VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

RAHUL N RAJU(1BM22CS215)

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
Sep-2024 to Jan-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 **RAHUL N RAJU (1BM22CS215),** 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.

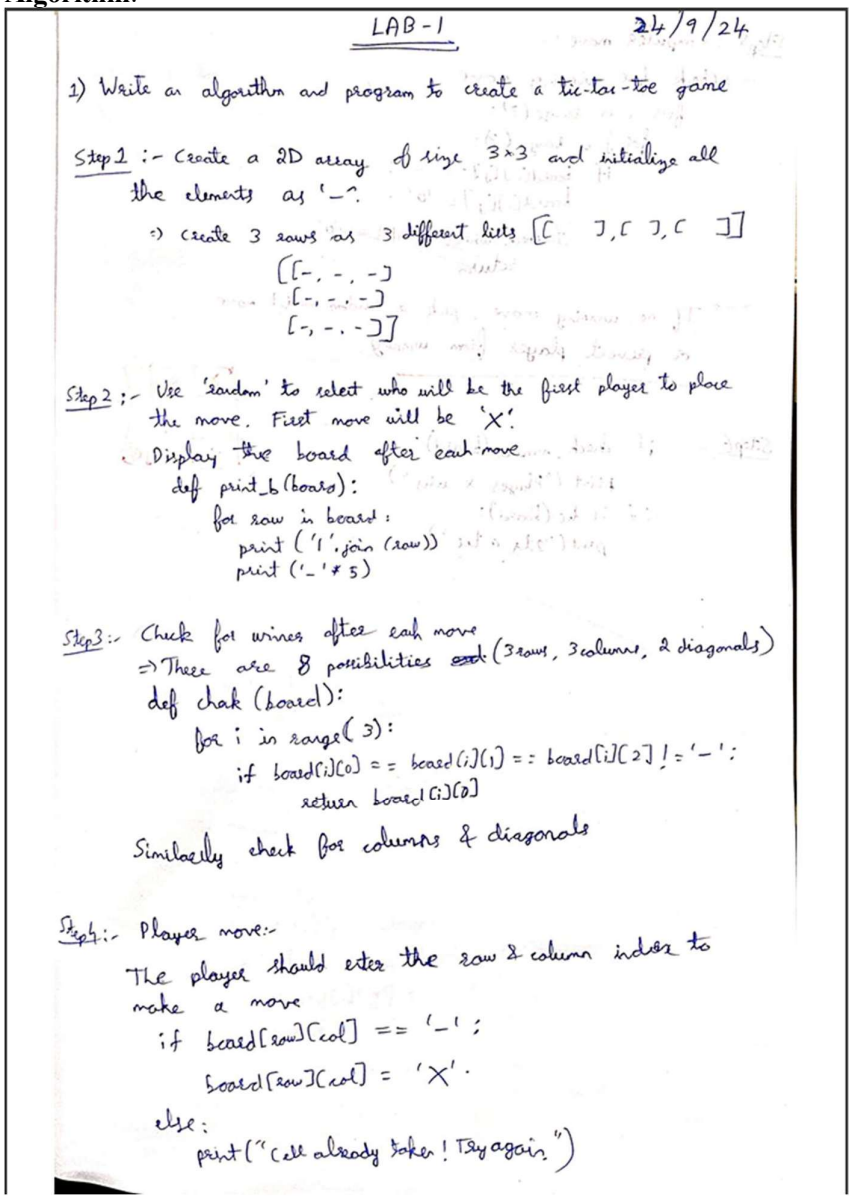
Radhika A D Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link : https://github.com/RahulCS215/Artificial-Intelligence-lab

LAB 1: Tic –Tac –Toe Game

Algorithm:



LAB-1 24/9/24

1) Write an algorithm and program to create a tic-tac-toe game

Step 1 :- Create a 2D array of size 3×3 and initialize all the elements as '-'.

⇒ Create 3 rows as 3 different lists [[ ], [ ], [ ]]

[[-, -, -]
[-, -, -]
[-, -, -]]

Step 2 :- Use 'random' to select who will be the first player to place the move. First move will be 'X'.

Display the board after each move.

```
def print_b(board):
    for row in board:
        print ('|'.join(row))
        print ('-' * 5)
```

Step 3 :- Check for wines after each move

⇒ There are 8 possibilities (3 rows, 3 columns, 2 diagonals)

```
def chak (board):
    for i in range( 3):
        if board[i][0] == board[i][1] == board[i][2] != '-':
            return board[i][0]
```

Similarly check for columns & diagonals

Step 4 :- Player move:-

The player should ester the row & column index to make a move

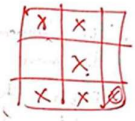
```
if board[row][col] == '-';
    board[row][col] = 'X'.
else:
    print("Cell already taken ! Try again")
```

Step 5 :- Computer move :-

→ Check for winning move

```
for i in range(3):
    for j in range(3):
        if board[i][j] == '_':
            board[i][j] = 'O'
            if check_winner(board) == 'O':
                return
```

→ If no winning move, pick a random valid move & prevent player from winning



Step 6 :-

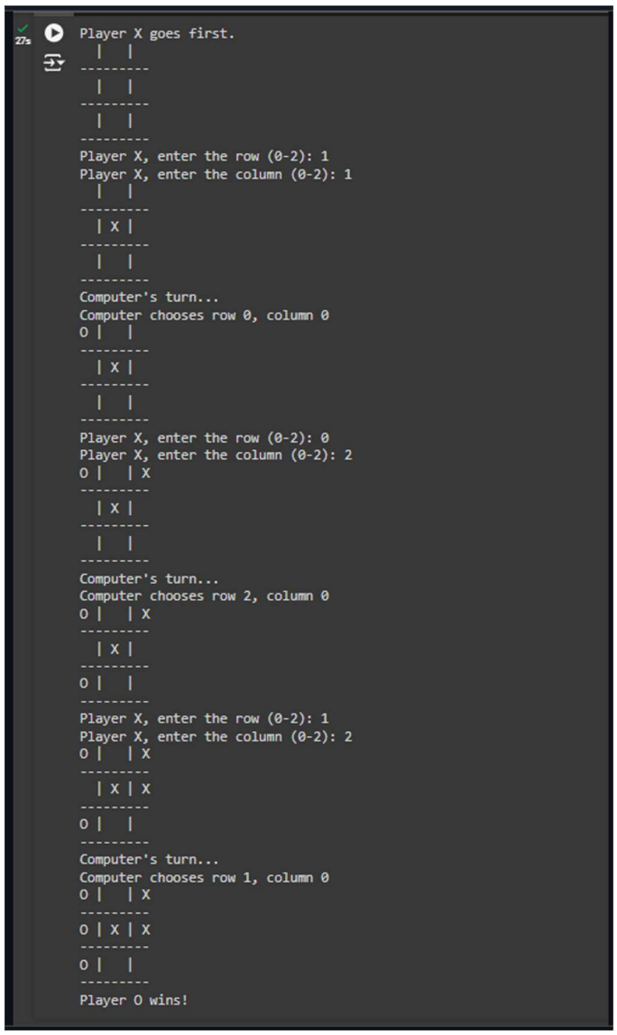
```
if check_winner(board):
    print("Player X wins")
if is_tie(board):
    print("It's a tie")
```

Code:

```
import random

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

def check_winner(board):
    # Check rows, columns, and diagonals for a winner
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != " ":
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != " ":
            return board[0][i]

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

    return None

def is_full(board):
    return all(cell != " " for row in board for cell in row)

def find_winning_move(board, player):
    for i in range(3):
        for j in range(3):
            if board[i][j] == " ":
                board[i][j] = player
                if check_winner(board) == player:
                    board[i][j] = " "  # Undo move
                    return (i, j)
                board[i][j] = " "  # Undo move
    return None

def get_computer_move(board):
    # Check for winning move
    move = find_winning_move(board, "O")
    if move:
        return move

    # Check for blocking move
    move = find_winning_move(board, "X")
    if move:
        return move

    # Take the center if available
    if board[1][1] == " ":
        return (1, 1)

    # Take a corner if available
    corners = [(0, 0), (0, 2), (2, 0), (2, 2)]
    for corner in corners:
        if board[corner[0]][corner[1]] == " ":
            return corner
```

```
    # Take any remaining space
    for i in range(3):
        for j in range(3):
            if board[i][j] == " ":
                return (i, j)

def tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = "X"  # X is the human player
    computer_player = "O"

    print("Player X goes first.")

    while True:
        print_board(board)

        if current_player == "X":
            while True:
                try:
                    row = int(input("Player X, enter the row (0-2): "))
                    col = int(input("Player X, enter the column (0-2): "))
                    if board[row][col] == " ":
                        break
                    else:
                        print("Cell is already taken! Try again.")
                except (ValueError, IndexError):
                    print("Invalid input! Please enter numbers between 0 and 2.")
        else:
            print("Computer's turn...")
            row, col = get_computer_move(board)
            print(f"Computer chooses row {row}, column {col}")

        board[row][col] = current_player

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

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

        current_player = computer_player if current_player == "X" else "X"

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

OUTPUT:


Player X goes first.
| |

| |

| |

Player X, enter the row (0-2): 1
Player X, enter the column (0-2): 1
| |

| X |

| |

Computer's turn...
Computer chooses row 0, column 0
O | |

| X |

| |

Player X, enter the row (0-2): 0
Player X, enter the column (0-2): 2
O | | X

| X |

| |

Computer's turn...
Computer chooses row 2, column 0
O | | X

| X |

O | |

Player X, enter the row (0-2): 1
Player X, enter the column (0-2): 2
O | | X

| X | X

O | |

Computer's turn...
Computer chooses row 1, column 0
O | | X

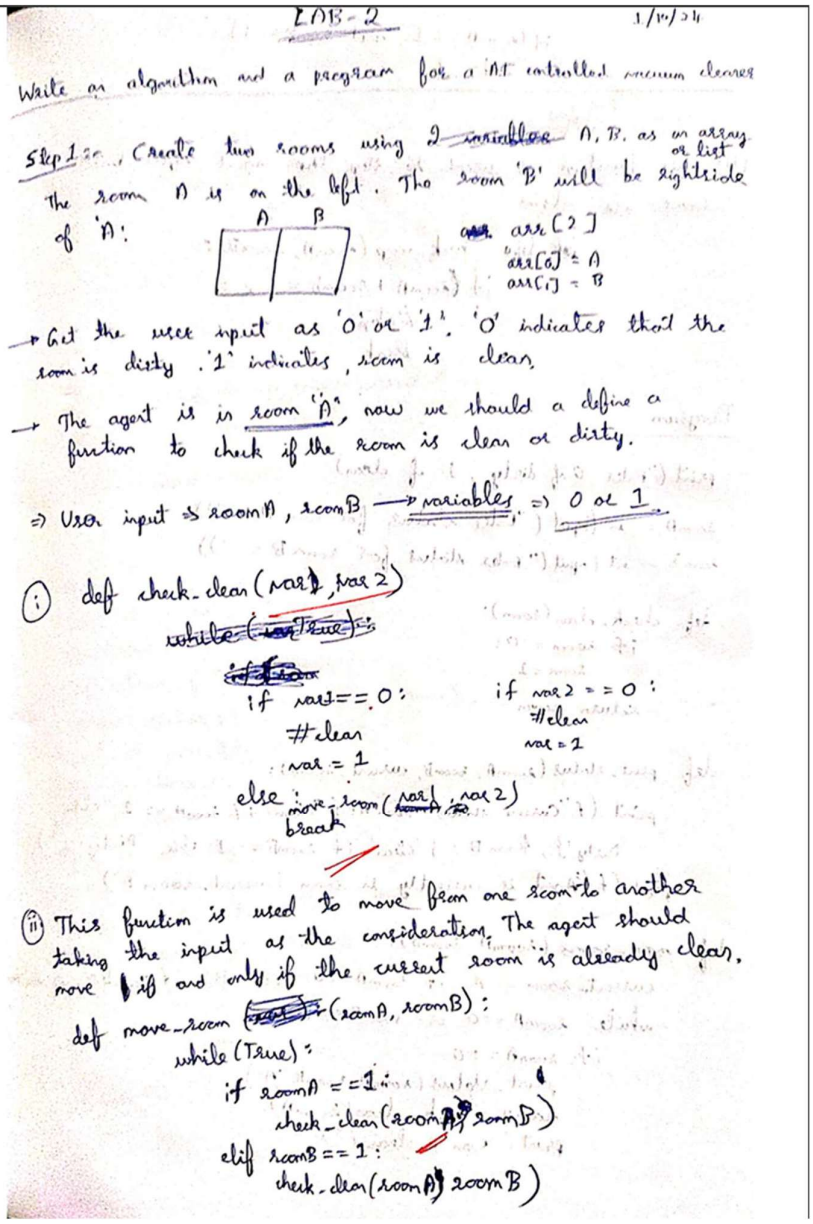
O | X | X

O | |

Player O wins!

LAB 2: Vacuum World

Algorithm:



LAB - 2

Write an algorithm and a program for a AI controlled vacuum cleaner

Step 1 :- Create two rooms using 2 variables A, B. as an array or list

The room A is on the left. The room 'B' will be rightside of 'A':

arr[ ]
arr[0] = A
arr[1] = B

→ Get the user input as '0' or '1'. '0' indicates that the room is dirty. '1' indicates, room is clean.

→ The agent is in room 'A', now we should a define a function to check if the room is clean or dirty.

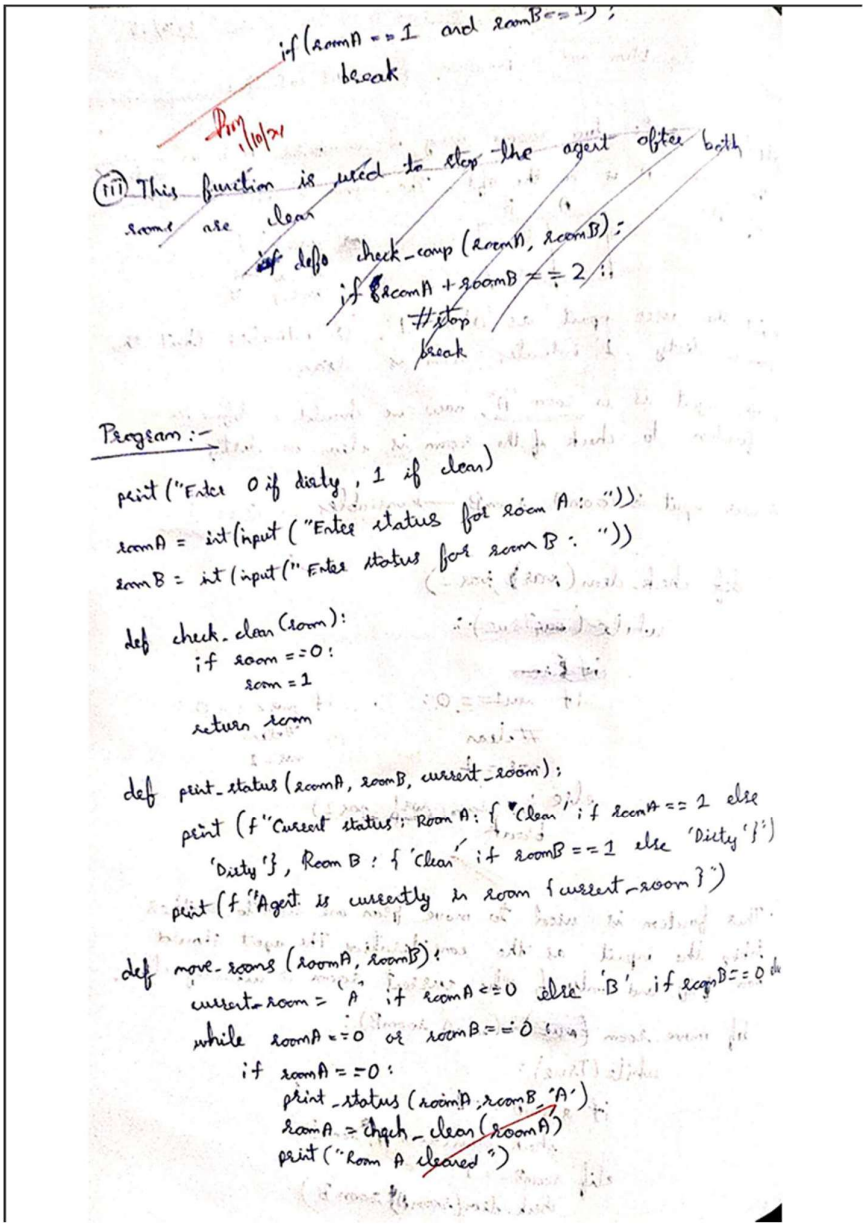
⇒ User input ⇒ roomA, roomB → variables ⇒ 0 or 1

(i) def check_clean(var1, var2)

```
while (True):
    if var1 == 0:
        #clean
        var = 1
    else: move_room(var1, var2)
          break

if var2 == 0:
    #clean
    var = 1
```

(ii) This function is used to move from one room to another taking the input as the consideration. The agent should move if and only if the current room is already clean.

```
def move_room (roomA, roomB):
    while (True):
        if roomA == 1:
            check_clean(roomA, roomB)
        elif roomB == 1:
            check_clean(roomA, roomB)
```



if (roomA == 1 and roomB == 1):
    break

Prog 1/10/24

(iii) This function is used to stop the agent after both rooms are clean

def check_comp (roomA, roomB):
    if roomA + roomB == 2:
        # stop
        break

Program :-

print ("Enter 0 if dirty, 1 if clean")
roomA = int (input ("Enter status for room A : "))
roomB = int (input ("Enter status for room B : "))

def check_clean (room):
    if room == 0:
        room = 1
    return room

def print_status (roomA, roomB, current_room):
    print (f"Current status: Room A: {'Clean' if roomA == 1 else 'Dirty'}, Room B : {'Clean' if roomB == 1 else 'Dirty'}")
    print (f"Agent is currently in room {current_room}")

def move_rooms (roomA, roomB):
    current_room = 'A' if roomA == 0 else 'B' if roomB == 0
    while roomA == 0 or roomB == 0:
        if roomA == 0:
            print_status (roomA, roomB, 'A')
            roomA = check_clean (roomA)
            print ("Room A cleared")

Code:

```
class VacuumCleaner:
    def __init__(self, room_a_status, room_b_status):
        # Initialize the status of the rooms and the vacuum's position
        self.rooms = {'A': room_a_status, 'B': room_b_status}
        self.current_room = 'A'

    def check_clean(self):
        """Checks if the current room is clean."""
        if self.rooms[self.current_room] == 'dirty':
            print(f"Room {self.current_room} is dirty. Cleaning now...")
            self.rooms[self.current_room] = 'clean'
        else:
            print(f"Room {self.current_room} is already clean.")

    def print_status(self):
        """Prints the status of both rooms."""
        print("\nRoom Status:")
        for room, status in self.rooms.items():
            print(f"Room {room}: {status}")
        print()

    def move_rooms(self):
        """Moves the vacuum cleaner to the other room."""
        if self.current_room == 'A':
            self.current_room = 'B'
        else:
            self.current_room = 'A'
        print(f"Moved to Room {self.current_room}.")

    def start_cleaning(self, steps):
        """Runs the cleaning process for a specified number of steps."""
        for step in range(steps):
            print(f"\nStep {step + 1}:")
            self.print_status()
            self.check_clean()
            self.move_rooms()
        print("\nFinal Room Status:")
        self.print_status()


# Main execution
def main():
    print("Enter the initial status of Room A (clean/dirty):")
    room_a_status = input().strip().lower()
    print("Enter the initial status of Room B (clean/dirty):")
    room_b_status = input().strip().lower()

    # Validate input
    valid_statuses = {'clean', 'dirty'}
    if room_a_status not in valid_statuses or room_b_status not in valid_statuses:
        print("Invalid input! Please enter 'clean' or 'dirty' for room statuses.")
        return

    vacuum = VacuumCleaner(room_a_status, room_b_status)
```

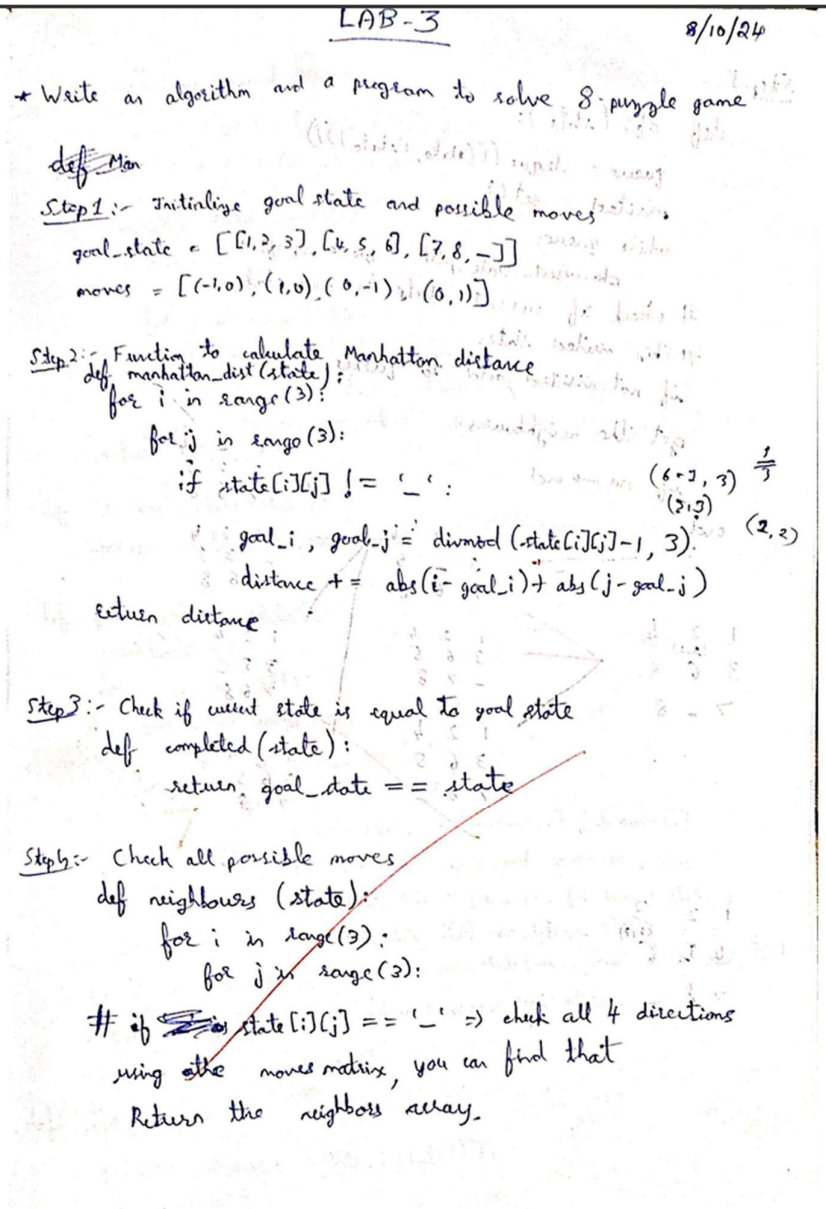
```
    steps = 4  # Number of steps for the simulation
    vacuum.start_cleaning(steps)


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

OUTPUT:

```
Enter the initial status of Room A (clean/dirty):
dirty
Enter the initial status of Room B (clean/dirty):
dirty

Step 1:

Room Status:
Room A: dirty
Room B: dirty

Room A is dirty. Cleaning now...
Moved to Room B.

Step 2:

Room Status:
Room A: clean
Room B: dirty

Room B is dirty. Cleaning now...
Moved to Room A.

Step 3:

Room Status:
Room A: clean
Room B: clean

Room A is already clean.
Moved to Room B.

Step 4:

Room Status:
Room A: clean
Room B: clean

Room B is already clean.
Moved to Room A.

Final Room Status:

Room Status:
Room A: clean
Room B: clean
```

LAB 3: Implement 8 puzzle problems

Algorithm:



LAB-3 8/10/24

* Write an algorithm and a program to solve 8-puzzle game

~~def Man~~

Step 1:- Initialize goal state and possible moves

```
goal_state = [[1,2,3],[4,5,6],[7,8,_]]
moves = [(-1,0),(1,0),(0,-1),(0,1)]
```

Step 2:- Function to calculate Manhattan distance

```
def manhattan_dist(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] != '_':
                goal_i, goal_j = divmod(state[i][j]-1, 3)
                distance += abs(i-goal_i)+abs(j-goal_j)
    return distance
```

Step 3:- Check if current state is equal to goal state

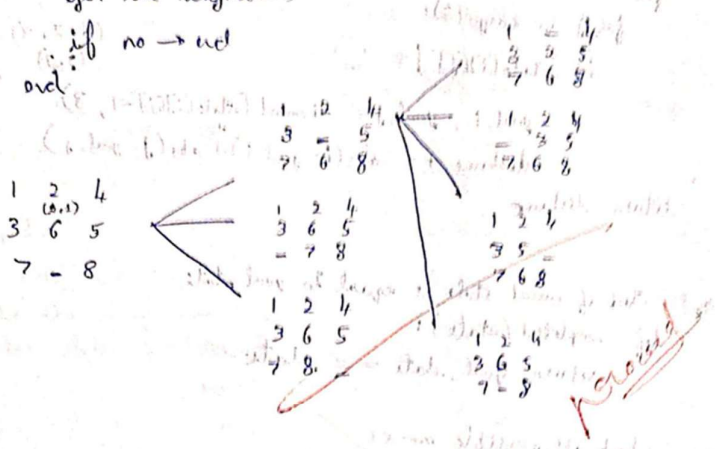
```
def completed(state):
    return goal_state == state
```

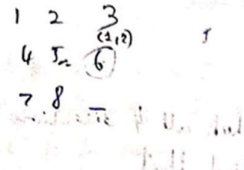
Step 4:- Check all possible moves

```
def neighbours(state):
    for i in range(3):
        for j in range(3):
# if state[i][j] == '_' => check all 4 directions
using the moves matrix, you can find that
Return the neighbors array.
```

Step 5 :-

```
def dfs (state):
    queue = deque ([(state, [state])])
    visited = set()
    while queue:
        current_state, path = queue.popleft()
    # check if current state is goal state
    # skip visited states
    if not visited, add to queue
    get the neighbours
    if no → end
end
```



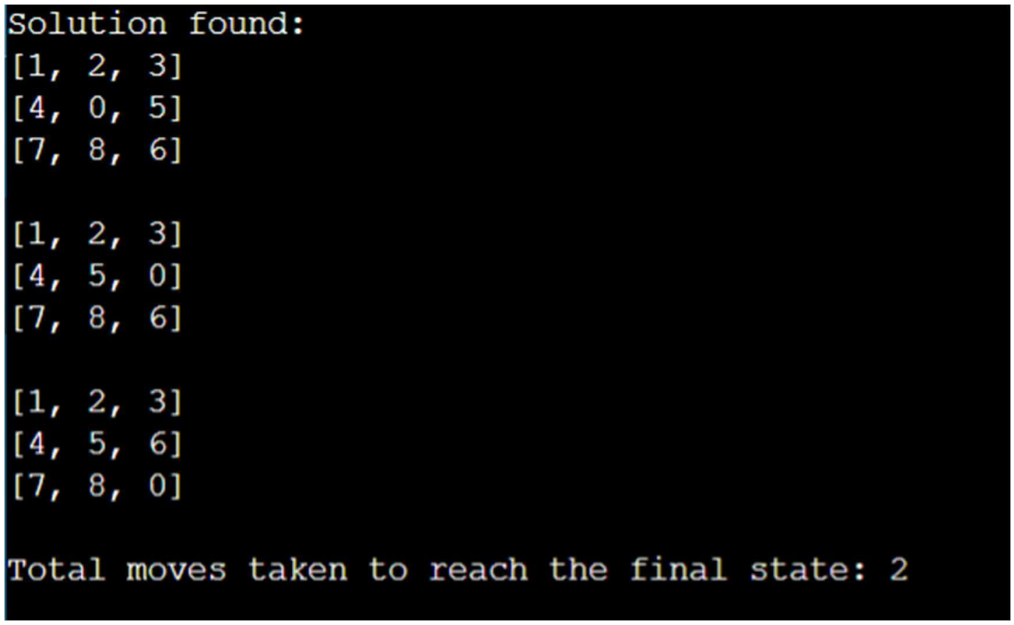


Code:

```
import numpy as np
from copy import deepcopy

class PuzzleSolver:
    def __init__(self, initial_state, goal_state):
        self.initial_state = initial_state
        self.goal_state = goal_state
        self.visited = set()

    def manhattan_distance(self, state):
        """Calculate the Manhattan distance of a state."""
        distance = 0
        for i in range(3):
            for j in range(3):
                value = state[i][j]
                if value != 0:  # Skip the blank tile
                    goal_x, goal_y = [(x, y) for x in range(3) for y in range(3) if self.goal_state[x][y] == value][0]
                    distance += abs(goal_x - i) + abs(goal_y - j)
        return distance

    def is_goal_state(self, state):
        """Check if a state matches the goal state."""
        return state == self.goal_state

    def get_neighbors(self, state):
        """Generate all valid neighbor states from the current state."""
        neighbors = []
        state_array = np.array(state)
        x, y = np.where(state_array == 0)
        x, y = x.item(), y.item()  # Blank tile's position as scalars
        moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right

        for dx, dy in moves:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                # Swap blank tile with the adjacent tile
                new_state = deepcopy(state)
                new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y], new_state[x][y]
                neighbors.append(new_state)

        return neighbors

    def dfs(self, state, depth=0, max_depth=50):
        """Perform DFS to find the solution."""
        if depth > max_depth:  # Depth limit to avoid infinite loops
            return None

        if self.is_goal_state(state):
            return [state]

        # Convert state to a tuple to hash it
        self.visited.add(tuple(map(tuple, state)))

        neighbors = self.get_neighbors(state)
        # Sort neighbors by Manhattan distance to prioritize promising states
        neighbors.sort(key=lambda n: self.manhattan_distance(n))
```

```
        for neighbor in neighbors:
            if tuple(map(tuple, neighbor)) not in self.visited:
                path = self.dfs(neighbor, depth + 1, max_depth)
                if path:
                    return [state] + path

        return None


# Main execution
def main():
    initial_state = [
        [1, 2, 3],
        [4, 0, 5],
        [7, 8, 6]
    ]
    goal_state = [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 0]
    ]

    solver = PuzzleSolver(initial_state, goal_state)
    solution = solver.dfs(initial_state)

    if solution:
        print("Solution found:")
        for step, state in enumerate(solution):
            print(f"Step {step}:")
            for row in state:
                print(row)
            print()
    else:
        print("No solution found within the depth limit.")


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

OUTPUT:


Solution found:
[1, 2, 3]
[4, 0, 5]
[7, 8, 6]
[1, 2, 3]
[4, 5, 0]
[7, 8, 6]
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Total moves taken to reach the final state: 2

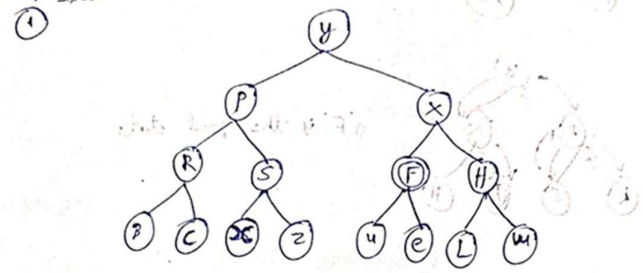
LAB 4: Iterative deepening search algorithm

Algorithm:

Lab - 04

* Write an algorithm and code for Iterative deepening depth first search and solve 8 puzzle using A* algorithm.

→ Iterative deepening DFS



⇒ Iterative deepening DFS is a combination of DFS and BFS. It goes to each level and traverses each level. If the goal state is not.

Step 1: Call the limit search function from range (1, max-size)
goal ⇒ Given as an i/p

```
def IDDFS(graph, limit, start):
    for depth = 0 to limit:
        result = DFS(start, limit, depth)
        if result:
            return result
        else:
            return None

def DFS(root, limit, depth):
    if root == goal:
        return root
    if depth == limit+1 : return
    for child in root:
        DFS(child, limit, depth)
```

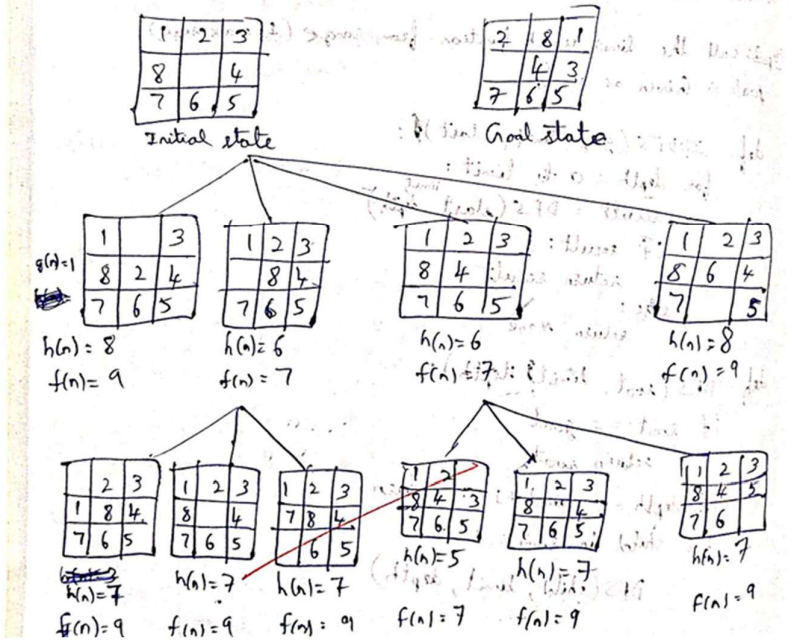
Depth = 1

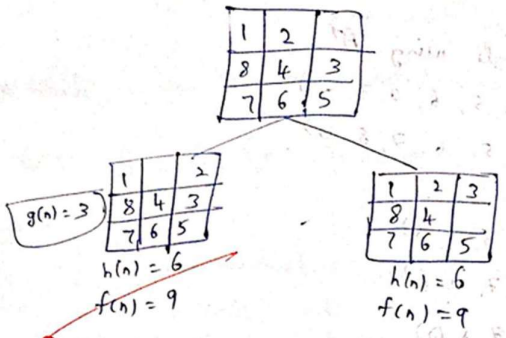
Depth = 2

Depth = 3

⇒ 'F' is the goal state

② 8 puzzle using A* algorithm





Algorithm :-

```
def Astar(prestate, goalstate):
    for i in range(1, maxDepth):
    cost = 0
    if prestate != visited[]:
        states[] = generateMoves(prestate)
        for j in states:
            f(n) = cost(j) + manhattan(prestate, goalState)
    count++;
    mini = min(f)
    visited.append(state)
    Astar(state, goalstate)
```

Output :-

1) Iterative deepening DFS

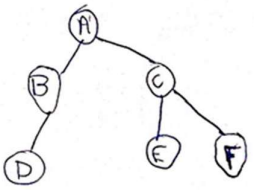
Iteration 1

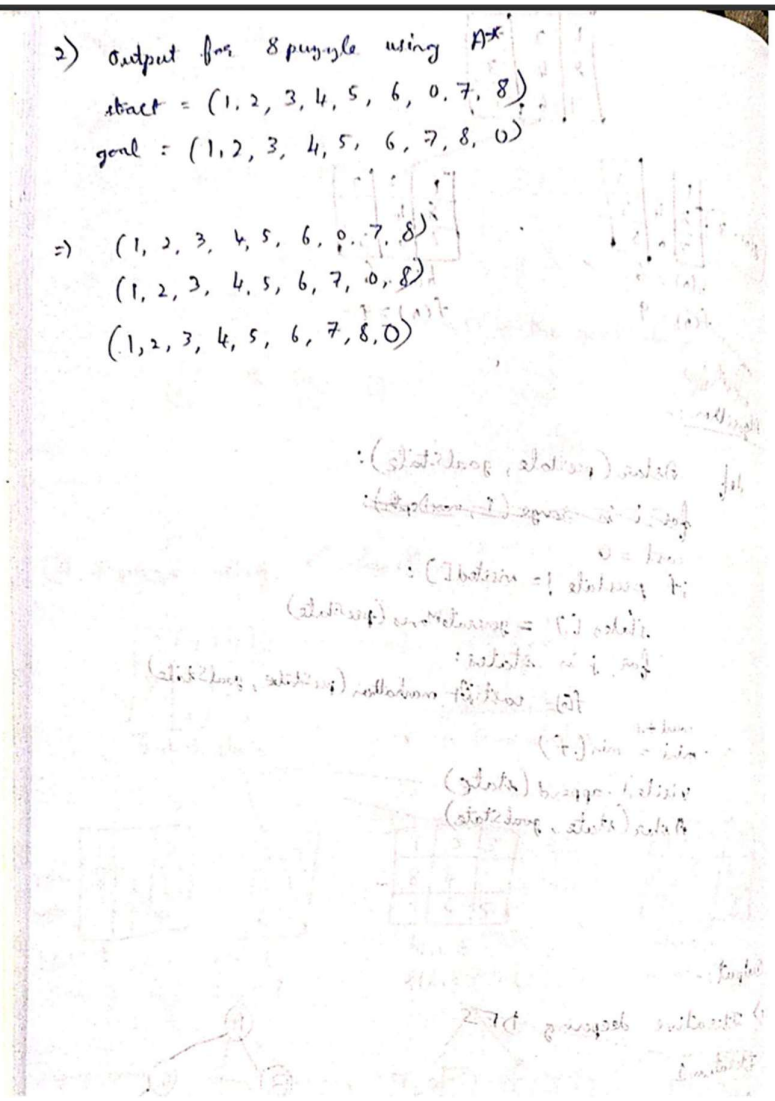
A → B → C →

Iteration 2

A → B → D → C → E →

Target node E found !





Code:

```
import heapq
from copy import deepcopy

class PuzzleSolver:
    def __init__(self, initial_state, goal_state):
        self.initial_state = initial_state
        self.goal_state = goal_state

    def manhattan_distance(self, state):
        """Calculate the Manhattan distance of a state."""
        distance = 0
        for i in range(3):
            for j in range(3):
                value = state[i][j]
                if value != 0:  # Skip the blank tile
                    goal_x, goal_y = [(x, y) for x in range(3) for y in range(3) if self.goal_state[x][y] == value][0]
```

```
                distance += abs(goal_x - i) + abs(goal_y - j)
    return distance

def is_goal_state(self, state):
    """Check if a state matches the goal state."""
    return state == self.goal_state

def get_neighbors(self, state):
    """Generate all valid neighbor states from the current state."""
    neighbors = []
    x, y = [(i, j) for i in range(3) for j in range(3) if state[i][j] == 0][0]  # Locate blank tile
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right

    for dx, dy in moves:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            # Swap blank tile with the adjacent tile
            new_state = deepcopy(state)
            new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y], new_state[x][y]
            neighbors.append(new_state)

    return neighbors

def iddfs(self, state, max_depth=50):
    """Perform Iterative Deepening Depth First Search (IDDFS)."""
    def dls(current_state, depth):
        if depth == 0:
            return None
        if self.is_goal_state(current_state):
            return [current_state]

        for neighbor in self.get_neighbors(current_state):
            result = dls(neighbor, depth - 1)
            if result:
                return [current_state] + result
        return None

    for depth in range(1, max_depth + 1):
        result = dls(state, depth)
        if result:
            return result
    return None

def a_star(self):
    """Perform A* search to solve the puzzle."""
    open_set = []
    heapq.heappush(open_set, (0, self.initial_state, []))  # (f, state, path)

    closed_set = set()

    while open_set:
        f, current_state, path = heapq.heappop(open_set)

        if self.is_goal_state(current_state):
            return path + [current_state]

        closed_set.add(tuple(map(tuple,  current_state)))
```

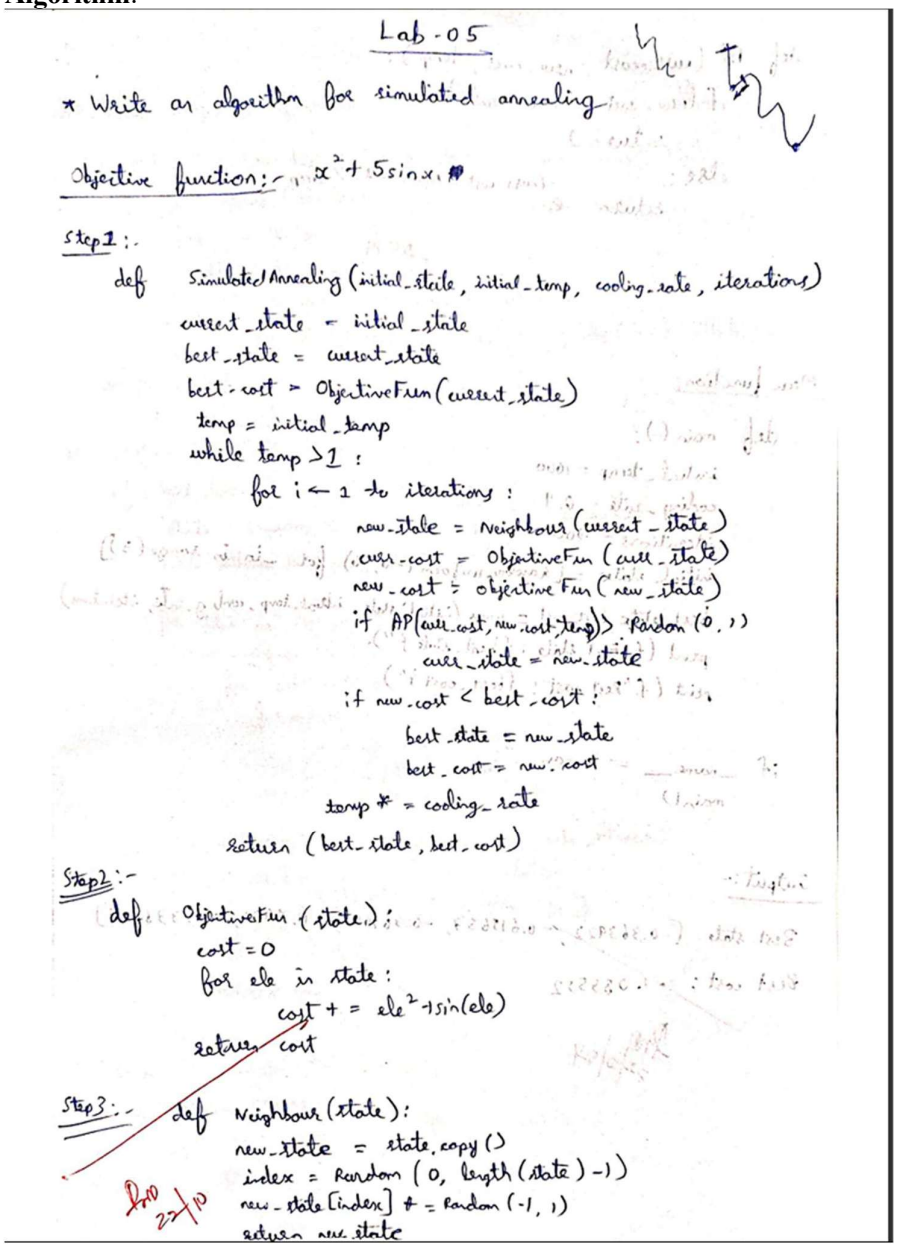
```
            for neighbor in self.get_neighbors(current_state):
                if tuple(map(tuple, neighbor)) not in closed_set:
                    g = len(path) + 1  # Cost to reach this neighbor
                    h = self.manhattan_distance(neighbor)  # Heuristic cost
                    heapq.heappush(open_set, (g + h, neighbor, path + [current_state]))

        return None


# Main execution
def main():
    initial_state = [
        [1, 2, 3],
        [4, 0, 5],
        [7, 8, 6]
    ]
    goal_state = [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 0]
    ]

    solver = PuzzleSolver(initial_state, goal_state)

    print("Using IDDFS:")
    iddfs_solution = solver.iddfs(initial_state)
    if iddfs_solution:
        for step, state in enumerate(iddfs_solution):
            print(f"Step {step}:")
            for row in state:
                print(row)
            print()
    else:
        print("No solution found with IDDFS.")

    print("Using A* Algorithm:")
    a_star_solution = solver.a_star()
    if a_star_solution:
        for step, state in enumerate(a_star_solution):
            print(f"Step {step}:")
            for row in state:
                print(row)
            print()
    else:
        print("No solution found with A* Algorithm.")


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

OUTPUT:

```
Using IDDFS:
Step 0:
[1, 2, 3]
[4, 0, 5]
[7, 8, 6]

Step 1:
[1, 2, 3]
[4, 5, 0]
[7, 8, 6]

Step 2:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Using A* Algorithm:
Step 0:
[1, 2, 3]
[4, 0, 5]
[7, 8, 6]

Step 1:
[1, 2, 3]
[4, 5, 0]
[7, 8, 6]

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

LAB 5: Simulated Annealing

Algorithm:



Lab-05

* Write an algorithm for simulated annealing

Objective function:- $x^2 + 5\sin x$

Step 1 :-

```
def SimulatedAnnealing(initial_state, initial_temp, cooling_rate, iterations)
    current_state = initial_state
    best_state = current_state
    best_cost = ObjectiveFun(current_state)
    temp = initial_temp
    while temp > 1 :
        for i <- 1 to iterations :
            new_state = Neighbour(current_state)
            curr_cost = ObjectiveFun(curr_state)
            new_cost = objective Fun(new_state)
            if AP(curr_cost, new_cost, temp) > Random(0,1)
                curr_state = new_state
            if new_cost < best_cost :
                best_state = new_state
                best_cost = new_cost
        temp *= cooling_rate
    return (best_state, best_cost)
```

Step 2 :-

```
def ObjectiveFun(state):
    cost = 0
    for ele in state :
        cost += ele^2 + sin(ele)
    return cost
```

Step 3 :-

```
def Neighbour(state):
    new_state = state.copy()
    index = Random(0, length(state)-1)
    new_state[index] += Random(-1, 1)
    return new_state
```

Prof 22/10

```
def AP (curr_cost , new_cost, temp ):
    if (new_cost < curr_cost ):
        return 1
    else :
        return e^-(curr_cost - new_cost)/temp
```

$e^{-\Delta E/T}$

Main function:-

```
def main ():
    initial_temp = 1000
    cooling_rate = 0.9
    iterations = 1000
    initial_state = [random.uniform (-10, 10) for _ in range(5)]
    best_state, best_cost = sima (initial_state, initial_temp, cooling_rate, iterations)
    print (f"Best state : {best_state}")
    print (f"Best cost : {best_cost}")

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

Output:-

Best state : [-0.363922, 0.615657, -0.367044, 0.35904, 0.336619]

Best cost : 1.085552

Prac 22/9/24

Code:

```
import math
import random

def objective_function(x):
    """Calculate the value of the objective function: f(x) = x^2 + 5*sin(x)."""
    return x**2 + 5 * math.sin(x)

def simulated_annealing(objective, x_start, temperature, cooling_rate, max_iterations):
    """
    Solve the objective function using the Simulated Annealing algorithm.

    Parameters:
    - objective: The objective function to minimize.
    - x_start: Initial guess.
    - temperature: Initial temperature.
    - cooling_rate: Rate at which the temperature decreases.
    - max_iterations: Maximum number of iterations.

    Returns:
    - Best solution found and its objective value.
    """
    current_x = x_start
    current_value = objective(current_x)
    best_x = current_x
    best_value = current_value

    for i in range(max_iterations):
        # Generate a new candidate solution in the neighborhood
        new_x = current_x + random.uniform(-1, 1)
        new_value = objective(new_x)

        # Calculate the change in the objective function
        delta = new_value - current_value

        # Accept the new solution with probability based on temperature
        if delta < 0 or random.uniform(0, 1) < math.exp(-delta / temperature):
            current_x = new_x
            current_value = new_value

            # Update the best solution found
            if current_value < best_value:
                best_x = current_x
                best_value = current_value

        # Decrease the temperature
        temperature *= cooling_rate

        # Log the process
        print(f"Iteration {i + 1}: Current x = {current_x:.5f}, Current value = {current_value:.5f}, Temperature =
{temperature:.5f}")

        # Stop if temperature is very low
        if temperature < 1e-8:
            break

    return best_x, best_value
```

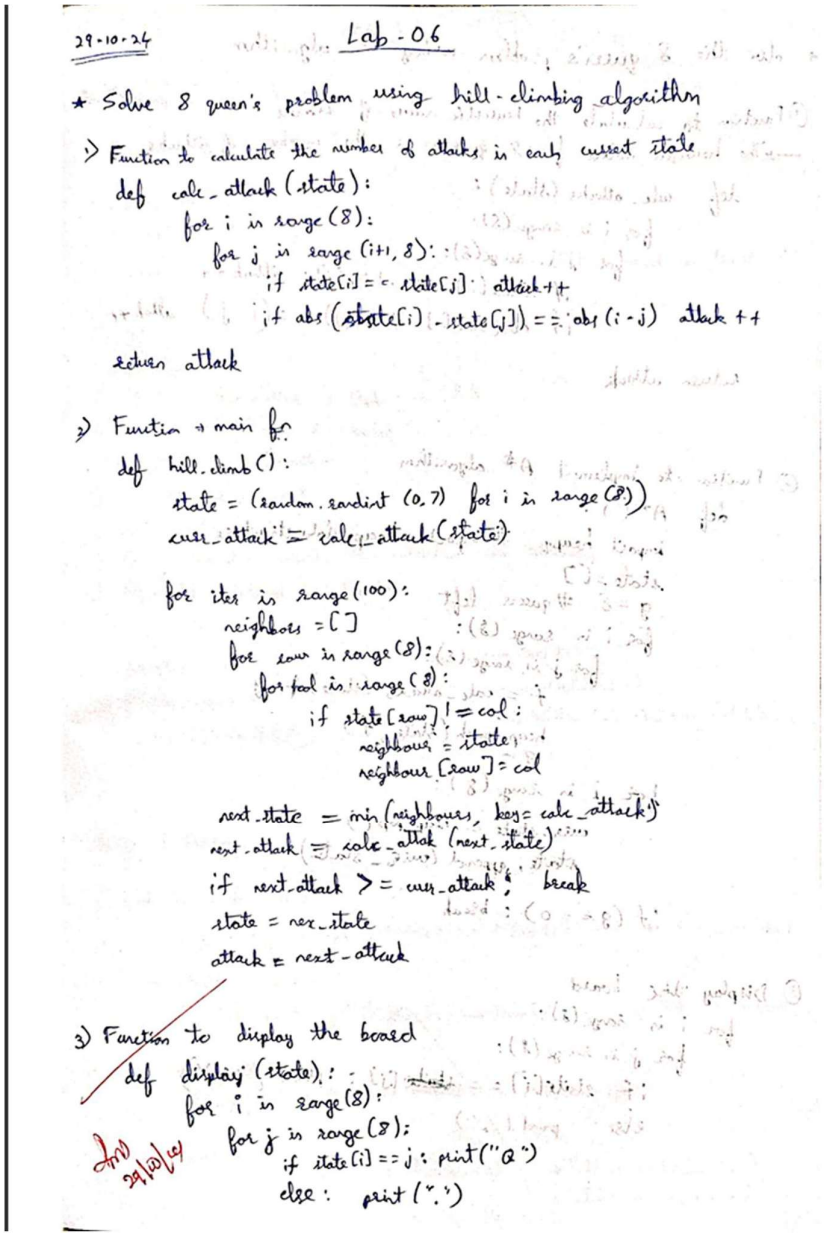
```
# Main execution
if __name__ == "__main__":
    # Problem settings
    initial_guess = random.uniform(-10, 10)  # Random initial guess in the range [-10, 10]
    initial_temperature = 1000
    cooling_rate = 0.99
    max_iterations = 1000

    # Solve using Simulated Annealing
    best_solution, best_value = simulated_annealing(
        objective_function,
        initial_guess,
        initial_temperature,
        cooling_rate,
        max_iterations
    )

    print("\nBest solution found:")
    print(f"x = {best_solution:.5f}")
    print(f"f(x) = {best_value:.5f}")
```

OUTPUT:

```
Iteration 1: Current x = -5.56416, Current value = 34.25313, Temperature = 990.00000
Iteration 2: Current x = -6.02604, Current value = 37.58479, Temperature = 980.10000
Iteration 3: Current x = -6.24016, Current value = 39.15466, Temperature = 970.29900
Iteration 4: Current x = -6.85487, Current value = 44.28396, Temperature = 960.59601
Iteration 5: Current x = -6.21720, Current value = 38.98326, Temperature = 950.99005
Iteration 6: Current x = -5.39550, Current value = 32.98950, Temperature = 941.48015
Iteration 7: Current x = -5.63427, Current value = 34.76662, Temperature = 932.06535
Iteration 8: Current x = -5.01814, Current value = 29.94981, Temperature = 922.74469
Iteration 9: Current x = -5.83909, Current value = 36.24321, Temperature = 913.51725
Iteration 10: Current x = -5.85223, Current value = 36.33726, Temperature = 904
    .38208
Iteration 11: Current x = -6.23692, Current value = 39.13040, Temperature = 895
    .33825
Iteration 12: Current x = -6.81731, Current value = 43.93031, Temperature = 886
    .38487
Iteration 13: Current x = -7.42666, Current value = 50.60492, Temperature = 877
    .52102
Iteration 14: Current x = -6.55917, Current value = 41.66021, Temperature = 868
    .74581
Iteration 15: Current x = -5.69145, Current value = 35.18161, Temperature = 860
    .05835
Iteration 16: Current x = -5.30006, Current value = 32.25183, Temperature = 851
    .45777
Iteration 17: Current x = -4.63293, Current value = 26.44825, Temperature = 842
    .94319
Iteration 18: Current x = -4.87561, Current value = 28.70509, Temperature = 834
    .51376
Iteration 19: Current x = -4.67690, Current value = 26.87028, Temperature = 826
    .16862
Iteration 20: Current x = -5.21288, Current value = 31.56089, Temperature = 817
    .90694
```

LAB 6: Implement Hill Climbing

Algorithm:



29-10-24 Lab - 06

★ Solve 8 queen's problem using hill-climbing algorithm

1) Function to calculate the number of attacks in each current state

```
def calc_attack(state):
    for i in range(8):
        for j in range(i+1, 8):
            if state[i] == state[j]: attack++
            if abs(state[i] - state[j]) == abs(i - j): attack++
    return attack
```

2) Function to main fn

```
def hill_climb():
    state = [random.randint(0,7) for i in range(8)]
    curr_attack = calc_attack(state)

    for iter in range(100):
        neighbors = []
        for row in range(8):
            for col in range(8):
                if state[row] != col:
                    neighbour = state
                    neighbour[row] = col

        next_state = min(neighbours, key = calc_attack)
        next_attack = calc_attack(next_state)
        if next_attack >= curr_attack: break
        state = nex_state
        attack = next_attack
```

3) Function to display the board

```
def display(state):
    for i in range(8):
        for j in range(8):
            if state[i] == j: print("Q")
            else: print(".")
```

29/10/24

* Solve the 8 queen's problem using A* algorithm

① Function to calculate the heuristic value of states
→ The heuristic value for 8 queens is the number of attacks

```
def calc_attacks (state):
    for i in range(8):
        for j in range(8):
            if state[i] == state[j]: attack ++
            if abs(state[i] - state[j]) == (i - j) attack++
    return attack
```

② Function to implement A* algorithm

```
def A*():
    import heapq # import a heap data structure
    state = []
    g = 8 # queens left
    for i in range (8):
        for j in range (8):
            f = calc_attacks (state) + g
            heap_push (state, f)
            g = -
    for i in range(8):
        ques_state = heap_pop ()
        state. append (ques_state)
    if (g == 0): break
```

③ Display the board

```
for i in range(8):
    for j in range(8):
        if state[i] = = state[j]: print ("Q")
        else: print (".")
```

Code:

```
import random

class EightQueensSolver:
    def __init__(self, size=8):
        self.size = size
        self.board = self.initialize_board()

    def initialize_board(self):
        """Initialize the board with one queen in each column at a random row."""
        return [random.randint(0, self.size - 1) for _ in range(self.size)]

    def calculate_conflicts(self, board):
        """Calculate the number of conflicts for a given board."""
        conflicts = 0
        for i in range(self.size):
            for j in range(i + 1, self.size):
                # Check if queens are in the same row or diagonal
                if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                    conflicts += 1
        return conflicts

    def get_neighbors(self, board):
        """Generate all possible neighbors of the current board."""
        neighbors = []
        for col in range(self.size):
            for row in range(self.size):
                if row != board[col]:
                    new_board = board[:]
                    new_board[col] = row
                    neighbors.append(new_board)
        return neighbors

    def hill_climbing(self):
        """Solve the 8-Queens problem using the Hill Climbing algorithm."""
        current_board = self.board
        current_conflicts = self.calculate_conflicts(current_board)

        while True:
            neighbors = self.get_neighbors(current_board)
            # Evaluate neighbors and find the best one
            neighbor_conflicts = [(self.calculate_conflicts(neighbor), neighbor) for neighbor in neighbors]
            best_neighbor_conflicts, best_neighbor = min(neighbor_conflicts, key=lambda x: x[0])

            # If no better neighbor is found, return the current board
            if best_neighbor_conflicts >= current_conflicts:
                return current_board, current_conflicts

            # Move to the better neighbor
            current_board = best_neighbor
            current_conflicts = best_neighbor_conflicts

    def print_board(self, board):
        """Print the chessboard."""
        for row in range(self.size):
            line = ""
            for col in range(self.size):
```

```
            if board[col] == row:
                line += "Q "
            else:
                line += ". "
        print(line)
    print()


# Main execution
if __name__ == "__main__":
    solver = EightQueensSolver()
    solution, conflicts = solver.hill_climbing()

    print("Solution found:")
    solver.print_board(solution)
    print(f"Number of conflicts: {conflicts}")
```

OUTPUT:

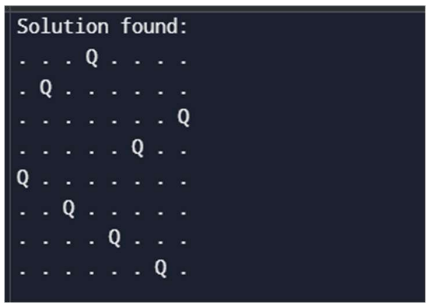
```
Solution found:
. . . . Q Q . .
. Q . . . . . .
. . . . . . . .
. . . . . . Q .
. . Q . . . . .
. . . . . . . Q
. . . Q . . . .
Q . . . . . . .

Number of conflicts: 2
Solution board (column positions for each row): [0, 6, 3, 5, 7, 1, 4, 2]
```

Code :

```
import heapq

class EightQueensSolverAStar:
    def __init__(self, size=8):
        self.size = size

    def calculate_conflicts(self, board):
        """Calculate the number of conflicts for a given board."""
        conflicts = 0
        for i in range(len(board)):
            for j in range(i + 1, len(board)):
                # Check for conflicts in rows or diagonals
                if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                    conflicts += 1
        return conflicts

    def get_neighbors(self, board):
```

```
        """Generate all possible neighbors of the current board."""
        neighbors = []
        for col in range(len(board)):
            for row in range(self.size):
                if board[col] != row:
                    new_board = board[:]
                    new_board[col] = row
                    neighbors.append(new_board)
        return neighbors

    def a_star(self):
        """Solve the 8-Queens problem using the A* algorithm."""
        # Priority queue for A* search
        open_set = []
        initial_board = [0] * self.size  # Start with all queens in the first row
        heapq.heappush(open_set, (0, 0, initial_board))  # (f, g, state)

        while open_set:
            f, g, current_board = heapq.heappop(open_set)

            # Calculate conflicts in the current state
            current_conflicts = self.calculate_conflicts(current_board)
            if current_conflicts == 0:
                return current_board  # Solution found

            # Generate neighbors and add them to the priority queue
            for neighbor in self.get_neighbors(current_board):
                h = self.calculate_conflicts(neighbor)  # Heuristic cost
                heapq.heappush(open_set, (g + 1 + h, g + 1, neighbor))  # f = g + h

        return None  # No solution found

    def print_board(self, board):
        """Print the chessboard."""
        for row in range(self.size):
            line = ""
            for col in range(self.size):
                if board[col] == row:
                    line += "Q "
                else:
                    line += ". "
            print(line)
        print()


# Main execution
if __name__ == "__main__":
    solver = EightQueensSolverAStar(size=8)
    solution = solver.a_star()

    if solution:
        print("Solution found:")
        solver.print_board(solution)
    else:
        print("No solution found.")
```

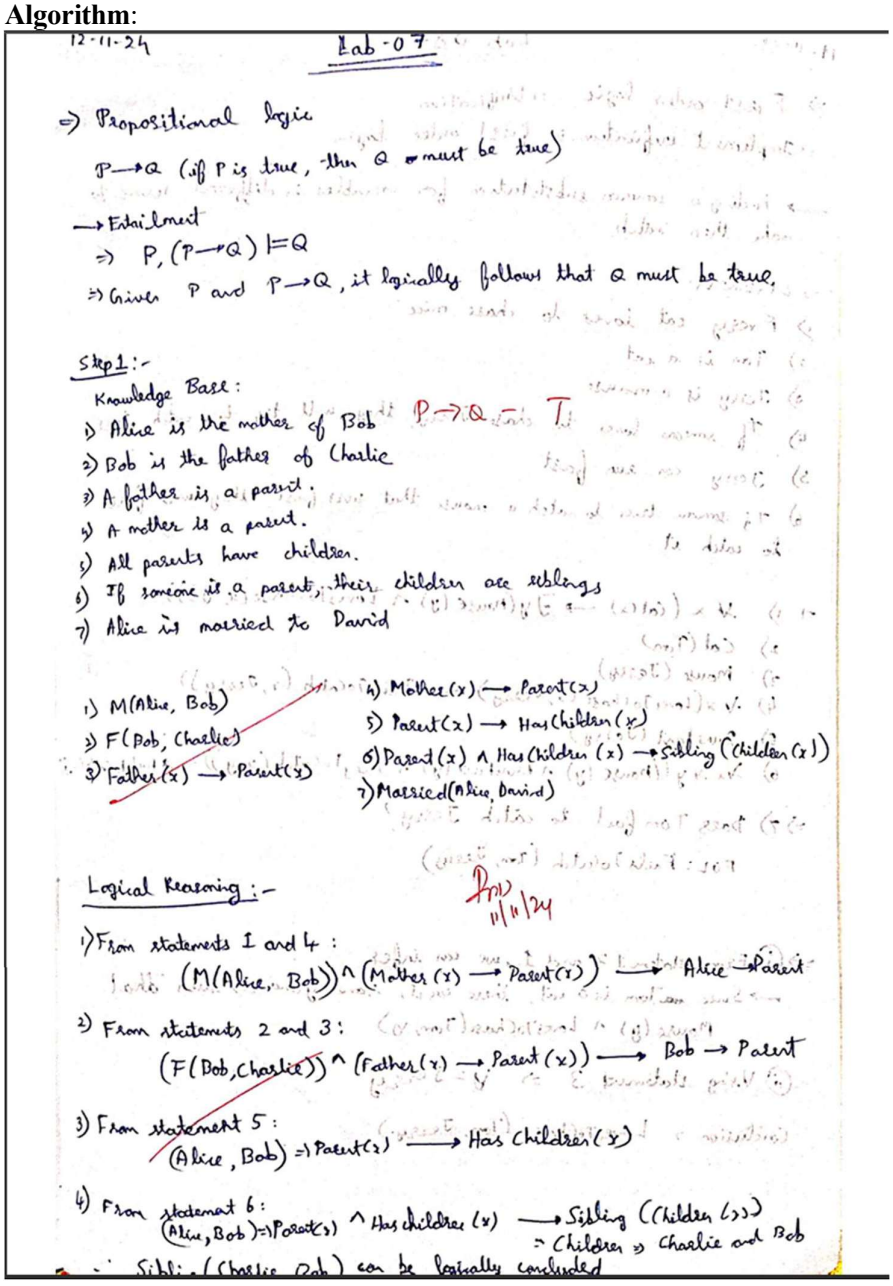
OUTPUT:



LAB 7: Propositional Logic

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

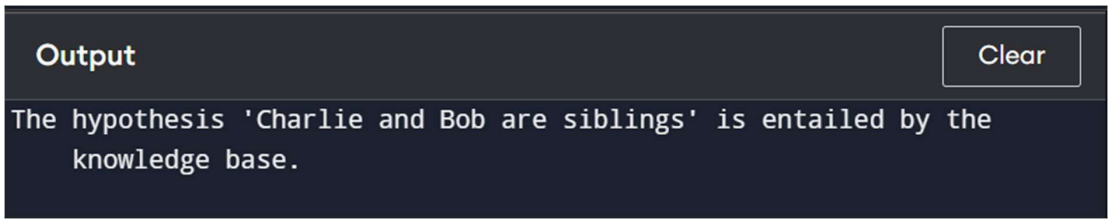
Algorithm:



Code:

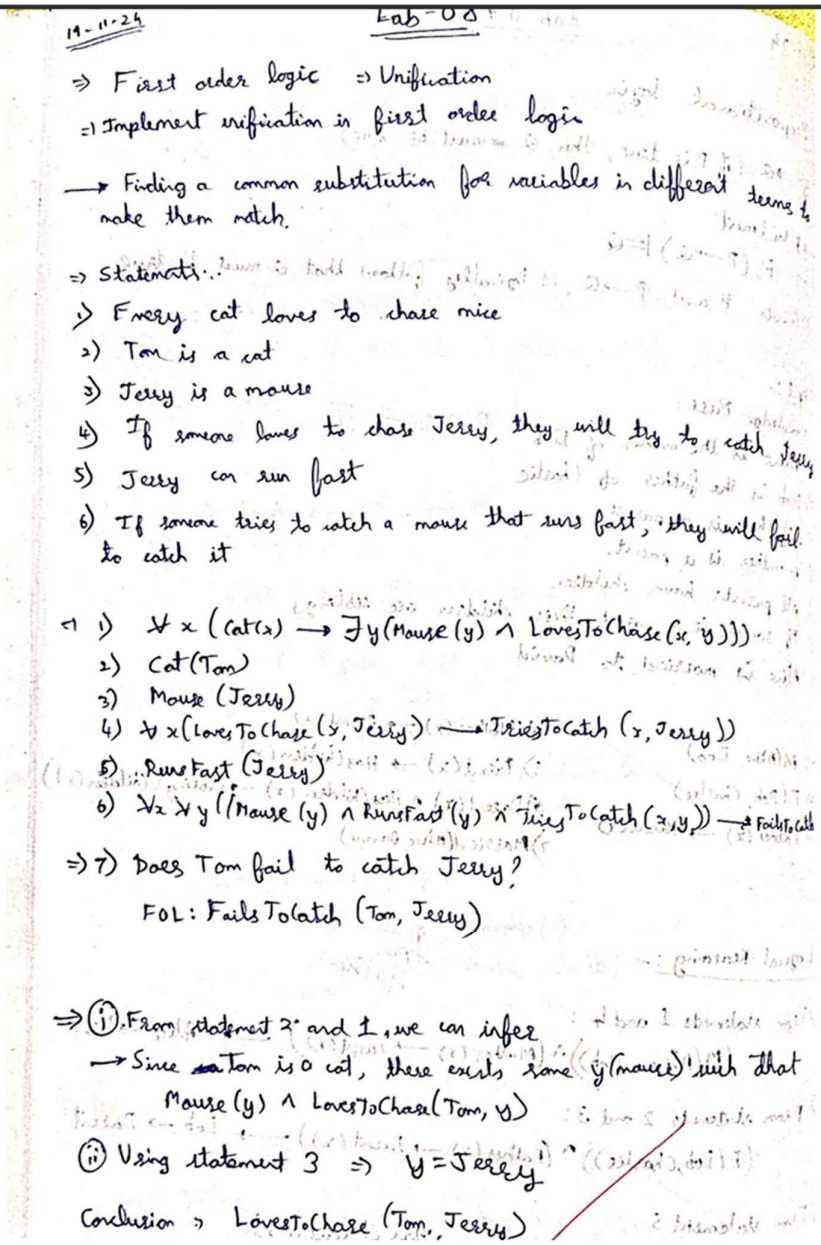
```
class KnowledgeBase:
    def __init__(self):
        self.facts = []
        self.rules = []

    def add_fact(self, fact):
        self.facts.append(fact)

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

    def infer(self):
        new_inferences = True

        while new_inferences:
            new_inferences = False

            for premise, conclusion in self.rules:
                if all(fact in self.facts for fact in premise) and conclusion not in self.facts:
                    self.facts.append(conclusion)
                    new_inferences = True

    def entails(self, hypothesis):
        return hypothesis in self.facts

# Example Usage
kb = KnowledgeBase()

# Adding facts
kb.add_fact("Alice is mother of Bob")
kb.add_fact("Bob is father of Charlie")
kb.add_fact("A father is a parent")
kb.add_fact("A mother is a parent")
kb.add_fact("All parents have children")
kb.add_fact("Alice is married to Davis")

# Adding rules
kb.add_rule(["Bob is father of Charlie", "A father is a parent"], "Bob is parent")
kb.add_rule(["Alice is mother of Bob", "A mother is a parent"], "Alice is parent")
kb.add_rule(["Bob is parent", "All parents have children"], "Charlie and Bob are siblings")

# Perform inference
kb.infer()

# Hypothesis
hypothesis = "Charlie and Bob are siblings"

if kb.entails(hypothesis):
    print(f"The hypothesis '{hypothesis}' is entailed by the knowledge base.")
else:
    print(f"The hypothesis '{hypothesis}' is not entailed by the knowledge base.")
```

OUTPUT:


Output
Clear
The hypothesis 'Charlie and Bob are siblings' is entailed by the
knowledge base.

LAB 8: Unification in first order logic

Algorithm:



19-11-24 Lab-08

⇒ First order logic ⇒ Unification

⇒ Implement unification in first order logic

→ Finding a common substitution for variables in different terms to make them match.

⇒ Statements:

1) Every cat loves to chase mice
2) Tom is a cat
3) Jerry is a mouse
4) If someone loves to chase Jerry, they will try to catch Jerry
5) Jerry can run fast
6) If someone tries to catch a mouse that runs fast, they will fail to catch it

⇒ 1) $\forall x\,(Cat(x) \rightarrow \exists y\,(Mouse(y) \wedge LovesToChase(x, y))$
2) $Cat(Tom)$
3) $Mouse(Jerry)$
4) $\forall x\,(LovesToChase(x, Jerry) \rightarrow TriesToCatch(x, Jerry))$
5) $RunsFast(Jerry)$
6) $\forall x \forall y\,((Mouse(y) \wedge RunsFast(y) \wedge TriesToCatch(x, y)) \rightarrow FailsToCatch$

⇒ 7) Does Tom fail to catch Jerry?

FOL: $FailsToCatch(Tom, Jerry)$

⇒ (i) From statement 2 and 1, we can infer
→ Since Tom is a cat, there exists some y(mouse) such that
$Mouse(y) \wedge LovesToChase(Tom, y)$

(ii) Using statement 3 ⇒ $y = Jerry$

Conclusion ⇒ $LovesToChase(Tom, Jerry)$

(iii) From statement 4 and the fact that LovesToChase(Tom, Jerry)
Conclusion: TriesToCatch(Tom, Jerry)

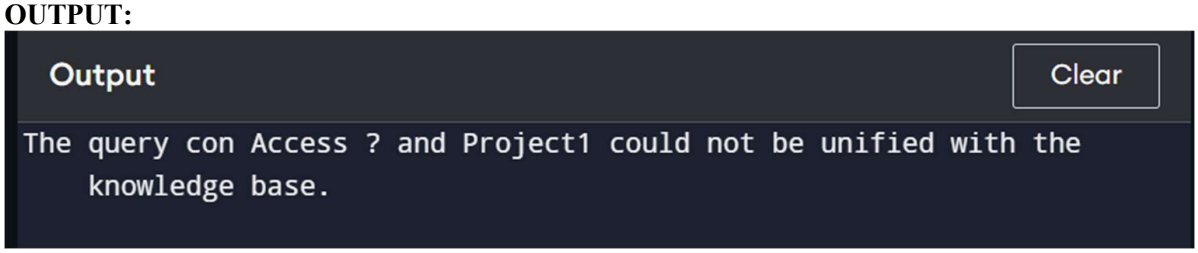
(iv) From statement 5 and 3, Jerry satisfies the conditions of statement 4
Since, Mouse(Jerry), RunsFast(Jerry) and TriesToCatch(Tom, Jerry),
we conclude ⇒ FailsToCatch(Tom, Jerry)

→ ∴ The unification is successful, and FailsToCatch(Tom, Jerry) is true based on the knowledge base.

Code:

```
def unify(kb, query):
    # Extract predicate and target project from the query
    predicate = query['predicate']
    target_project = query['arguments'][1]

    result = []

    # Iterate through knowledge base (kb)
    for item in kb:
        if item["type"] == "eule" and predicate in item:
            rule = item["rule"]

            if "Assigned To" in rule and "con Access" in rule:
                # Check for the "Assigned To" and "con Access" facts
                for fact in kb:
                    if fact["type"] == "fort" and "Assigned To" in fact:
                        fact_parts = fact["Assigned To"].split("(")
                        fact_parts = fact_parts[1].strip(")").split(",")
                        person, project = fact_parts[0].strip(), fact_parts[1].strip()

                        if project == target_project:
                            result.append(person)

    if result:
        return f"The query {query['predicate']} {query['arguments'][0]} and {target_project} has been unified."
    else:
        return f"The query {query['predicate']} {query['arguments'][0]} and {target_project} could not be unified with the
knowledge base."

# Example knowledge base
kb = [
    {"type": "eule", "rule": "Writes( Alice, Project1)"},
    {"type": "fort", "Assigned To": "Alice(Project1)"},
    {"type": "fort", "Assigned To": "Bob(Project2)"},
]

# Example query
query = {"predicate": "con Access", "arguments": ["?", "Project1"]}

# Run unification
result = unify(kb, query)
print(result)
```

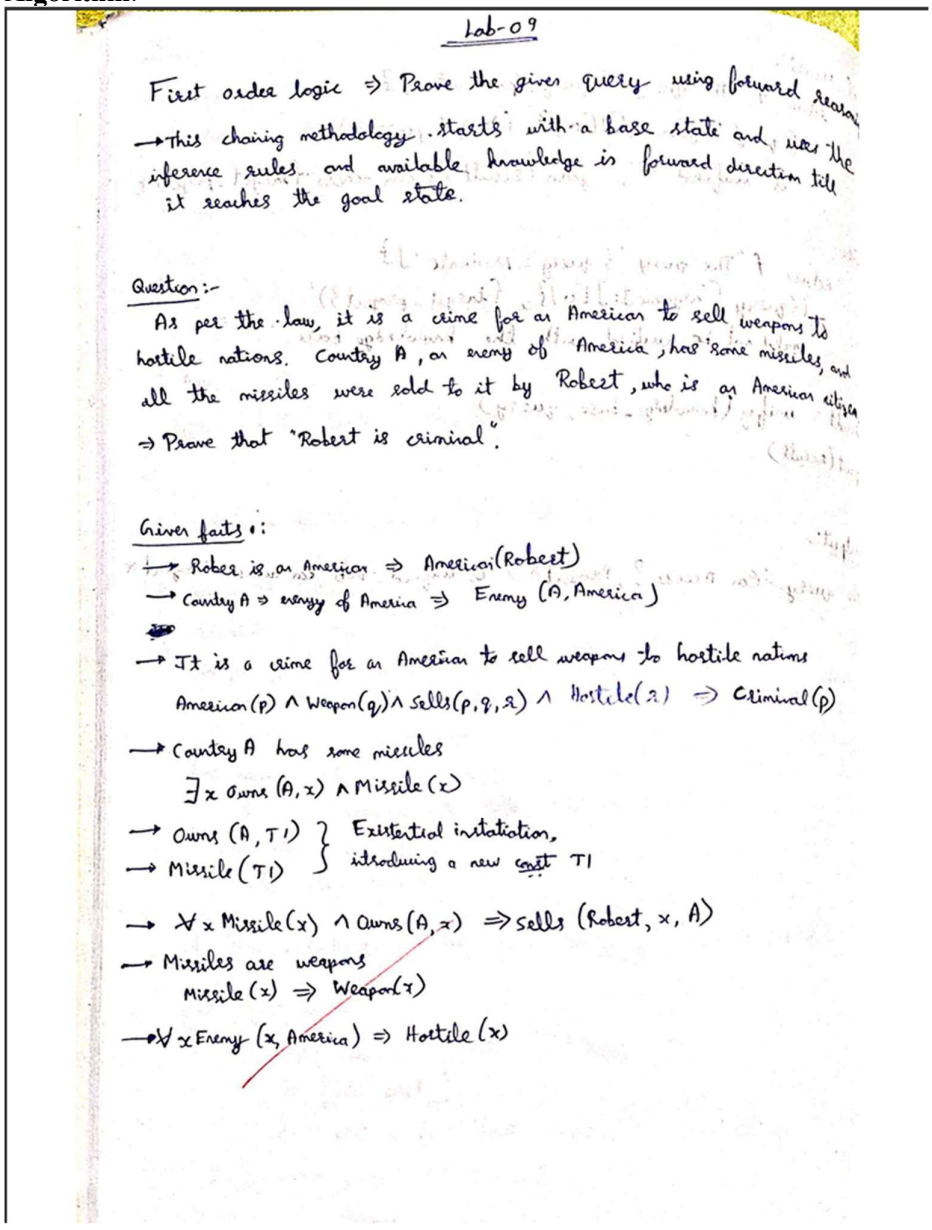
OUTPUT:



LAB 9: Forward Chaining

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

Algorithm:



Lab-09

First order logic ⇒ Prove the given query using forward reasoning

→ This chaining methodology starts with a base state and uses the inference rules and available knowledge in forward direction till it reaches the goal state.

Question:-

As per the law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, and all the missiles were sold to it by Robert, who is an American citizen

⇒ Prove that "Robert is criminal".

Given facts:

→ Robert is an American ⇒ American(Robert)

→ Country A ⇒ enemy of America ⇒ Enemy (A, America)

→ It is a crime for an American to sell weapons to hostile nations

$American(p) \wedge Weapon(q) \wedge Sells(p, q, r) \wedge Hostile(r) \Rightarrow Criminal(p)$

→ Country A has some missiles

$\exists x\ Owns(A, x) \wedge Missile(x)$

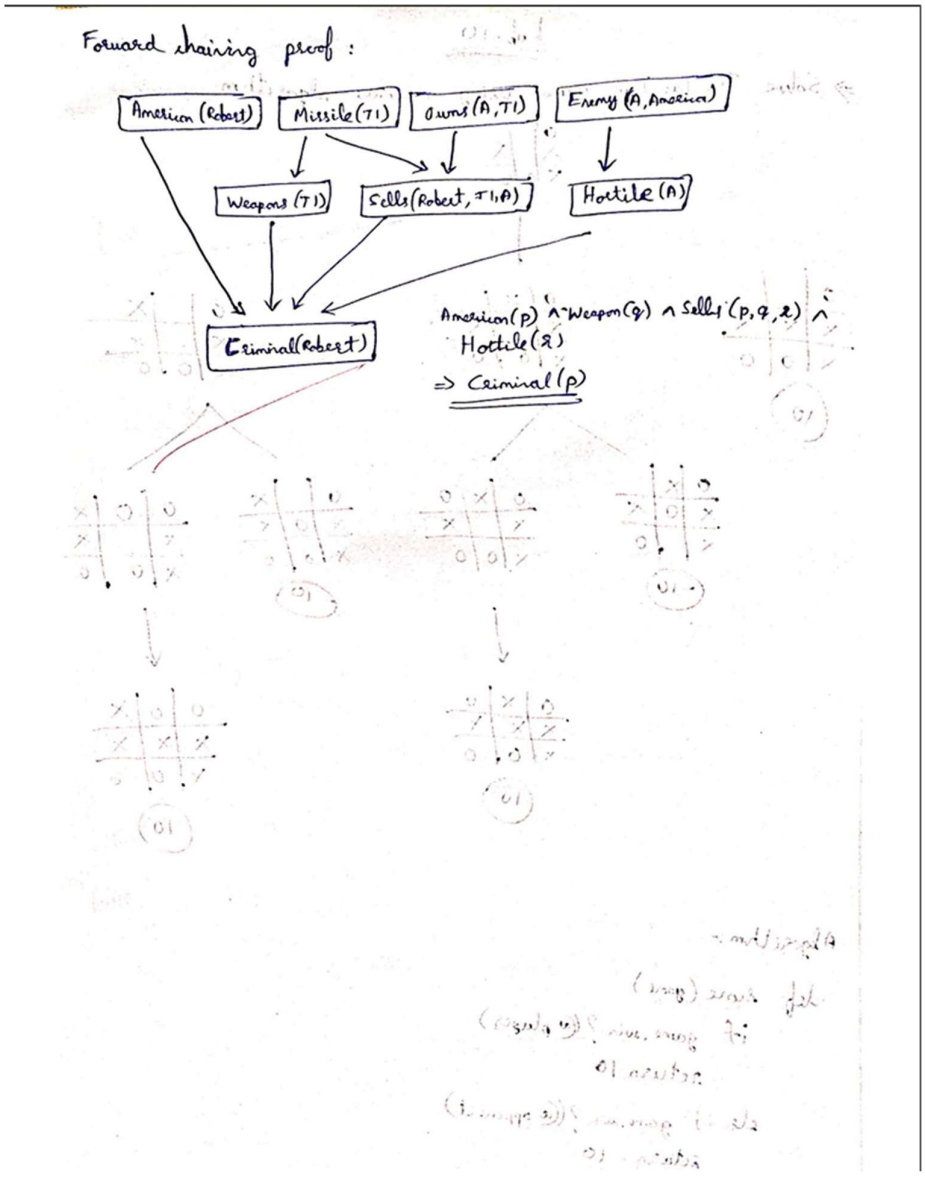
→ Owns (A, T1) } Existential instantiation,
→ Missile (T1) } introducing a new constant T1

→ $\forall x\ Missile(x) \wedge Owns(A, x) \Rightarrow Sells(Robert, x, A)$

→ Missiles are weapons

$Missile(x) \Rightarrow Weapon(x)$

→ $\forall x\ Enemy(x, America) \Rightarrow Hostile(x)$


Forward chaining proof :
American (Robert)
Missile (T1)
Owns (A, T1)
Enemy (A, America)
Weapons (T1)
Sells (Robert, T1, A)
Hostile (A)
Criminal (Robert)
American (p) ∧ Weapon (q) ∧ Sells (p, q, r) ∧
Hostile (r)
⇒ Criminal (p)

Code:

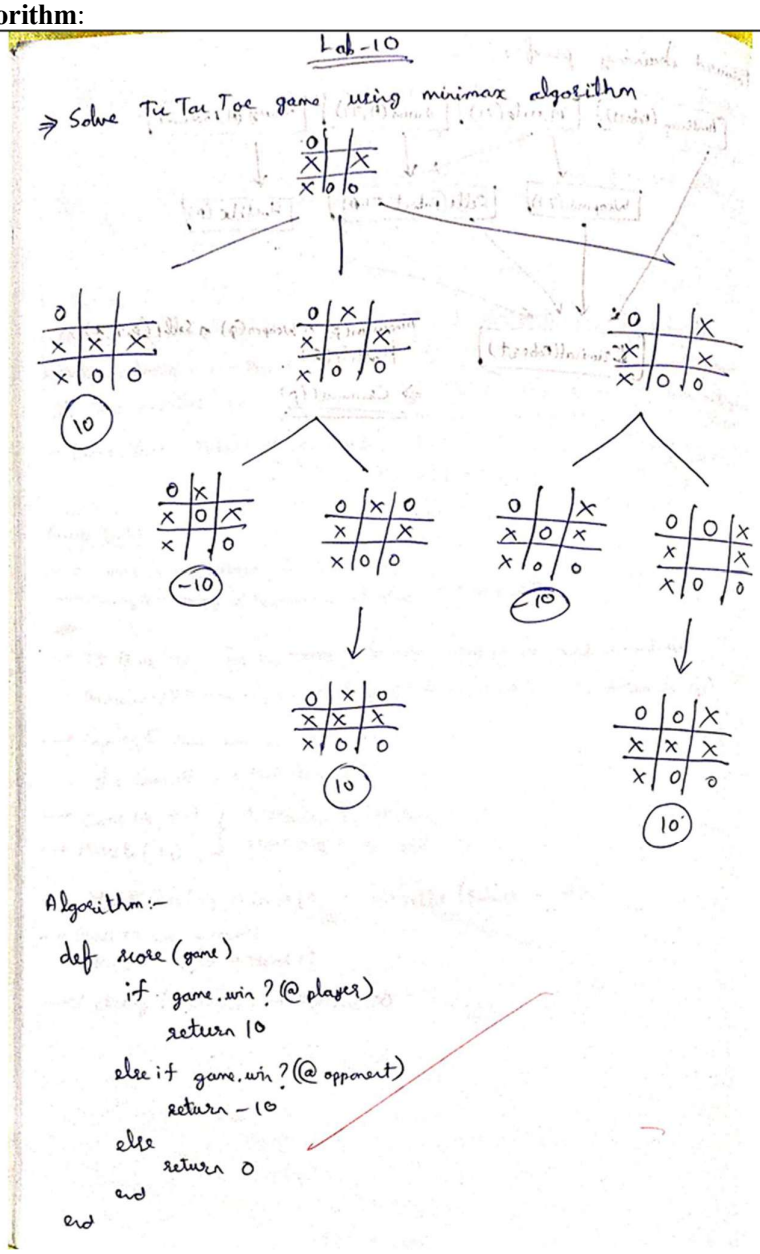
```
# Define the knowledge base (KB) as a set of facts
KB = set()

# Premises based on the provided FOL problem
KB.add('American(Robert)')
KB.add('Enemy(America, A)')
KB.add('Missile(T1)')
KB.add('Owns(A, T1)')

# Define inference rules
def modus_ponens(fact1, fact2, conclusion):
    """ Apply modus ponens inference rule: if fact1 and fact2 are true, then conclude conclusion """
    if fact1 in KB and fact2 in KB:
        KB.add(conclusion)
        print(f"Inferred: {conclusion}")

def forward_chaining():
    """ Perform forward chaining to infer new facts until no more inferences can be made """
    # 1. Apply: Missile(x) → Weapon(x)
    if 'Missile(T1)' in KB:
        KB.add('Weapon(T1)')
        print(f"Inferred: Weapon(T1)")

    # 2. Apply: Sells(Robert, T1, A) from Owns(A, T1) and Weapon(T1)
    if 'Owns(A, T1)' in KB and 'Weapon(T1)' in KB:
        KB.add('Sells(Robert, T1, A)')
        print(f"Inferred: Sells(Robert, T1, A)")

    # 3. Apply: Hostile(A) from Enemy(A, America)
    if 'Enemy(America, A)' in KB:
        KB.add('Hostile(A)')
        print(f"Inferred: Hostile(A)")

    # 4. Now, check if the goal is reached (i.e., if 'Criminal(Robert)' can be inferred)
    if 'American(Robert)' in KB and 'Weapon(T1)' in KB and 'Sells(Robert, T1, A)' in KB and 'Hostile(A)' in KB:
        KB.add('Criminal(Robert)')
        print("Inferred: Criminal(Robert)")

    # Check if we've reached our goal
    if 'Criminal(Robert)' in KB:
        print("Robert is a criminal!")
    else:
        print("No more inferences can be made.")

# Run forward chaining to attempt to derive the conclusion
forward_chaining()
```

OUTPUT:

```
Inferred: Weapon(T1)
Inferred: Sells(Robert, T1, A)
Inferred: Hostile(A)
Inferred: Criminal(Robert)
Robert is a criminal!
```

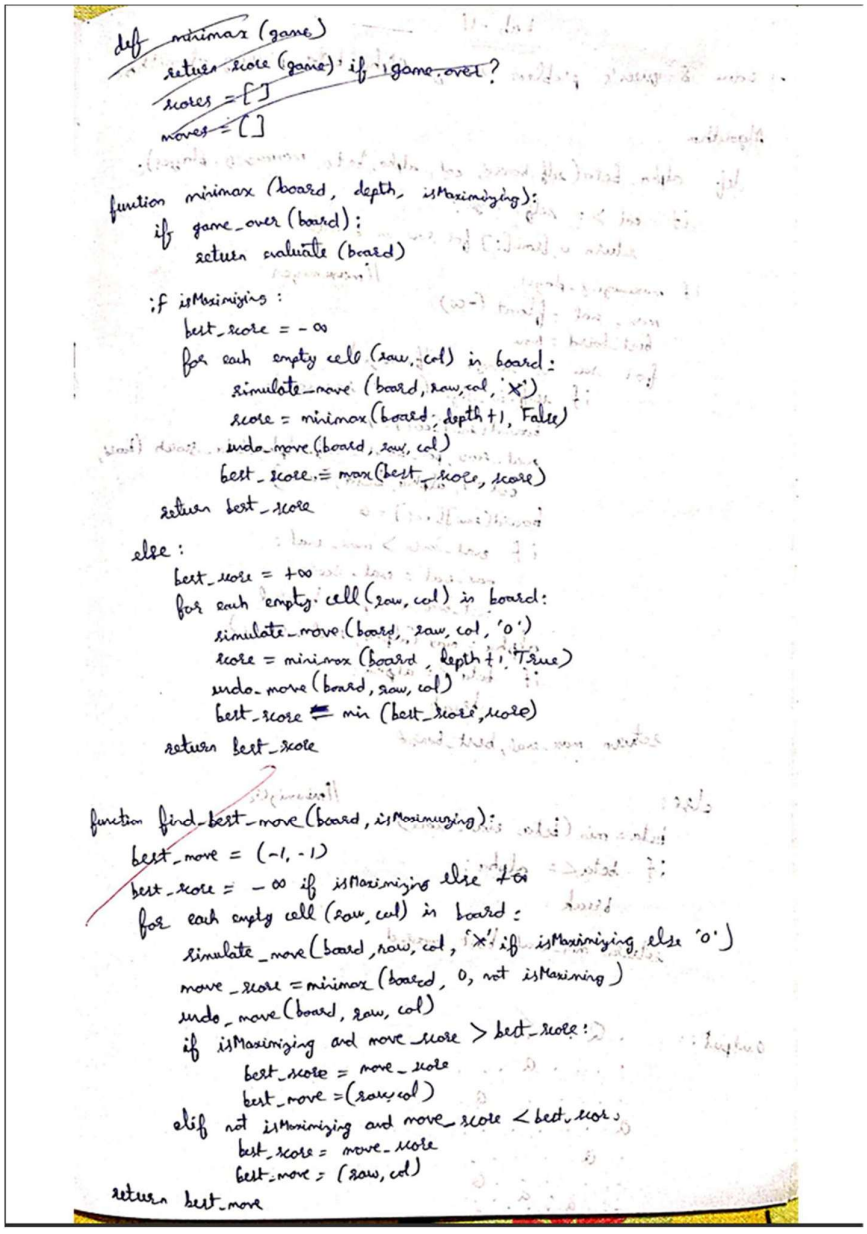
LAB 10: Implement Tic Tac Toe using Min Max

Algorithm:



Algorithm:-

```
def score(game)
    if game.win?(@player)
        return 10
    else if game.win?(@opponent)
        return -10
    else
        return 0
    end
end
```



```
def minimax (game)
    return score (game) if game_over?
    scores = []
    moves = []

function minimax (board, depth, isMaximizing):
    if game_over (board):
        return evaluate (board)

    if isMaximizing:
        best_score = -∞
        for each empty cell (row, col) in board:
            simulate_move (board, row, col, 'X')
            score = minimax (board, depth+1, False)
            undo_move (board, row, col)
            best_score = max (best_score, score)
        return best_score

    else:
        best_score = +∞
        for each empty cell (row, col) in board:
            simulate_move (board, row, col, 'O')
            score = minimax (board, depth+1, True)
            undo_move (board, row, col)
            best_score = min (best_score, score)
        return best_score

function find_best_move (board, isMaximizing):
    best_move = (-1, -1)
    best_score = -∞ if isMaximizing else +∞
    for each empty cell (row, col) in board:
        simulate_move (board, row, col, 'X' if isMaximizing else 'O')
        move_score = minimax (board, 0, not isMaximizing)
        undo_move (board, row, col)
        if isMaximizing and move_score > best_score:
            best_score = move_score
            best_move = (row, col)
        elif not isMaximizing and move_score < best_score:
            best_score = move_score
            best_move = (row, col)
    return best_move
```

Lab - 11

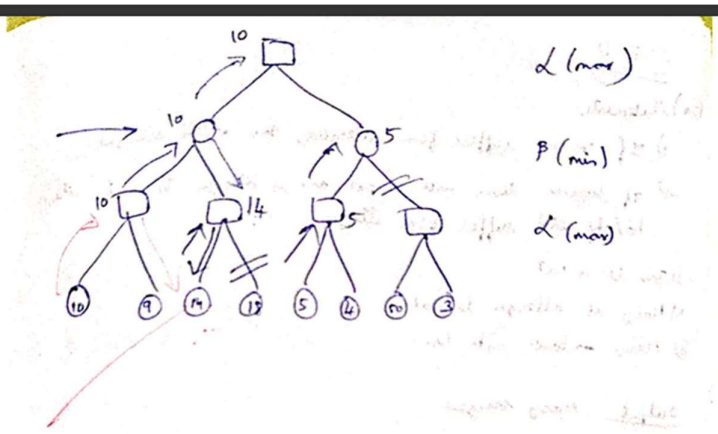
⇒ Solve 8-queen's problem using Alpha-beta pruning algorithm

Algorithm:

```
def alpha_beta(self,board, col, alpha, beta, maximizing_player):
    if col >= self.size:
        return 0,[row[:] for row in board]
    if maximizing_player:                    //maximizer
        max_eval =float(-∞)
        best_board = None
        for row in range(self.size):
            if self.is_safe(board, row, col):
                board[row][col]=1
                eval_score, pot_board = self.alpha_beta_search(board,
                  col+1, alpha, beta, False)
                board[row][col]=0
                if eval_score > max_eval:
                    max_eval = eval_score
                    best_board = pot_board
                alpha = max(alpha, eval_score)
                if beta <= alpha:
                    break
        return max_eval, best_board
    else:                                    //minimizer
        beta = min(beta, eval_score)
        if beta<= alpha:
            break
        return min_eval, best_board
```

output:

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


10
10
5
10
14
5
α (max)
β (min)
α (max)
10
9
14
15
5
4
20
3

Code:

```
import math

 def printBoard(board):
    for row in board:
       print(" | ".join(cell if cell != "" else " " for cell in row))
       print("-" * 9)

def evaluateBoard(board):
   for row in board:
      if row[0] == row[1] == row[2] and row[0] != "":
         return 10 if row[0] == 'X' else -10
   for col in range(3):
      if board[0][col] == board[1][col] == board[2][col] and board[0][col] != "":
         return 10 if board[0][col] == 'X' else -10
   if board[0][0] == board[1][1] == board[2][2] and board[0][0] != "":
      return 10 if board[0][0] == 'X' else -10
   if board[0][2] == board[1][1] == board[2][0] and board[0][2] != "":
      return 10 if board[0][2] == 'X' else -10
   return 0

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

def minimax(board, depth, isMaximizing):
   score = evaluateBoard(board)
   if score == 10 or score == -10:
      return score
    if isDraw(board):
      return 0

   if isMaximizing:
      bestScore = -math.inf
      for i in range(3):
         for j in range(3):
            if board[i][j] == "":
               board[i][j] = 'X'
               score = minimax(board, depth + 1, False)
               board[i][j] = ""
               bestScore = max(bestScore, score)
      return bestScore
   else:
      bestScore = math.inf
      for i in range(3):
         for j in range(3):
            if board[i][j] == "":
               board[i][j] = 'O'
               score = minimax(board, depth + 1, True)
               board[i][j] = ""
               bestScore = min(bestScore, score)
      return bestScore

def findBestMove(board):
```

```
    bestValue = -math.inf
    bestMove = (-1, -1)
    for i in range(3):
        for j in range(3):
            if board[i][j] == "":
                board[i][j] = 'X'
                moveValue = minimax(board, 0, False)
                board[i][j] = ""
                if moveValue > bestValue:
                    bestMove = (i, j)
                    bestValue = moveValue
    return bestMove

def playGame():
    board = [["" for _ in range(3)] for _ in range(3)]
    print("Tic Tac Toe!")
    print("You are 'O'. The AI is 'X'.")
    printBoard(board)

    while True:
        while True:
            try:
                row, col = map(int, input("Enter your move (row and column: 0, 1, or 2): ").split())
                if board[row][col] == "":
                    board[row][col] = 'O'
                    break
                else:
                    print("Cell is already taken. Choose another.")
            except (ValueError, IndexError):
                print("Invalid input. Enter row and column as two numbers between 0 and 2.")

        print("Your move:")
        printBoard(board)

        if evaluateBoard(board) == -10:
            print("You win!")
            break
        if isDraw(board):
            print("It's a draw!")
            break

        print("AI is making its move...")
        bestMove = findBestMove(board)
        board[bestMove[0]][bestMove[1]] = 'X'

        print("AI's move:")
        printBoard(board)

        if evaluateBoard(board) == 10:
            print("AI wins!")
            break
        if isDraw(board):
            print("It's a draw!")
            break

playGame()
```

OUTPUT:

```
You are 'O'. The AI is 'X'.
  |   |
---------
  |   |
---------
  |   |
---------
Enter your move (row and column: 0, 1, or 2): 1 1
Your move:
  |   |
---------
  | O |
---------
  |   |
---------
AI is making its move...
AI's move:
X |   |
---------
  | O |
---------
  |   |
---------
Enter your move (row and column: 0, 1, or 2): 1 2
Your move:
X |   |
---------
  | O | O
---------
  |   |
---------
AI is making its move...
AI's move:
X |   |
---------
X | O | O
---------
  |   |
---------
Enter your move (row and column: 0, 1, or 2): 1 2
Cell is already taken. Choose another.
Enter your move (row and column: 0, 1, or 2): 0 2
Your move:
X |   | O
---------
X | O | O
---------
  |   |
---------
AI is making its move...
AI's move:
X |   | O
---------
X | O | O
---------
X |   |
---------
AI wins!
```

PART 2: Implement Alpha-Beta Pruning

Code:

```
def is_valid(board, row, col):

    for i in range(row):
        if board[i] == col or \
           abs(board[i] - col) == abs(i - row):
            return False
    return True

def alpha_beta(board, row, alpha, beta, isMaximizing):

    if row == len(board):
        return 1

     if isMaximizing:
         max_score = 0
        for col in range(len(board)):
            if is_valid(board, row, col):
                board[row] = col
                max_score += alpha_beta(board, row + 1, alpha, beta, False)
                board[row] = -1
                alpha = max(alpha, max_score)
                if beta <= alpha:
                    break
        return max_score
    else:
        min_score = float('inf')
        for col in range(len(board)):
            if is_valid(board, row, col):
                board[row] = col
                min_score = min(min_score, alpha_beta(board, row + 1, alpha, beta, True))
                board[row] = -1
                beta = min(beta, min_score)
                if beta <= alpha:
                    break
        return min_score

def solve_8_queens():

    board = [-1] * 8
    alpha = -float('inf')
    beta = float('inf')
    return alpha_beta(board, 0, alpha, beta, True)

solutions = solve_8_queens()
print(f"Number of solutions for the 8 Queens problem: {solutions}")
```

OUTPUT: