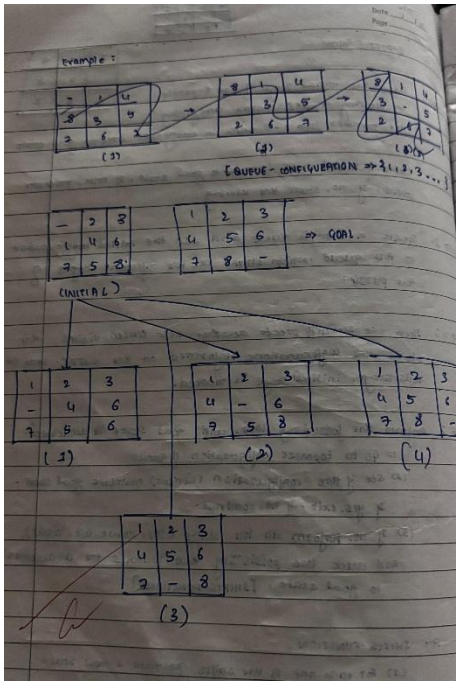
Program 2

Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm

1.8 PUZZLE USING DFS

Algorithm:





Code:

```
def get_neighbors_dfs(state):
    neighbors = []
    index = state.index('0')
    row, col = divmod(index, 3)
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for r, c in moves:
        new_row, new_col = row + r, col + c
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_index = new_row * 3 + new_col
            new_state = list(state)
            new_state[index], new_state[new_index] = new_state[new_index], new_state[index]
            neighbors.append("".join(new_state))
    return neighbors


def dfs(start_state, goal_state):
```

```
    stack = [start_state]
    visited = set()
    parent = {start_state: None}

    while stack:
        current = stack.pop()
        if current == goal_state:
            path = []
            while current:
                path.append(current)
                current = parent[current]
            return path[::-1]
        if current not in visited:
            visited.add(current)
            neighbors = get_neighbors_dfs(current)
            neighbors.reverse()
            for neighbor in neighbors:
                if neighbor not in visited:
                    parent[neighbor] = current
                    stack.append(neighbor)
    return None


print("Enter the initial state (enter 3 digits per row, separated by spaces, 0 for empty):")
initial_state_rows = []
for i in range(3):
    row = input(f"Row {i+1}: ").split()
    initial_state_rows.extend(row)
initial_state = "".join(initial_state_rows)

print("\nEnter the goal state (enter 3 digits per row, separated by spaces, 0 for empty):")
goal_state_rows = []
for i in range(3):
    row = input(f"Row {i+1}: ").split()
    goal_state_rows.extend(row)
goal_state = "".join(goal_state_rows)

solution = dfs(initial_state, goal_state)

if solution:
    print("\nDFS solution path:")
    for s in solution:
        print(s[:3])
```

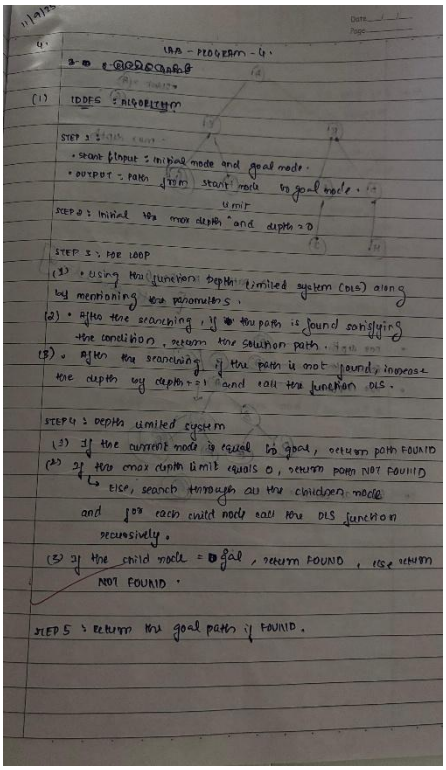
```
        print(s[3:6])
        print(s[6:])
        print()
else:
    print("\nNo solution found.")
```

Output:

```
Streaming output truncated to the last 5000 lines.
164
320
785

160
324
785

106
324
785

126
304
785

126
384
705

126
384
075

126
084
375

026
184
375

206
184
375

286
104
375

286
174
305
```

2.ITERATIVE DEEPENING

Algorithm:


AIML - PROGRAM - 4.

(1) IDDFS : ALGORITHM

STEP 1 :
• start input : initial node and goal node.
• OUTPUT : path from start node to goal node.

STEP 2 : initial the max depth and depth = 0

STEP 3 : FOR LOOP
(1) • using the function: depth limited system (DLS) along by mentioning the parameters.
(2) • After the searching, if the path is found satisfying the condition, return the solution path.
(3) • After the searching, if the path is not found, increase the depth by depth + 1 and call the function DLS.

STEP 4 : DEPTH LIMITED SYSTEM
(1) If the current node is equal to goal, return path FOUND
(2) If the max depth limit equals 0, return path NOT FOUND
↳ Else, search through all the children node and for each child node call the DLS function recursively.
(3) If the child node = goal, return FOUND, else return NOT FOUND.

STEP 5 : return the goal path if FOUND.

Code:

```
from copy import deepcopy

# Define the goal state
GOAL_STATE = [1, 2, 3, 4, 5, 6, 7, 8, 0]

# Generate all valid neighbor states by moving the blank tile
def get_neighbors(state):
    neighbors = []
    index = state.index(0)
    row, col = divmod(index, 3)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # up, down, left, right

    for dr, dc in directions:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_index = new_row * 3 + new_col
            new_state = state[:]
            new_state[index], new_state[new_index] = new_state[new_index], new_state[index]
```

```
            neighbors.append(new_state)

    return neighbors

# Depth-Limited Search
def dls(state, depth, visited):
    if state == GOAL_STATE:
        return [state]
    if depth == 0:
        return None
    visited.add(tuple(state))
    for neighbor in get_neighbors(state):
        if tuple(neighbor) not in visited:
            path = dls(neighbor, depth - 1, visited)
            if path:
                return [state] + path
    return None

# Iterative Deepening DFS
def iddfs(start_state, max_depth=20):
    for depth in range(max_depth):
        visited = set()
        path = dls(start_state, depth, visited)
        if path:
            return path
    return None

# Example start state
start = [1, 2, 3, 4, 0, 6, 7, 5, 8]
solution = iddfs(start)

# Print the solution path
if solution:
    print("Solution found in", len(solution) - 1, "moves:")
    for step in solution:
        print(step[:3])
        print(step[3:6])
        print(step[6:])
        print("-----")
else:
    print("No solution found within depth limit.")
```

Output:

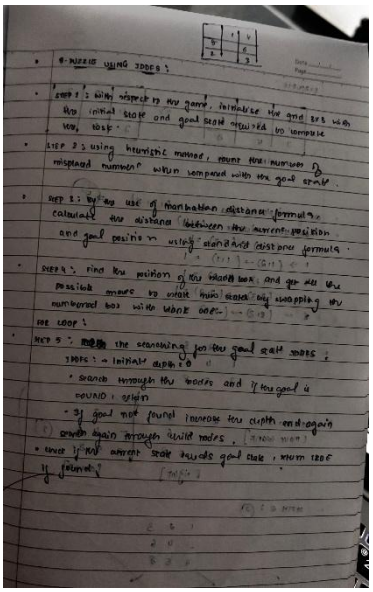
```
PS C:\Users\student\Desktop\Puzzle -  iddfs> & "C:\Program Files\Py
Solution found in 2 moves:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
-----
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
-----
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
-----
```

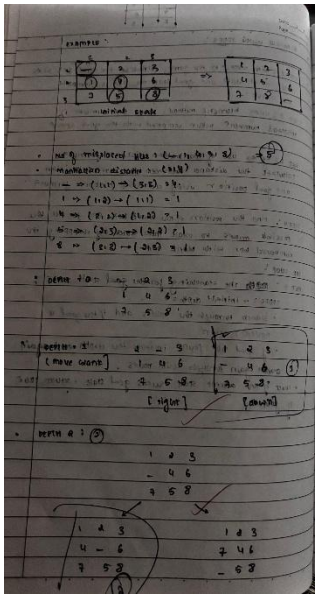
Program 3

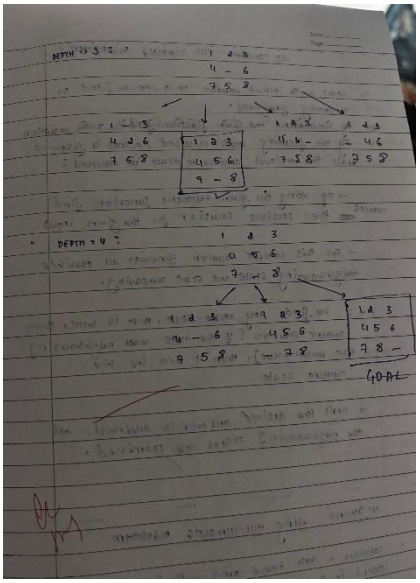
Implement A* search algorithm

1.A* SEARCH USING MANNHATTAN DISTANCE METHOD

Algorithm:







Code:

```
import heapq

class PuzzleState:
    def __init__(self, board, moves=0, previous=None):
        self.board = board
        self.moves = moves
        self.previous = previous
        self.size = 3

    def __eq__(self, other):
        return self.board == other.board

    def __hash__(self):
        return hash(self.board)

    def get_neighbors(self):
        neighbors = []
        zero_index = self.board.index(0)
```

```
        x, y = divmod(zero_index, self.size)
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < self.size and 0 <= new_y < self.size:
                new_zero_index = new_x * self.size + new_y
                new_board = list(self.board)
                new_board[zero_index], new_board[new_zero_index] = new_board[new_zero_index],
new_board[zero_index]
                neighbors.append(PuzzleState(tuple(new_board), self.moves + 1, self))
        return neighbors

    def misplaced_tiles(self, goal):
        count = 0
        for i in range(len(self.board)):
            if self.board[i] != 0 and self.board[i] != goal.board[i]:
                count += 1
        return count

    def __lt__(self, other):
        return False


def a_star(start, goal):
    open_set = []
    heapq.heappush(open_set, (start.misplaced_tiles(goal), start))
    g_score = {start: 0}
    f_score = {start: start.misplaced_tiles(goal)}
    closed_set = set()
    while open_set:
        current_f, current = heapq.heappop(open_set)
        if current == goal:
            path = []
            while current:
                path.append(current)
                current = current.previous
            path.reverse()
            return path
        closed_set.add(current)
        for neighbor in current.get_neighbors():
            if neighbor in closed_set:
                continue
            tentative_g = g_score[current] + 1
```

```
            if neighbor not in g_score or tentative_g < g_score[neighbor]:
                neighbor.previous = current
                g_score[neighbor] = tentative_g
                f = tentative_g + neighbor.misplaced_tiles(goal)
                f_score[neighbor] = f
                heapq.heappush(open_set, (f, neighbor))
    return None


def print_path(path):
    for state in path:
        for i in range(3):
            print(state.board[i*3:(i+1)*3])
        print()


def get_input_state(prompt):
    print(prompt)
    while True:
        try:
            values = list(map(int, input("Enter 9 numbers (0 for blank) separated by spaces:
").strip().split()))
            if len(values) != 9 or sorted(values) != list(range(9)):
                raise ValueError
            return tuple(values)
        except ValueError:
            print("Invalid input. Please enter numbers 0 to 8 without duplicates.")


start_board = get_input_state("Enter initial state:")
goal_board = get_input_state("Enter goal state:")

start_state = PuzzleState(start_board)
goal_state = PuzzleState(goal_board)

solution_path = a_star(start_state, goal_state)

if solution_path:
    print(f"Solution found in {len(solution_path)-1} moves:")
    print_path(solution_path)
else:
    print("No solution found.")
```

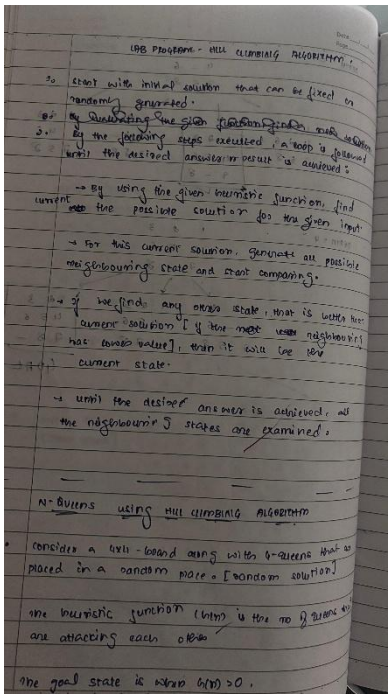
Output:

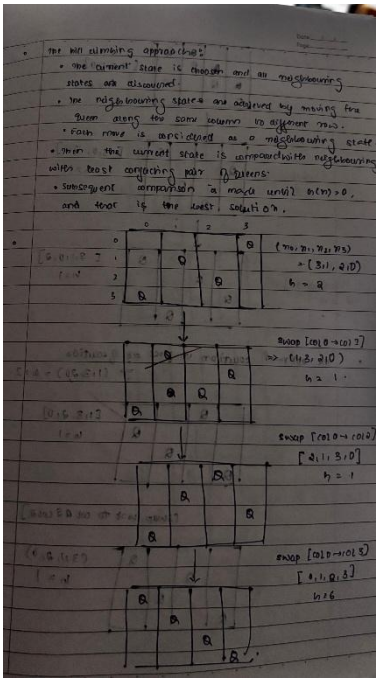
```
Enter initial state:
Enter 9 numbers (0 for blank) separated by spaces: 2 8 3 1 6 4 7 0 5
Enter goal state:
Enter 9 numbers (0 for blank) separated by spaces: 1 2 3 8 0 4 7 6 5
Solution found in 5 moves:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

(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)
```

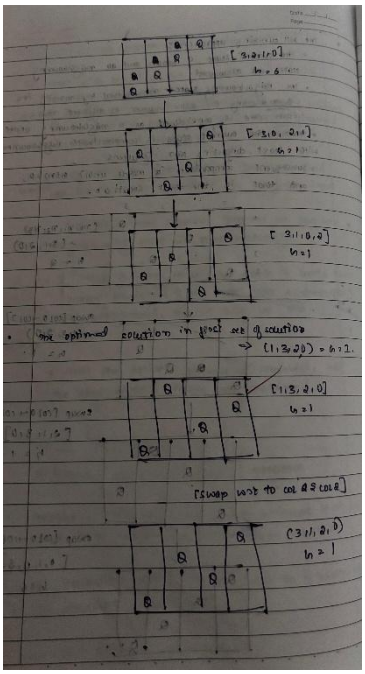
Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:







Code:

```
import random

# Count number of attacking pairs
def compute_attacks(state):
    attacks = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                attacks += 1
    return attacks

# Generate neighbors by moving one queen in its column
def get_neighbors(state):
    neighbors = []
    for col in range(len(state)):
        for row in range(len(state)):
            if state[col] != row:
```

```
            neighbor = state.copy()
            neighbor[col] = row
            neighbors.append(neighbor)
    return neighbors

# Hill climbing algorithm
def hill_climb():
    current = [random.randint(0, 3) for _ in range(4)]
    current_score = compute_attacks(current)

    while True:
        neighbors = get_neighbors(current)
        next_state = current
        next_score = current_score

        for neighbor in neighbors:
            score = compute_attacks(neighbor)
            if score < next_score:
                next_state = neighbor
                next_score = score

        if next_score == current_score:
            break  # No better neighbor found

        current = next_state
        current_score = next_score

    return current

# Convert solution to matrix
def print_board(state):
    size = len(state)
    board = [['.' for _ in range(size)] for _ in range(size)]
    for col, row in enumerate(state):
        board[row][col] = 'Q'
    for row in board:
        print(' '.join(row))

# Run the algorithm
solution = hill_climb()
print("4 Queens Solution in Matrix Form:")
print_board(solution)
```

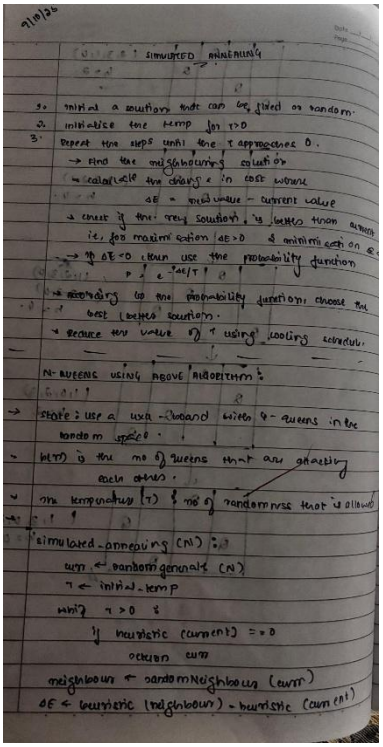
Output:

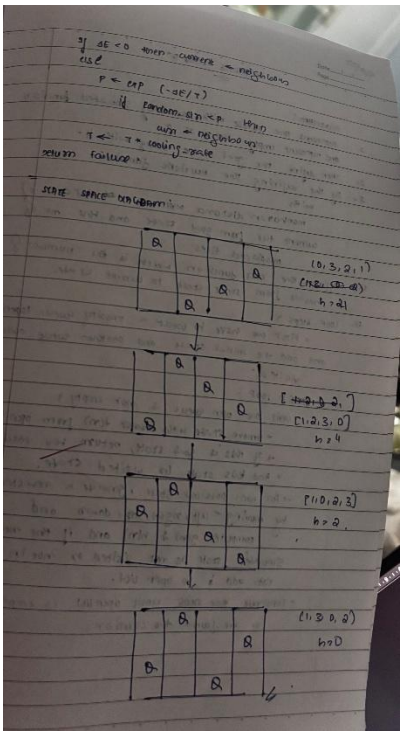
```
4 Queens Solution (Simulated Annealing):
. . Q .
Q . . .
. . . Q
. Q . .
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:





Code:

```
import random
import math

# Count attacking pairs
def compute_attacks(state):
    attacks = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                attacks += 1
    return attacks

# Generate a neighbor by moving one queen
def get_neighbor(state):
    neighbor = state.copy()
    col = random.randint(0, 3)
```

```
    new_row = random.randint(0, 3)
    while neighbor[col] == new_row:
        new_row = random.randint(0, 3)
    neighbor[col] = new_row
    return neighbor

# Simulated Annealing algorithm
def simulated_annealing():
    current = [random.randint(0, 3) for _ in range(4)]
    current_score = compute_attacks(current)
    temperature = 100.0
    cooling_rate = 0.95
    min_temp = 0.01

    while temperature > min_temp and current_score > 0:
        neighbor = get_neighbor(current)
        neighbor_score = compute_attacks(neighbor)
        delta = neighbor_score - current_score

        if delta < 0 or random.random() < math.exp(-delta / temperature):
            current = neighbor
            current_score = neighbor_score

        temperature *= cooling_rate

    return current

# Print board matrix
def print_board(state):
    board = [['.' for _ in range(4)] for _ in range(4)]
    for col, row in enumerate(state):
        board[row][col] = 'Q'
    for row in board:
        print(' '.join(row))

# Run the algorithm
solution = simulated_annealing()
print("4 Queens Solution (Simulated Annealing):")
print_board(solution)
```

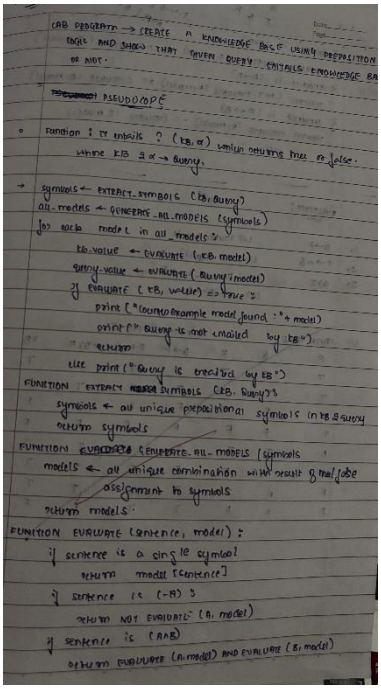
Output:

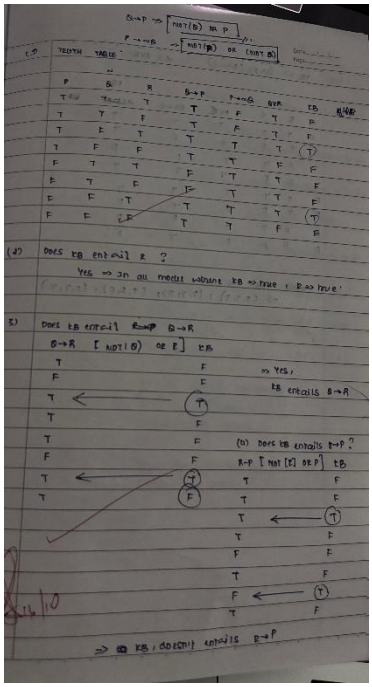
```
4 Queens Solution (Simulated Annealing):
. . Q .
Q . . .
. . . Q
. Q . .
```

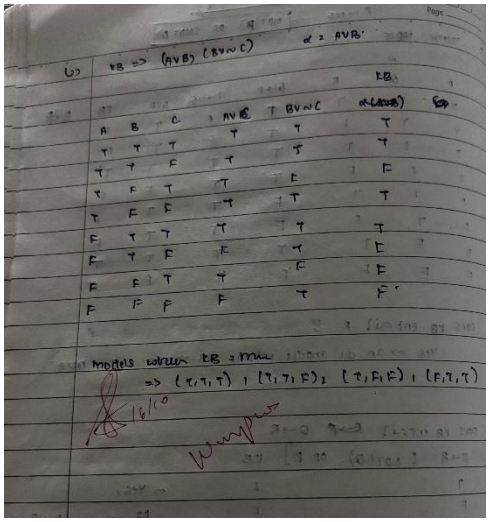
Program 6

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

Algorithm:







Code:

```
import itertools

values = [True, False]

def implies(a, b):
    return (not a) or b

def KB(P, Q, R):
    return implies(Q, P) and implies(P, not Q) and (Q or R)

def query_R(P, Q, R):
    return R

def query_R_implies_P(P, Q, R):
    return implies(R, P)
```

```
def query_Q_implies_R(P, Q, R):
    return implies(Q, R)

print(f"{'P':^3} {'Q':^3} {'R':^3} {'Q→P':^5} {'P→¬Q':^6} {'QvR':^5} {'KB':^4} {'R':^3}
{'R→P':^6} {'Q→R':^6}")
print("-" * 60)

kb_models = []
for P, Q, R in itertools.product(values, repeat=3):
    q1 = implies(Q, P)
    q2 = implies(P, not Q)
    q3 = Q or R
    kb_true = q1 and q2 and q3
    r_val = R
    r_imp_p = implies(R, P)
    q_imp_r = implies(Q, R)

    print(f"{P!s:^3} {Q!s:^3} {R!s:^3} {q1!s:^5} {q2!s:^6} {q3!s:^5} {kb_true!s:^4} {r_val!s:^3}
{r_imp_p!s:^6} {q_imp_r!s:^6}")

    if kb_true:
        kb_models.append((P, Q, R))
def entails(query):
    for (P, Q, R) in kb_models:
        if not query(P, Q, R):
            return False
    return True

print("\nModels where KB is True:", kb_models)
print("\nEntailment Results:")
print("Does KB entail R?        ->", entails(query_R))
print("Does KB entail (R → P)?  ->", entails(query_R_implies_P))
print("Does KB entail (Q → R)?  ->", entails(query_Q_implies_R))
```

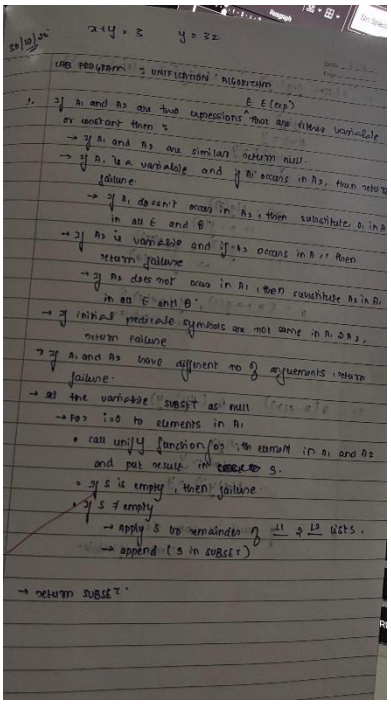
Output:

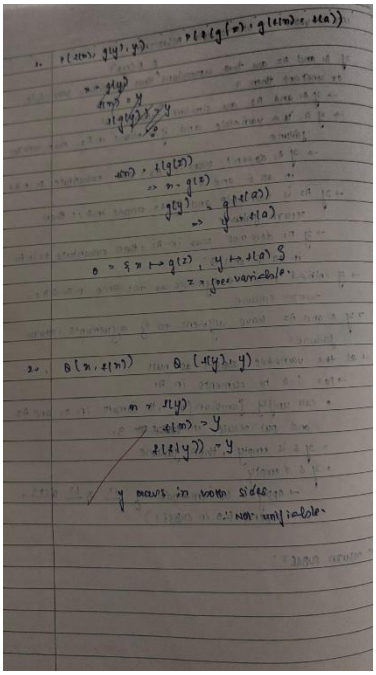
```
PS C:\Users\BMSCECSE\Documents\ISTG\AI> python -u "c:\Users\BMSCECSE\Documents\ISTG\AI\code.py"
 P    Q    R    Q→P    P→¬Q    QvR    KB    R    R→P    Q→R
-------------------------------------------------------------
True True True True  False  True  False True  True   True
True True False True  False  True  False False  True  False
True False True True   True  True  True True  True   True
True False False True   True  False False False  True   True
False True True False  True  True  False True False   True
False True False False  True  True  False False  True  False
False False True True   True  True  True True False   True
False False False True   True  False False False  True   True

Models where KB is True: [(True, False, True), (False, False, True)]

Entailment Results:
Does KB entail R?        -> True
Does KB entail (R → P)?  -> False
Does KB entail (Q → R)?  -> True
PS C:\Users\BMSCECSE\Documents\ISTG\AI>
```

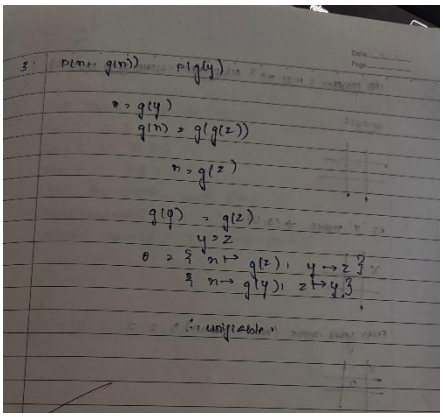
Program 7

Implement unification in first order logic

Algorithm:







Code:

```
def unify(psi1, psi2):
    if is_variable_or_constant(psi1) or is_variable_or_constant(psi2):
        if psi1 == psi2:
            return {}
        elif is_variable(psi1):
            if occurs_in(psi1, psi2):
                return "FAILURE"
            else:
                return {psi1: psi2}
        elif is_variable(psi2):
            if occurs_in(psi2, psi1):
                return "FAILURE"
            else:
                return {psi2: psi1}
        else:
            return "FAILURE"
```

```
    if predicate_symbol(psi1) != predicate_symbol(psi2):
        return "FAILURE"

    if len(psi1['args']) != len(psi2['args']):
        return "FAILURE"

    SUBST = {}

    for i in range(len(psi1['args'])):
        S = unify(psi1['args'][i], psi2['args'][i])
        if S == "FAILURE":
            return "FAILURE"
        if S:
            psi1 = apply_substitution(S, psi1)
            psi2 = apply_substitution(S, psi2)
            SUBST.update(S)

    return SUBST


def is_variable_or_constant(x):
    return isinstance(x, str) and (x.islower() or x.isalpha())


def is_variable(x):
    return isinstance(x, str) and x.islower()


def occurs_in(var, expr):
    if var == expr:
        return True
    if isinstance(expr, dict):
        return any(occurs_in(var, arg) for arg in expr.get('args', []))
    return False


def predicate_symbol(expr):
    if isinstance(expr, dict) and 'pred' in expr:
        return expr['pred']
    return None


def apply_substitution(subst, expr):
    if isinstance(expr, str):
        return subst.get(expr, expr)
    elif isinstance(expr, dict):
```

```
        return {
            'pred': expr['pred'],
            'args': [apply_substitution(subst, arg) for arg in expr.get('args', [])]
        }
    return expr


# Example usage
psi1 = {'pred': 'P', 'args': ['x', 'y']}
psi2 = {'pred': 'P', 'args': ['a', 'b']}
result = unify(psi1, psi2)
print(result)
```

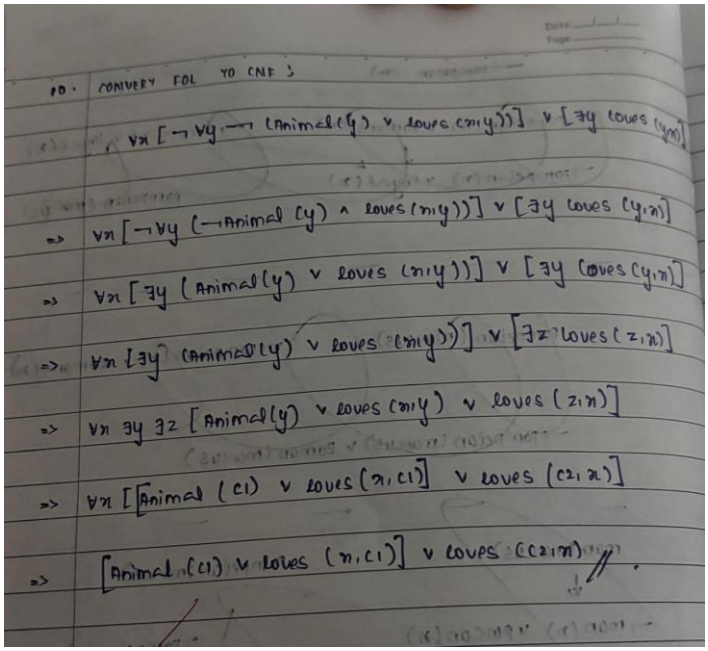
Output:

```
{'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:



Code:

```
import itertools

def pred(name, *args):
    return ('pred', name, *args)

def forall(var, f):
    return ('forall', var, f)

def exists(var, f):
    return ('exists', var, f)

def neg(f):
    return ('not', f)

def conj(*args):
    return ('and',) + args

def disj(*args):
    return ('or',) + args

def var(v):
    return ('var', v)
```

```
def pretty(f):
    t = f[0]
    if t == 'pred':
        return "{}({})".format(f[1], ", ".join(f[2:]))
    if t == 'not':
        return "¬" + "(" + pretty(f[1]) + ")"
    if t == 'and' or t == 'or':
        op = " ∧ " if t == 'and' else " ∨ "
        return "(" + op.join(pretty(x) for x in f[1:]) + ")"
    if t == 'forall':
        return "∀{} {}".format(f[1], pretty(f[2]))
    if t == 'exists':
        return "∃{} {}".format(f[1], pretty(f[2]))
    return str(f)


A = forall('x',
           disj(
               neg(forall('y', neg(disj(pred('Animal','y'), pred('Loves','x','y'))))),
               exists('y', pred('Loves','y','x'))
           )
)

print("Original formula:")
print(pretty(A))
print()


def eliminate_implications(f):

    t = f[0]
    if t in ('pred',):
        return f
    if t == 'not':
        return ('not', eliminate_implications(f[1]))
    if t in ('and','or'):
        return (t,) + tuple(eliminate_implications(x) for x in f[1:])
    if t in ('forall','exists'):
        return (t, f[1], eliminate_implications(f[2]))
    return f

B = eliminate_implications(A)
print("After eliminating implications (no change here):")
print(pretty(B))
print("-------------------------------------------------------------")
print()
```

```
def push_negation(f):
    t = f[0]
    if t == 'not':
        inner = f[1]
        it = inner[0]
        if it == 'not':
            return push_negation(inner[1])
        if it == 'and':
            return disj(*[push_negation(neg(s)) for s in inner[1:]])
        if it == 'or':
            return conj(*[push_negation(neg(s)) for s in inner[1:]])
        if it == 'forall':
            # ¬∀y P  =>  ∃y ¬P
            return exists(inner[1], push_negation(neg(inner[2])))
        if it == 'exists':
            # ¬∃y P  =>  ∀y ¬P
            return forall(inner[1], push_negation(neg(inner[2])))
        return ('not', push_negation(inner))
    elif t in ('and','or'):
        return (t,) + tuple(push_negation(x) for x in f[1:])
    elif t in ('forall','exists'):
        return (t, f[1], push_negation(f[2]))
    else:
        return f

C = push_negation(B)
print("After pushing negations inward :")
print(pretty(C))
print("-----------------------------------------------------------")
print()


def flatten(f):
    t = f[0]
    if t in ('and','or'):
        items = []
        for x in f[1:]:
            y = flatten(x)
            if y[0] == t:
                items.extend(y[1:])
            else:
                items.append(y)
        return (t,)+tuple(items)
    if t in ('not',):
        return ('not', flatten(f[1]))
```

```
    if t in ('forall','exists'):
        return (t, f[1], flatten(f[2]))
    return f

D = flatten(C)
print("After simplifying:")
print(pretty(D))
print("------------------------------------------------------------")
print()


counter = itertools.count()
def standardize_apart(f, env=None):
    if env is None: env = {}
    t = f[0]
    if t == 'pred':
        return ('pred', f[1]) + tuple(env.get(arg, arg) for arg in f[2:])
    if t in ('forall','exists'):
        oldv = f[1]
        newv = f"{oldv}_{next(counter)}"
        env2 = env.copy()
        env2[oldv] = newv
        return (t, newv, standardize_apart(f[2], env2))
    if t in ('and','or'):
        return (t,) + tuple(standardize_apart(x, env) for x in f[1:])
    if t == 'not':
        return ('not', standardize_apart(f[1], env))
    return f

E = standardize_apart(D)
print("After standardizing variables apart:")
print(pretty(E))
print("------------------------------------------------------------")
print()


skolem_count = itertools.count()
def skolemize(f, universal_vars=None):
    if universal_vars is None: universal_vars = []
    t = f[0]
    if t == 'forall':
        return forall(f[1], skolemize(f[2], universal_vars + [f[1]]))
    if t == 'exists':
        varname = f[1]

        sk_name = f"sk_{next(skolem_count)}"
        if universal_vars:
```

```
            args = "_".join(universal_vars)
            func = f"{sk_name}({args})"
        else:
            func = f"{sk_name}()"

        body = substitute_var(f[2], varname, func)
        return skolemize(body, universal_vars)
    if t in ('and','or'):
        return (t,) + tuple(skolemize(x, universal_vars) for x in f[1:])
    if t == 'not':
        return ('not', skolemize(f[1], universal_vars))
    if t == 'pred':
        return f
    return f

def substitute_var(f, varname, term):
    t = f[0]
    if t == 'pred':
        return ('pred', f[1]) + tuple((term if a == varname else a) for a in f[2:])
    if t in ('and','or'):
        return (t,) + tuple(substitute_var(x, varname, term) for x in f[1:])
    if t in ('forall','exists'):

        if f[1] == varname:
            return f
        return (t, f[1], substitute_var(f[2], varname, term))
    if t == 'not':
        return ('not', substitute_var(f[1], varname, term))
    return f

F = skolemize(E)
print("After Skolemization :")
print(pretty(F))
print("------------------------------------------------------------")
print()


def drop_universal(f):
    t = f[0]
    if t == 'forall':
        return drop_universal(f[2])
    if t in ('and','or'):
        return (t,) + tuple(drop_universal(x) for x in f[1:])
    if t == 'not':
        return ('not', drop_universal(f[1]))
    return f
```

```
G = drop_universal(F)
print("After dropping universal quantifiers :")
print(pretty(G))
print("------------------------------------------------------------")
print()


def to_cnf(f):

    if f[0] == 'and':
        return ('and',) + tuple(to_cnf(x) for x in f[1:])
    if f[0] == 'or':

        items = [to_cnf(x) for x in f[1:]]
        return distribute_or(items)
    if f[0] == 'not' or f[0] == 'pred':
        return f
    return f

def distribute_or(items):

    if not items:
        return ('or',)
    result = items[0]
    for nxt in items[1:]:
        result = distribute_two(result, nxt)
    return result

def distribute_two(a, b):

    if a[0] == 'and':
        return ('and',) + tuple(distribute_two(x, b) for x in a[1:])
    if b[0] == 'and':
        return ('and',) + tuple(distribute_two(a, x) for x in b[1:])

    a_items = (a[1:] if a[0] == 'or' else (a,))
    b_items = (b[1:] if b[0] == 'or' else (b,))
    return ('or',) + tuple(x for x in list(a_items) + list(b_items))

H = to_cnf(G)
print("After distributing OR over AND (CNF):")
print(pretty(H))
print("------------------------------------------------------------")
print()


def clauses_of_cnf(f):
```

```
    if f[0] == 'and':
        clauses = []
        for x in f[1:]:
            clauses.extend(clauses_of_cnf(x))
        return clauses
    if f[0] == 'or':

        lits = []
        for lit in f[1:]:
            lits.append(lit)
        return [lits]

    return [[f]]

clauses = clauses_of_cnf(H)
print("Final CNF as clause list:")
for i, cl in enumerate(clauses, 1):
    print(f"Clause {i}: " + " ∨ ".join(pretty(l) for l in cl))
print()
```

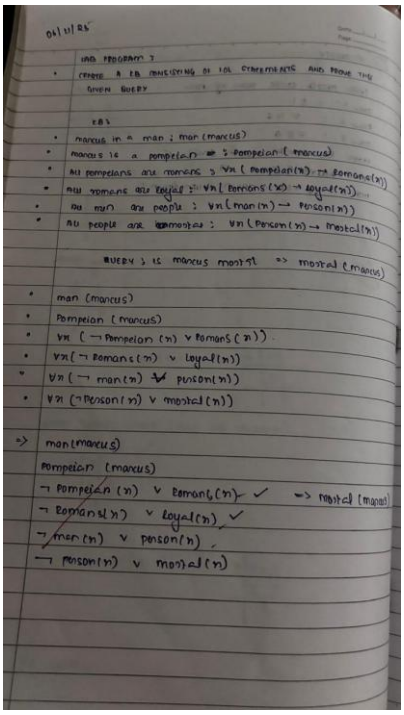
Output:

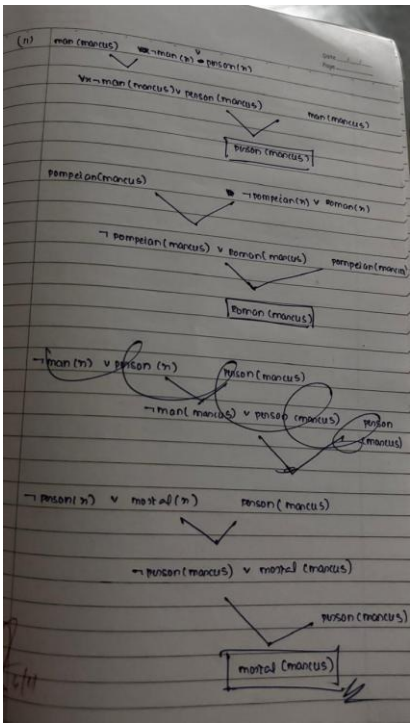
```
PS C:\Users\BMSCECSE\Documents\ISTG\AI\FOL to CNF> python -u "c:\Users\BMSCECSE\Documents\ISTG\AI\FOL to CNF\code.py"
Original formula:
∀x (¬(∀y ¬((Animal(y) ∨ Loves(x, y)))) ∨ ∃y Loves(y, x))

After eliminating implications (no change here):
∀x (¬(∀y ¬((Animal(y) ∨ Loves(x, y)))) ∨ ∃y Loves(y, x))
------------------------------------------------------------

After pushing negations inward :
∀x (∃y (Animal(y) ∨ Loves(x, y)) ∨ ∃y Loves(y, x))
------------------------------------------------------------

After simplifying:
∀x (∃y (Animal(y) ∨ Loves(x, y)) ∨ ∃y Loves(y, x))
------------------------------------------------------------

After standardizing variables apart:
∀x_0 (∃y_1 (Animal(y_1) ∨ Loves(x_0, y_1)) ∨ ∃y_2 Loves(y_2, x_0))
------------------------------------------------------------

After Skolemization :
∀x_0 ((Animal(sk_0(x_0)) ∨ Loves(x_0, sk_0(x_0))) ∨ Loves(sk_1(x_0), x_0))
------------------------------------------------------------

After dropping universal quantifiers :
((Animal(sk_0(x_0)) ∨ Loves(x_0, sk_0(x_0))) ∨ Loves(sk_1(x_0), x_0))
------------------------------------------------------------

After distributing OR over AND (CNF):
(Animal(sk_0(x_0)) ∨ Loves(x_0, sk_0(x_0)) ∨ Loves(sk_1(x_0), x_0))
------------------------------------------------------------

Final CNF as clause list:
Clause 1: Animal(sk_0(x_0)) ∨ Loves(x_0, sk_0(x_0)) ∨ Loves(sk_1(x_0), x_0)

PS C:\Users\BMSCECSE\Documents\ISTG\AI\FOL to CNF>
```

Program 9

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

Algorithm:





Code:

```
facts = {
   "Man(Marcus)",
   "Pompeian(Marcus)"
}

rules = [
   ("Pompeian(x) -> Roman(x)"),
   ("Roman(x) -> Loyal(x)"),
   ("Man(x) -> Person(x)"),
   ("Person(x) -> Mortal(x)")
]

def substitute(statement, var, const):
   return statement.replace(f"({var})", f"({const})").replace(f"{var},", f"{const},").replace(f",{var}",
f",{const}")
```

```
def forward_chain(facts, rules):
    step = 1
    new_fact_added = True
    while new_fact_added:
        new_fact_added = False
        print(f"\n--- Step {step} ---")
        print("Current Facts:", ", ".join(sorted(facts)))
        for rule in rules:
            premise, conclusion = rule.split("->")
            premise = premise.strip()
            conclusion = conclusion.strip()
            var = 'x'
            for fact in list(facts):
                if premise.startswith(fact.split("(")[0]):
                    const = fact[fact.find("(")+1:fact.find(")")]
                    new_fact = substitute(conclusion, var, const)
                    if new_fact not in facts:
                        print(f"Applied Rule: {rule}")
                        print(f"Matched Fact: {fact}")
                        print(f"Inferred: {new_fact}")
                        print("---------------------------------")
                        facts.add(new_fact)
                        new_fact_added = True
        step += 1
    return facts

print("=== Forward Reasoning Process ===")
derived_facts = forward_chain(facts.copy(), rules)

query = "Mortal(Marcus)"
print("\n=== Final Result ===")
if query in derived_facts:
    print(f" Query '{query}' is PROVED TRUE.")
else:
    print(f" Query '{query}' cannot be proved from the Knowledge Base.")
```

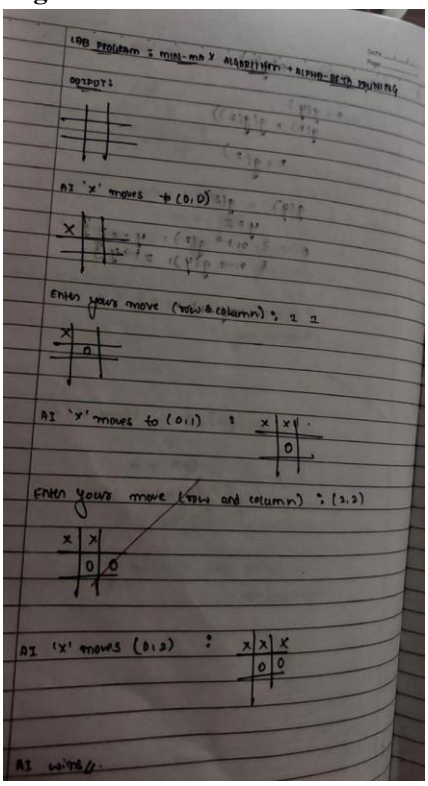
Output:

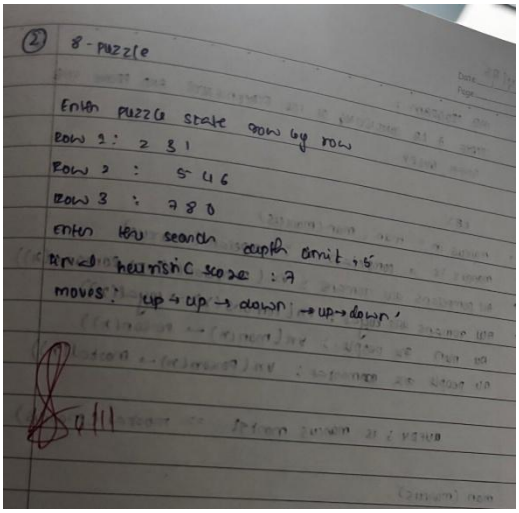
```
PS C:\Users\BMSCECSE\Documents\ISTG\AI\Forward chaining> python -u "c:\Users\BMSCECSE\Documents\ISTG\AI\Forward chaining\code.py"
=== Forward Reasoning Process ===

--- Step 1 ---
Current Facts: Man(Marcus), Pompeian(Marcus)
Applied Rule: Pompeian(x) -> Roman(x)
Matched Fact: Pompeian(Marcus)
Inferred: Roman(Marcus)
---------------------------------
Applied Rule: Roman(x) -> Loyal(x)
Matched Fact: Roman(Marcus)
Inferred: Loyal(Marcus)
---------------------------------
Applied Rule: Man(x) -> Person(x)
Matched Fact: Man(Marcus)
Inferred: Person(Marcus)
---------------------------------
Applied Rule: Person(x) -> Mortal(x)
Matched Fact: Person(Marcus)
Inferred: Mortal(Marcus)
---------------------------------

--- Step 2 ---
Current Facts: Loyal(Marcus), Man(Marcus), Mortal(Marcus), Person(Marcus), Pompeian(Marcus), Roman(Marcus)

=== Final Result ===
 Query 'Mortal(Marcus)' is PROVED TRUE.
```

Program 10

Implement Alpha-Beta Pruning.

Algorithm:





Code:

```
import math

PLAYER = 'X'
OPPONENT = 'O'

def check_winner(board):
    win_combos = [
        [0,1,2], [3,4,5], [6,7,8],
        [0,3,6], [1,4,7], [2,5,8],
        [0,4,8], [2,4,6]
    ]
    for combo in win_combos:
        if board[combo[0]] == board[combo[1]] == board[combo[2]] != ' ':
            return board[combo[0]]
    return None

def is_full(board):
    return ' ' not in board
```

```
def evaluate(board):
    winner = check_winner(board)
    if winner == PLAYER:
        return 10
    elif winner == OPPONENT:
        return -10
    else:
        return 0

def get_children(board, player):
    children = []
    for i in range(9):
        if board[i] == ' ':
            new_board = board[:]
            new_board[i] = player
            children.append((new_board, i))
    return children

def alpha_beta(board, depth, alpha, beta, maximizing):
    score = evaluate(board)
    if depth == 0 or score in [10, -10] or is_full(board):
        return score

    if maximizing:
        max_eval = -math.inf
        for child, _ in get_children(board, PLAYER):
            eval = alpha_beta(child, depth - 1, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval
    else:
        min_eval = math.inf
        for child, _ in get_children(board, OPPONENT):
            eval = alpha_beta(child, depth - 1, alpha, beta, True)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return min_eval

def find_best_move(board):
    best_val = -math.inf
    best_move = -1
    for child, move in get_children(board, PLAYER):
        move_val = alpha_beta(child, depth=5, alpha=-math.inf, beta=math.inf, maximizing=False)
```

```
            if move_val > best_val:
                best_val = move_val
                best_move = move
        return best_move


if __name__ == "__main__":
    board = [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
    while not is_full(board) and not check_winner(board):
        print("\nCurrent board:")
        print(f"{board[0]} | {board[1]} | {board[2]}")
        print(f"{board[3]} | {board[4]} | {board[5]}")
        print(f"{board[6]} | {board[7]} | {board[8]}")

        move = find_best_move(board)
        board[move] = PLAYER
        print(f"\nAI plays at position {move}")

        if check_winner(board) or is_full(board):
            break

        user_move = int(input("Your move (0-8): "))
        if board[user_move] == ' ':
            board[user_move] = OPPONENT
        else:
            print("Invalid move. Try again.")

    print("\nFinal board:")
    print(f"{board[0]} | {board[1]} | {board[2]}")
    print(f"{board[3]} | {board[4]} | {board[5]}")
    print(f"{board[6]} | {board[7]} | {board[8]}")
    winner = check_winner(board)
    if winner:
        print(f"Winner: {winner}")
    else:
        print("It's a draw!")
```

Output:

```
Current board:
  |   |
  |   |
  |   |

AI plays at position 0
Your move (0-8): 4

Current board:
X |   |
  | O |
  |   |

AI plays at position 1
Your move (0-8): 2

Current board:
X | X | O
  | O |
  |   |

AI plays at position 6
Your move (0-8): 3

Current board:
X | X | O
O | O |
X |   |

AI plays at position 5
Your move (0-8): 7

Current board:
X | X | O
O | O | X
X | O |

AI plays at position 8

Final board:
X | X | O
O | O | X
X | O | X
It's a draw!

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