VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

REVANTH K(1BM22CS220)

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 **REVANTH K (1BM22CS220),** 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.

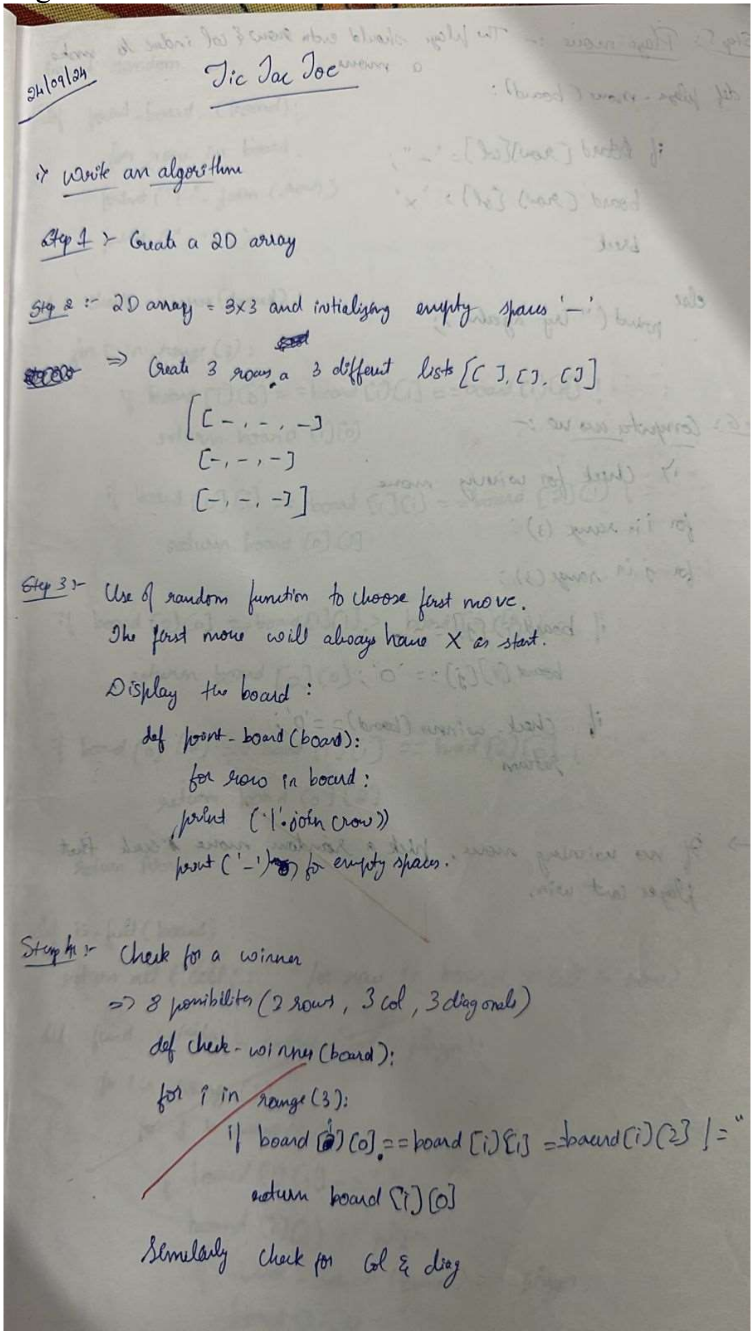
Radhika A D Assistant Professor Department of CSE, BMSCE	Dr Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

GITHUB LINK:
https://github.com/Revanth220/AI

LAB 1: Tic –Tac –Toe Game

Algorithm:



24/09/24

Tic Tac Toe

i) write an algorithm

Step 1 :- Creat a 2D array

Step 2 :- 2D array = 3×3 and intializing empty spaces '–'

⇒ Creat 3 rows, a 3 differnt lists [[ ], [ ], [ ]]

```
[[–, –, –]
 [–, –, –]
 [–, –, –]]
```

Step 3 :- Use of random function to choose first move.

The first move will always have X as start.

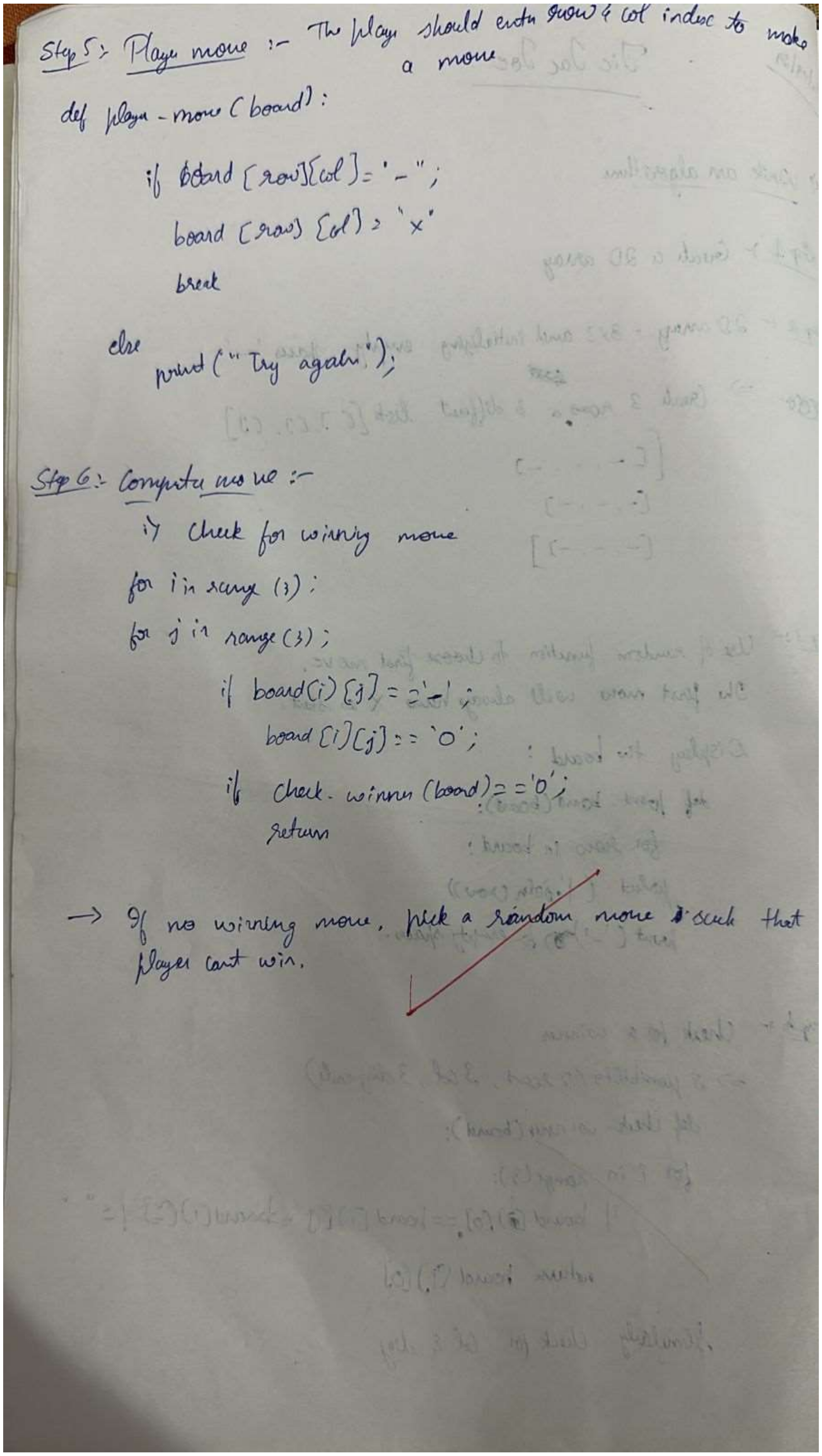
Display the board :

```
def print_board(board):
    for row in board:
    print ('|'.join(row))
    print ('–') for empty spaces.
```

Step 4 :- Check for a winner

⇒ 8 possibilites (3 rows, 3 col, 3 diagonals)

```
def check_winner(board):
for i in range(3):
    if board [i] [0] == board [i][1] == board(i)(2) != ''
    return board [i] [0]
Similarly check for Col & diag
```



Step 5:- Player move :- The player should enter row & col index to make a move

```
def player - move (board):

        if board [row][col] = '-";
        board [row] [col] = 'x'
        break

    else
        print ("Try again");
```

Step 6:- Computer move :-

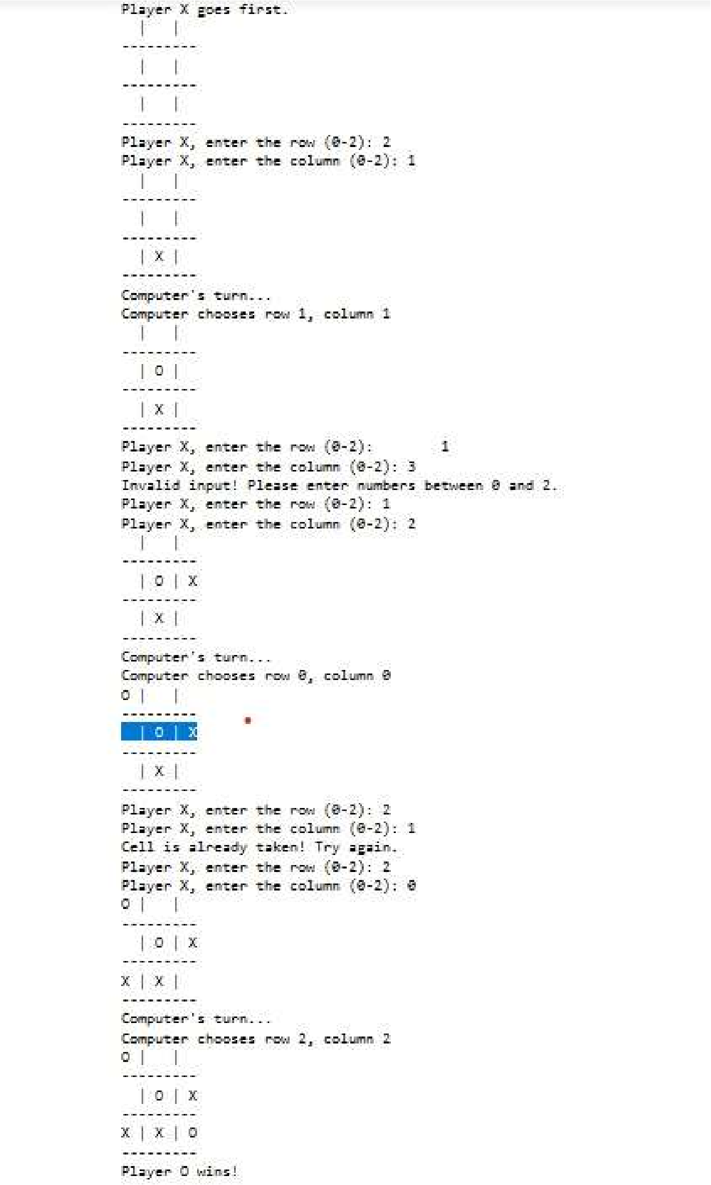
```
    i) Check for winning move
    for i in range (3):
    for j in range (3);
            if board(i) [j] = ='-';
                board [i][j] = = 'O';
            if check- winner (board) = ='O';
                return
```

→ If no winning move, pick a random move such that player cant win.

Code:

```
import random

def win(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 printBoard(board):
    print(“\n”.join([“ | “.join(row) for row in board]))

def draw(board):
    return all(cell != “” for row in board for cell in row)

def user_move(board):
    while True:
        try:
            move = int(input(“Enter your move (1-9): “)) – 1
            row, col = divmod(move, 3)
            if board[row][col] == “”:
                board[row][col] = “X”
                break
            else:
                print(“That space is already taken. Try again.”)
        except (ValueError, IndexError):
            print(“Invalid input. Please enter a number from 1 to 9.”)

def computer_move(board):
    while True:
        move = random.randint(0, 8)
        row, col = divmod(move, 3)
        if board[row][col] == “”:
            board[row][col] = “O”
            break

def _main():
    board = [[“” for _ in range(3)] for _ in range(3)]

    while True:
        printBoard(board)
        user_move(board)
        if win(board):
            printBoard(board)
            print(“You win!”)
            break
        if draw(board):
            printBoard(board)
            print(“It’s a draw!”)
            break
        computer_move(board)
        if win(board):
            printBoard(board)
            print(“Computer wins!”)
            break
        if draw(board):
            printBoard(board)
            print(“It’s a draw!”)
            break

if __name__ == “__main__”:
    _main()
```

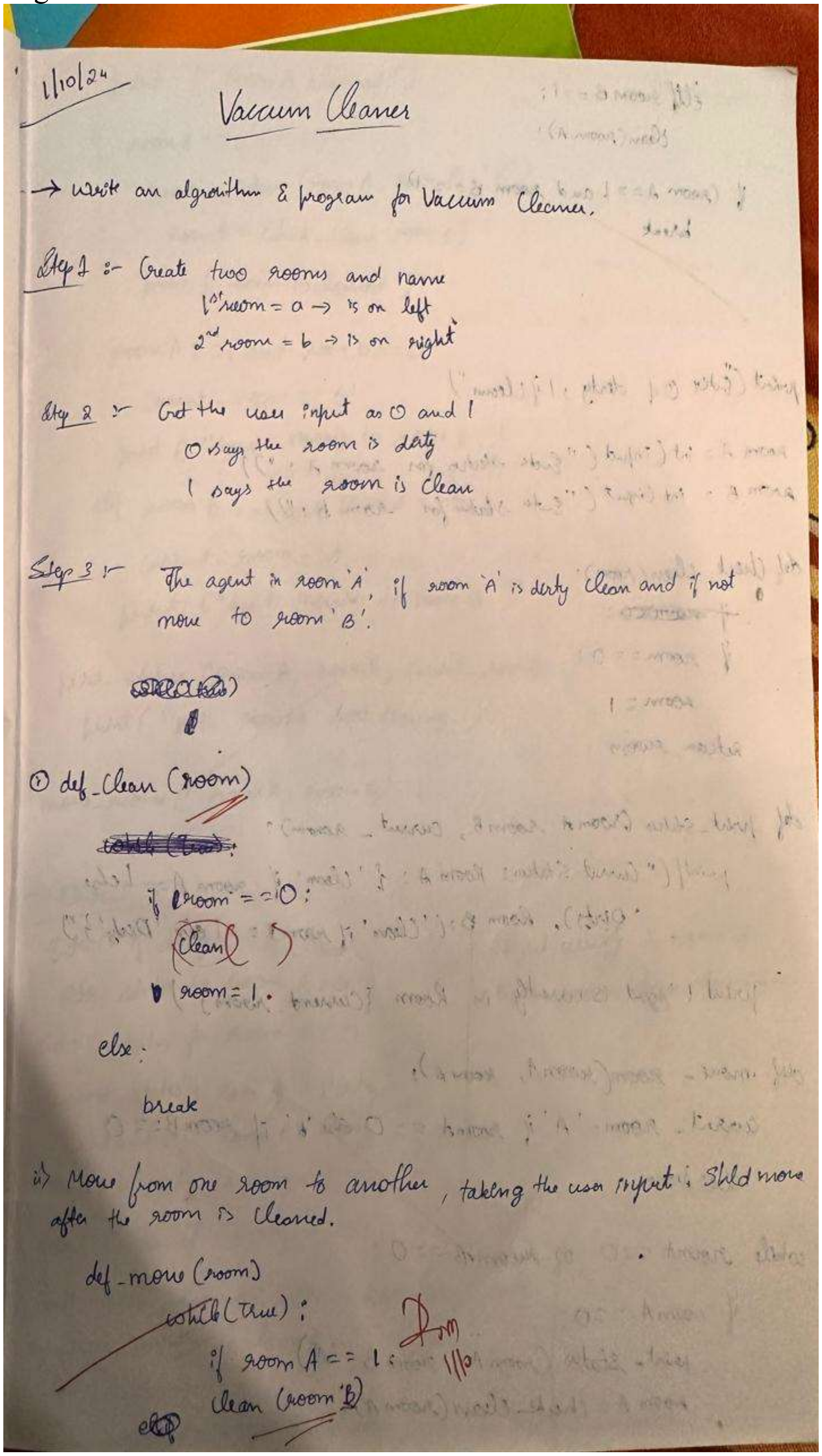
OUTPUT:



```
Player X goes first.
  |   |
---------
  |   |
---------
  |   |
---------
Player X, enter the row (0-2): 2
Player X, enter the column (0-2): 1
  |   |
---------
  |   |
---------
  | X |
---------
Computer's turn...
Computer chooses row 1, column 1
  |   |
---------
  | O |
---------
  | X |
---------
Player X, enter the row (0-2):        1
Player X, enter the column (0-2): 3
Invalid input! Please enter numbers between 0 and 2.
Player X, enter the row (0-2): 1
Player X, enter the column (0-2): 2
  |   |
---------
  | O | X
---------
  | X |
---------
Computer's turn...
Computer chooses row 0, column 0
O |   |
---------
  | O | X
---------
  | X |
---------
Player X, enter the row (0-2): 2
Player X, enter the column (0-2): 1
Cell is already taken! Try again.
Player X, enter the row (0-2): 2
Player X, enter the column (0-2): 0
O |   |
---------
  | O | X
---------
X | X |
---------
Computer's turn...
Computer chooses row 2, column 2
O |   |
---------
  | O | X
---------
X | X | O
---------
Player O wins!
```

LAB 2: Vacuum World

Algorithm:



1/10/24

Vaccum Cleaner

→ write an algorithm & program for Vaccum Cleaner.

Step 1 :- Create two rooms and name
1st room = a → is on left
2nd room = b → is on right

Step 2 :- Get the user input as 0 and 1
0 says the room is dirty
1 says the room is Clean

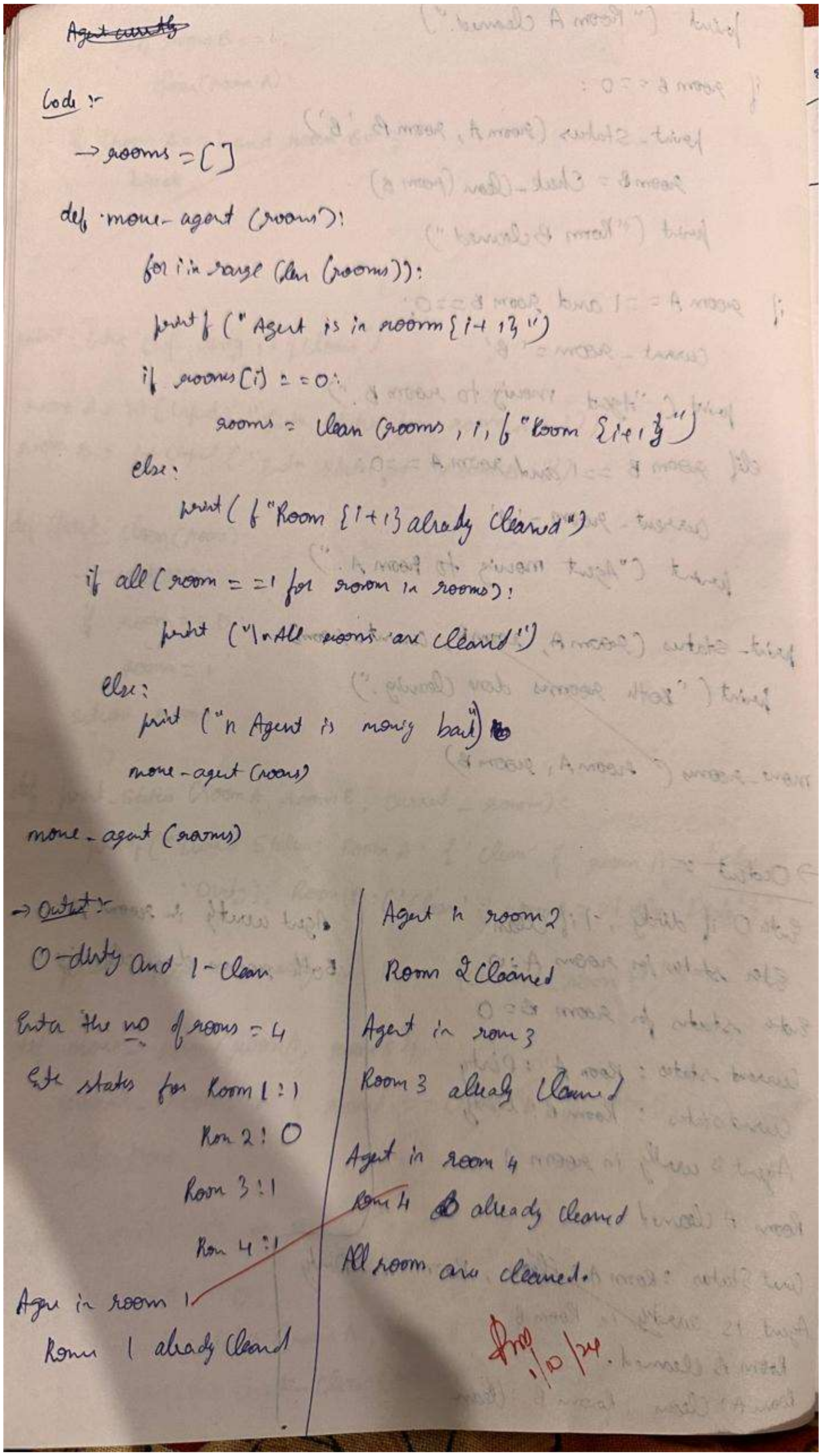
Step 3 :- The agent in room 'A', if room 'A' is derty clean and if not move to room 'B'.

```
① def_Clean (room)
    ~~while (True):~~
        if room = = 0 :
            Clean( )
            room = 1.
    else :
        break
```

ii) Move from one room to another, taking the user input & shld move after the room is cleaned.

```
    def_move (room)
        while(True) :
            if room A = = 1 :
            clean (room 'B')
```

1/10



~~Agent [illegible]~~

Code :-

```
→ rooms = []
def move-agent (rooms):
      for i in range (len (rooms)):
      printf ("Agent is in room {i+1}")
      if rooms [i] = = 0:
            rooms = clean (rooms, i, f"Room {i+1}")
      else:
            print (f"Room {i+1} already cleaned")
   if all (room = = 1 for room in rooms):
         print ("\nAll rooms are cleaned")
   else:
      print ("n Agent is moving back")
      move-agent (rooms)
move-agent (rooms)
```

→ Output :-

```
0-dirty and 1-clean
Enter the no of rooms = 4
Enter status for Room 1 : 1
                 Room 2 : 0
                 Room 3 : 1
                 Room 4 : 1
Agent in room 1
Room 1 already cleaned
```

```
Agent in room 2
Room 2 cleaned
Agent in room 3
Room 3 already cleaned
Agent in room 4
Room 4 already cleaned
All room are cleaned.
```

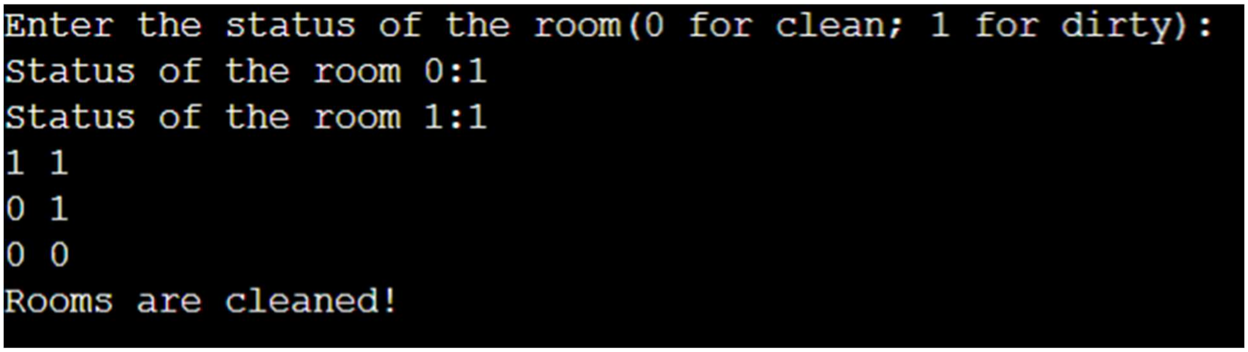
1/10/24.

Code:

For 2 rooms:

```
def printArr(arr):
    n=len(arr)
    print(arr[0],arr[1])

def clean(arr,vac):
    if(arr[vac] == 1):
        arr[vac]=0
    if(arr[vac] == 0):
        return

def check(arr):
    if(arr[0]==0 and arr[1]==0):
        return False
    else:
        return True

print("Enter the status of the room(0 for clean; 1 for dirty):")
arr1 = []
for i in range(0,2):
    a=int(input("Status of the room %d:" %i))
    arr1.append(a)

vac=0
while(True):
    printArr(arr1)
    if(check(arr1) == False):
        break
    clean(arr1,vac)
    if(vac==0):
        vac=1
    else:
        vac=0
print("Rooms are cleaned!")
```

OUTPUT:

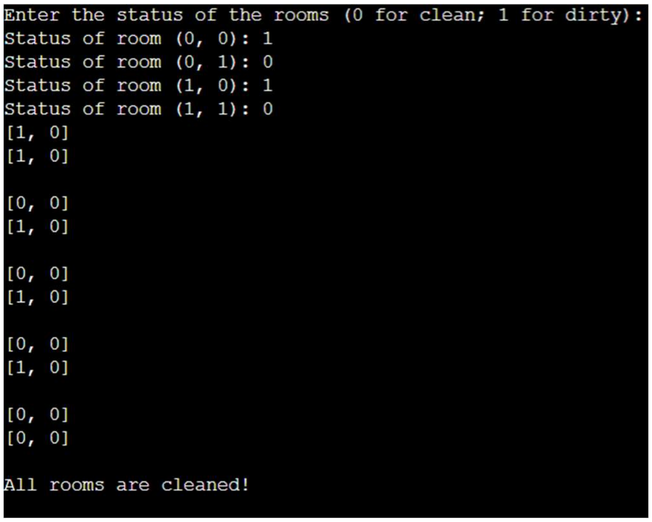


```
Enter the status of the room(0 for clean; 1 for dirty):
Status of the room 0:1
Status of the room 1:1
1 1
0 1
0 0
Rooms are cleaned!
```

for 4 rooms:

```
def printArr(arr):
    for row in arr:
        print(row)
    print()

def clean(arr, x, y):
    if arr[x][y] == 1:
        arr[x][y] = 0

def check(arr):
    for row in arr:
        if 1 in row:
            return True
    return False

# Directions: right (0,1), down (1,0), left (0,-1), up (-1,0)
directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
direction_index = 0  # Start moving right

# Get room status
print("Enter the status of the rooms (0 for clean; 1 for dirty):")
arr1 = []
for i in range(2):
    row = []
    for j in range(2):
        a = int(input(f"Status of room ({i}, {j}): "))
        row.append(a)
    arr1.append(row)

x, y = 0, 0  #Start cleaning from the first room

while True:
    printArr(arr1)
    if not check(arr1):
        break
    clean(arr1, x, y)

    #Move to the next room in the current direction
    dx, dy = directions[direction_index]
    new_x, new_y = x + dx, y + dy

    #Check bounds
    if 0 <= new_x < 2 and 0 <= new_y < 2:
        x, y = new_x, new_y
    else:
        #Change direction(turn right)
        direction_index = (direction_index + 1) % 4
        dx, dy = directions[direction_index]
        x, y = x + dx, y + dy  #Move in the new direction

print("All rooms are cleaned!")
```

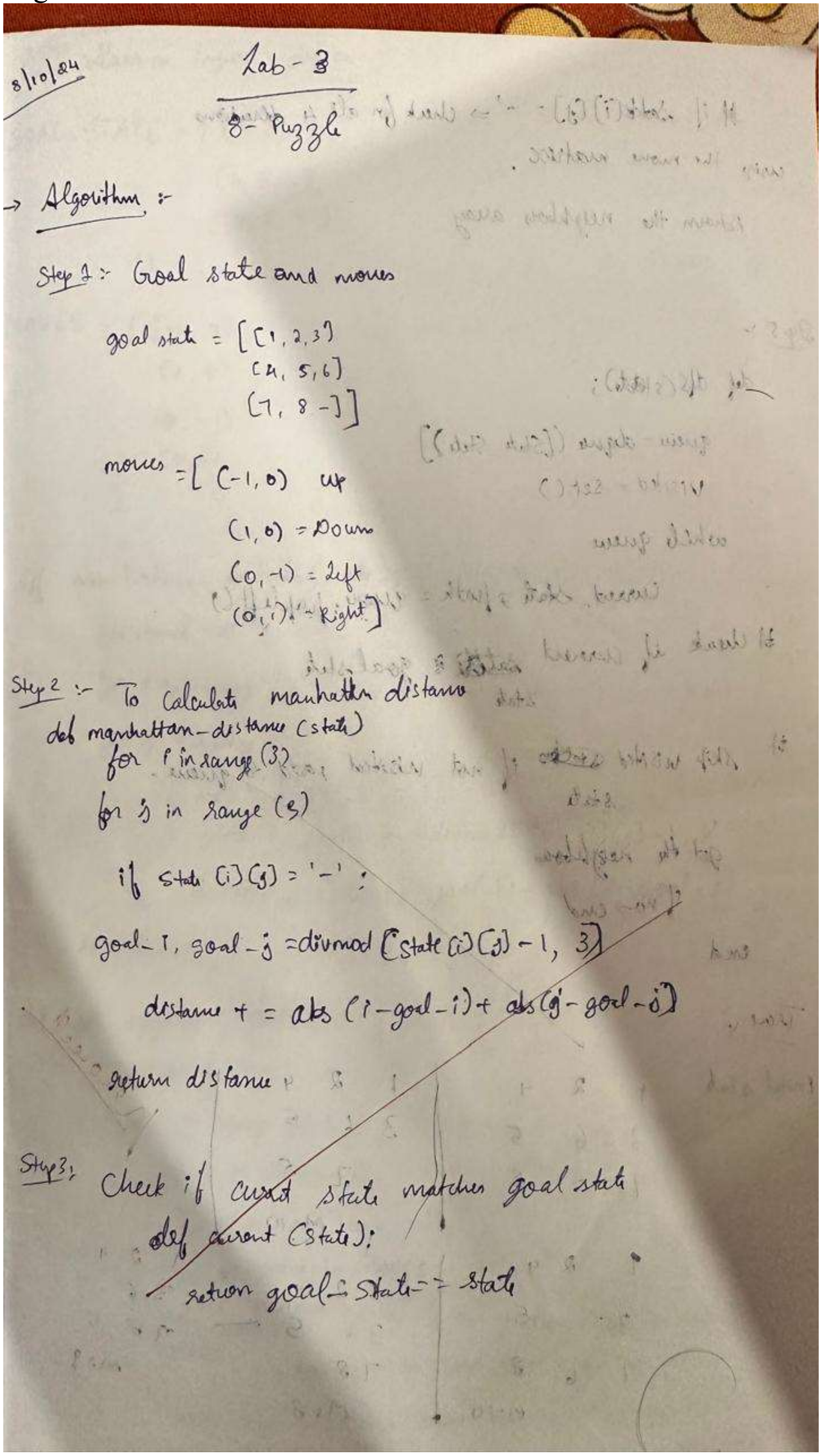
OUTPUT:



```
Enter the status of the rooms (0 for clean; 1 for dirty):
Status of room (0, 0): 1
Status of room (0, 1): 0
Status of room (1, 0): 1
Status of room (1, 1): 0
[1, 0]
[1, 0]

[0, 0]
[1, 0]

[0, 0]
[1, 0]

[0, 0]
[1, 0]

[0, 0]
[0, 0]

All rooms are cleaned!
```

LAB 3: Implement 8 puzzle problems

Algorithm:



8/10/24

Lab - 3

8- Puzzle

→ Algorithm :-

Step 1 :- Goal state and moves

```
goal state = [[1, 2, 3]
              [4, 5, 6]
              [7, 8 -]]

moves = [(-1, 0)  up
         (1, 0) = Down
         (0, -1) = Left
         (0, 1) = Right]
```

Step 2 :- To Calculate manhattan distance

```
def manhattan-distance (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 curnt state matches goal state

```
    def current (state):
       return goal-state == state
```

10

```
if path :
    print ("Solution found :")
    for state in path :
        for row in state :
            print (row)
        print ()
else:
    print (No solution found.")
```

Output :

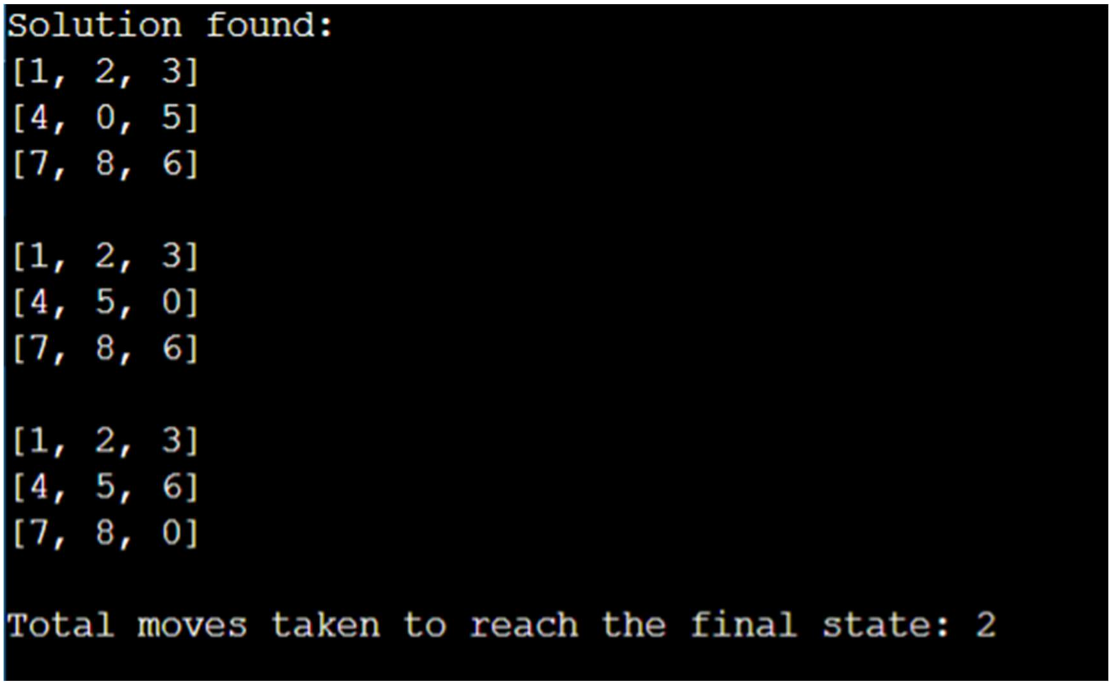
$$\begin{tikzcd}
\begin{bmatrix} 4, & 1, & 3 \\ 7, & 2, & 6 \\ 5, & 8, & - \end{bmatrix} \arrow[r] & \begin{bmatrix} 4, & 1, & 3 \\ 7 & 2 & 6 \\ 5 & - & 8 \end{bmatrix} \arrow[r] & \begin{bmatrix} 4 & 1 & 3 \\ - & 2 & 6 \\ 7 & 5 & 8 \end{bmatrix} \arrow[d] \\
\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & - & 8 \end{bmatrix} \arrow[d] & \begin{bmatrix} 1 & - & 3 \\ 4 & 2 & 6 \\ 7 & 5 & 8 \end{bmatrix} \arrow[l] & \begin{bmatrix} - & 1 & 3 \\ 4 & 2 & 6 \\ 7 & 5 & 8 \end{bmatrix} \arrow[l] \\
\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & - \end{bmatrix} & &
\end{tikzcd}$$

8/10/24

Code:

```
class PuzzleState:
    def __init__(self, board, moves=0, previous=None):
        self.board = board
        self.moves = moves
        self.previous = previous
        self.empty_pos = self.find_empty()

    def find_empty(self):
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return (i, j)

    def manhattan_distance(self):
        dist = 0
        for i in range(3):
            for j in range(3):
                tile = self.board[i][j]
                if tile != 0:
                    target_x = (tile - 1) // 3
                    target_y = (tile - 1) % 3
                    dist += abs(i - target_x) + abs(j - target_y)
        return dist

    def generate_moves(self):
        moves = []
        x, y = self.empty_pos
        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 < 3 and 0 <= new_y < 3:
                new_board = [row[:] for row in self.board]
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y], new_board[x][y]
                moves.append(PuzzleState(new_board, self.moves + 1, self))

        return moves

def dfs(start_board, max_depth):
    stack = [PuzzleState(start_board)]
    visited = set()
    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

    while stack:
        current_state = stack.pop()

        if current_state.board == goal_state:
            return current_state

        visited.add(tuple(map(tuple, current_state.board)))

        if current_state.moves < max_depth:
            for next_state in current_state.generate_moves():
                if tuple(map(tuple, next_state.board)) not in visited:
                    if next_state.manhattan_distance() < 10:
                        stack.append(next_state)
    return None

def print_solution(solution):
    path = []
    while solution:
        path.append(solution.board)
        solution = solution.previous
    for step in reversed(path):
        for row in step:
            print(row)
        print()
    print(f"Total moves taken to reach the final state: {len(path) - 1}")


initial_board = [[1, 2, 3], [4, 0, 5], [7, 8, 6]]
max_depth = 10
solution = dfs(initial_board, max_depth)
if solution:
    print("Solution found:")
    print_solution(solution)
else:
    print("No solution found.")
```

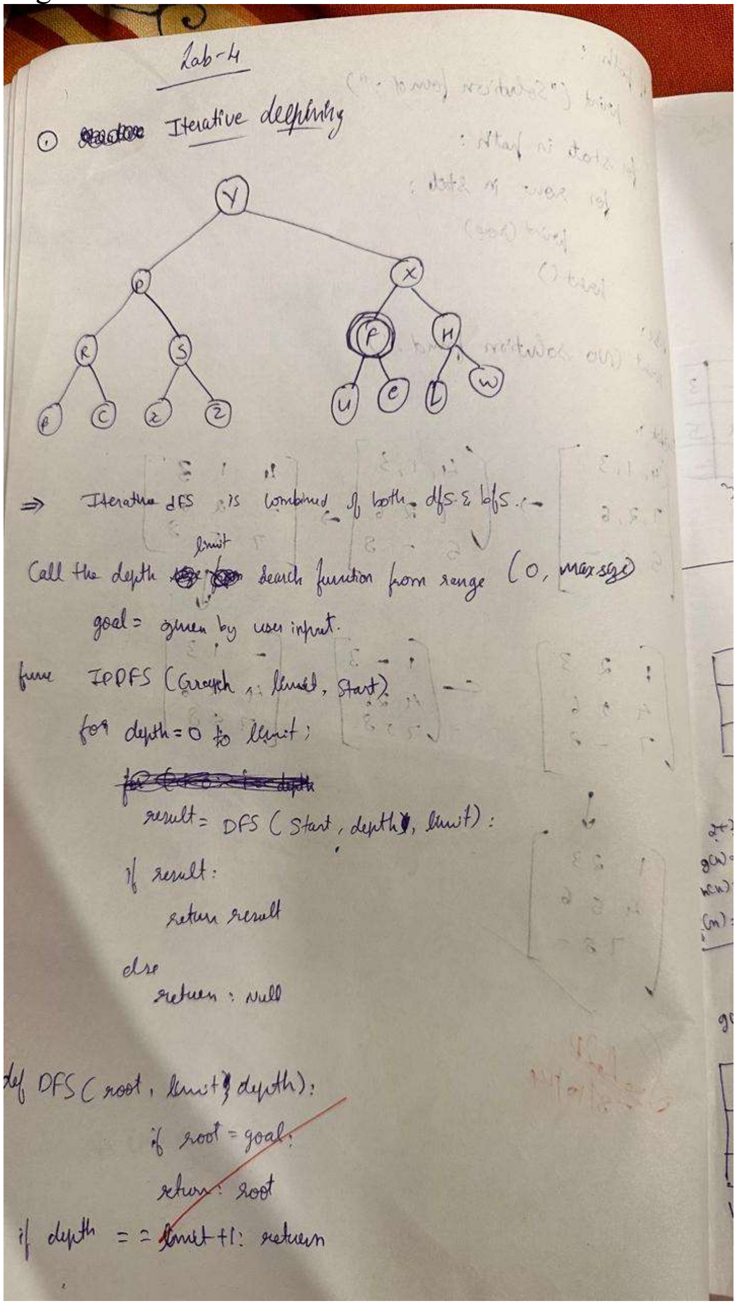
OUTPUT:



```
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:


Y
P
X
R
S
F
H
P
C
Z
Z
U
C
L
W

Lab-4

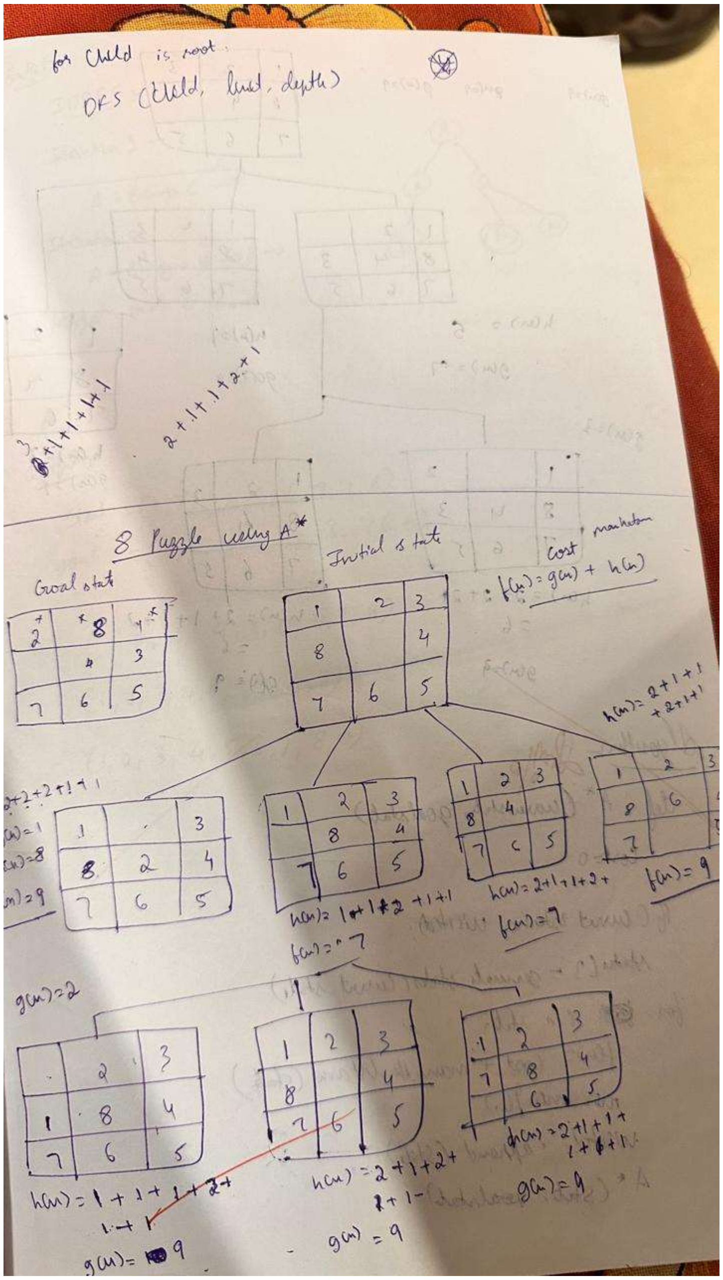
① Iterative deepining

⇒ Iterative dFS is combined of both dfs & bfs.

Call the depth limit search function from range (0, maxsize)

goal = given by user input.

```
func IDDFS (Graph, limit, Start)
    for depth = 0 to limit;
        result = DFS ( Start, depth, limit) :
        if result:
            return result
        else
            return : Null

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



for Child is root:

DFS (Child, limit, depth)

3+1+1+1+1

2+1+1+2+1

8 Puzzle using A*

Goal state

2	8	1
	4	3
7	6	5

Initial state

1	2	3
8		4
7	6	5

Cost manhatan

f(n) = g(n) + h(n)

1		3
8	2	4
7	6	5

2+2+2+1+1

g(n)=1

h(n)=8

f(n)=9

1	2	3
	8	4
7	6	5

h(n)= 1+1+2+1+1

f(n)= 7

1	2	3
8	4	
7	6	5

h(n)=2+1+1+2+

f(n)=7

1	2	3
8	6	
7		

h(n)= 2+1+1+2+1+1

f(n)= 9

g(n)=2

	2	3
1	8	4
7	6	5

h(n)= 1+1+1+2+1+1

g(n)= 9

1	2	3
8		4
7	6	5

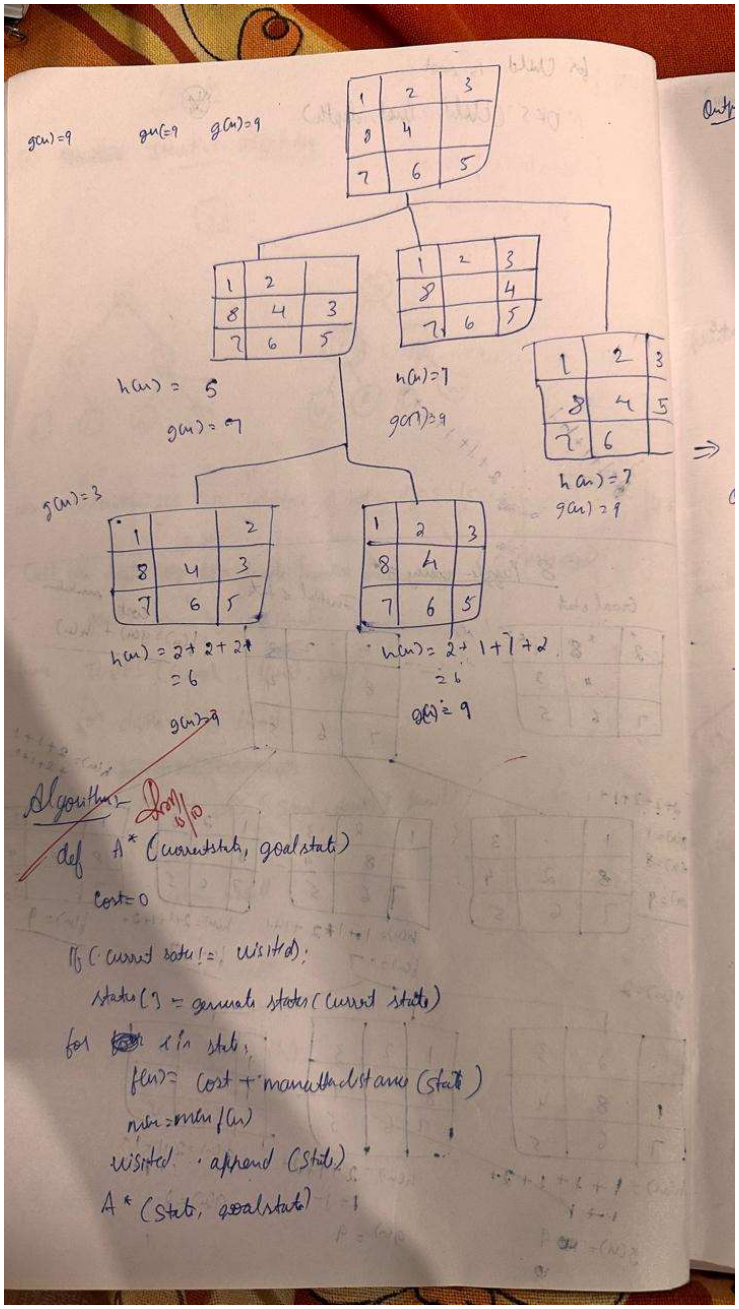
h(n) = 2+1+2+2+1

g(n) = 9

1	2	3
7	8	4
	6	5

h(n) = 2+1+1+1+6+1

g(n) = 9



g(n)=9 gn(=9 g(n)=9

1	2	3
8	4	
7	6	5

1	2	
8	4	3
7	6	5

h(n) = 5

g(n) = 9

1	2	3
8		4
7	6	5

h(n)=7

g(n)=9

1	2	3
8	4	5
7	6	

h(n)=7

g(n)=9

⇒

g(n)=3

1		2
8	4	3
7	6	5

h(n) = 2+2+2
=6

g(n)=9

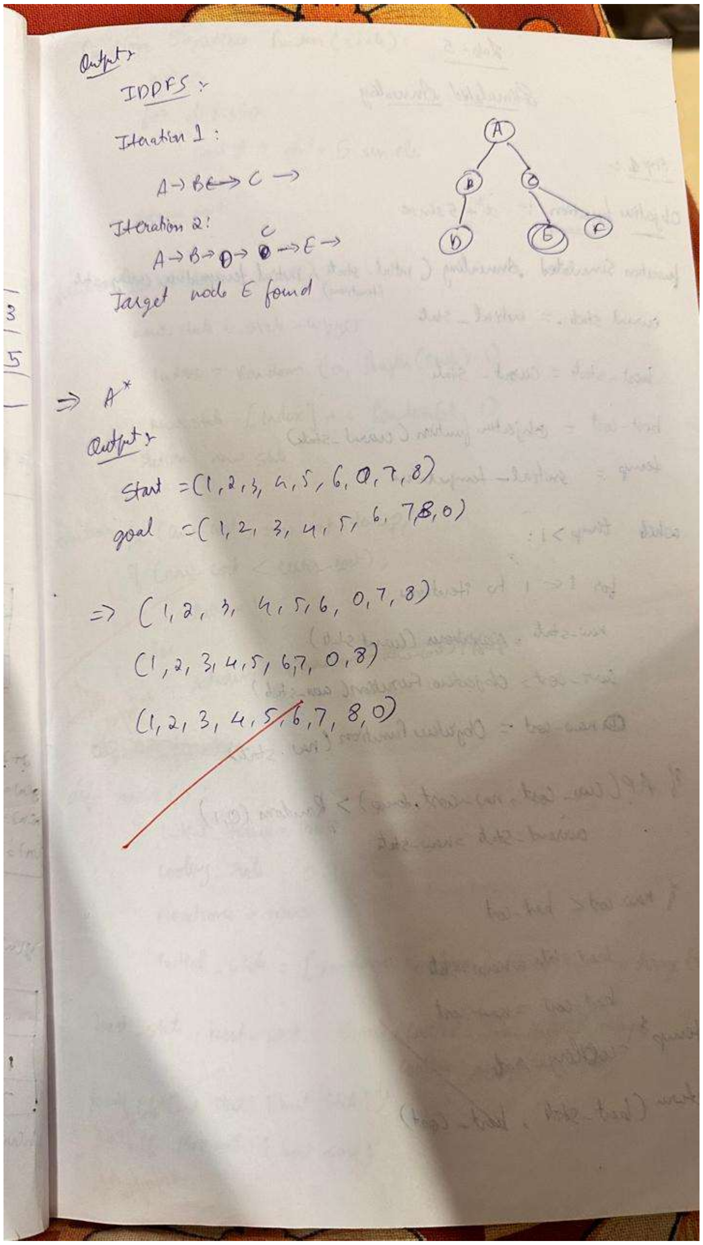
1	2	3
8	4	
7	6	5

h(n)= 2+1+1+2
=6

g(n)= 9

Algorithm:- 15/10

```
def A*(currentstate, goalstate)
    cost=0
    If (current state != visited):
        states[] = generate states(current state)
    for i in state:
        f(n)= cost + manhattandistance(state)
        min=min f(n)
        visited.append(state)
        A*(state, goalstate)
```



Output:

IDDFS:

Iteration 1:

A → B → C →

Iteration 2:

A → B → D → C → E →

Target node E found

⇒ A*

Output:

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

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

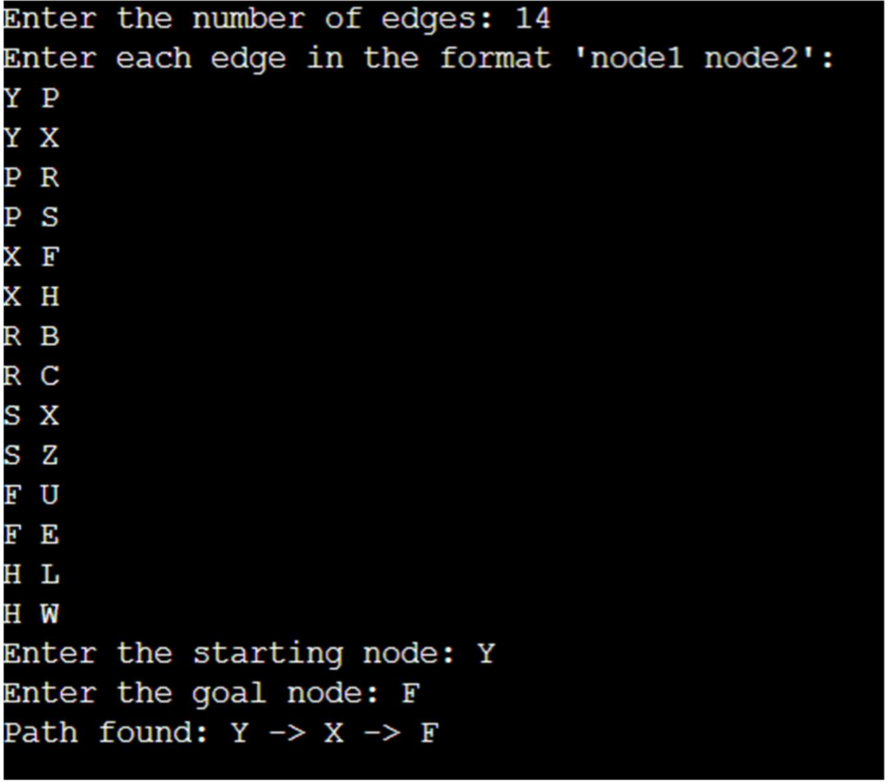
⇒ (1, 2, 3, 4, 5, 6, 0, 7, 8)

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

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

Code:

```
def iterative_deepening_search(graph, start, goal):
    def depth_limited_search(node, goal, depth):
        if depth == 0:
            if node == goal:
                return [node]
            else:
                return None
        elif depth > 0:
            for child in graph.get(node, []):
                result = depth_limited_search(child, goal, depth - 1)
                if result is not None:
                    return [node] + result
        return None
    depth = 0
    while True:
        result = depth_limited_search(start, goal, depth)
        if result is not None:
            return result
        depth += 1
def get_user_input_graph():
    graph = {}
    num_edges = int(input("Enter the number of edges: "))
    print("Enter each edge in the format 'node1 node2':")
    for _ in range(num_edges):
        node1, node2 = input().split()
        if node1 in graph:
            graph[node1].append(node2)
        else:
            graph[node1] = [node2]
        if node2 in graph:
            graph[node2].append(node1)
        else:
            graph[node2] = [node1]
    return graph
def main():
    graph = get_user_input_graph()
    start_node = input("Enter the starting node: ")
    goal_node = input("Enter the goal node: ")
    path = iterative_deepening_search(graph, start_node, goal_node)
    if path:
        print(f"Path found: {' -> '.join(path)}")
    else:
        print("No path found")
if __name__ == "__main__":
    main()
```



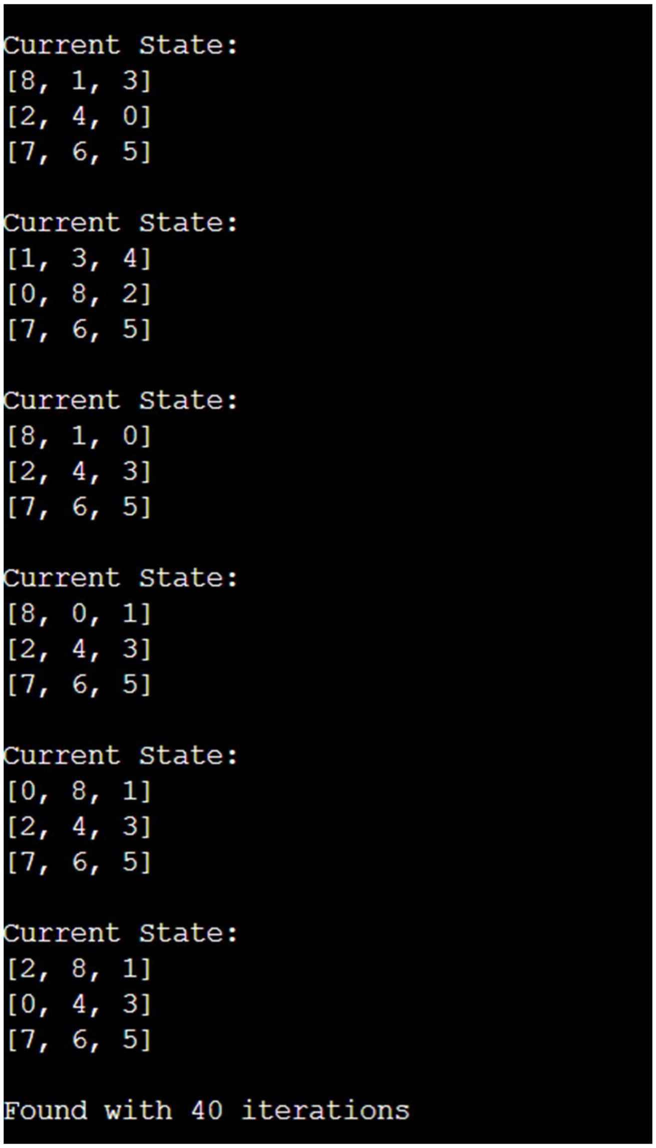
```
Enter the number of edges: 14
Enter each edge in the format 'node1 node2':
Y P
Y X
P R
P S
X F
X H
R B
R C
S X
S Z
F U
F E
H L
H W
Enter the starting node: Y
Enter the goal node: F
Path found: Y -> X -> F
```

PART 2: Implement A* search algorithm

Code:

```
def H_n(state, target):
    return sum(x != y for x, y in zip(state, target))
def F_n(state_with_lvl, target):
    state, lvl = state_with_lvl
    return H_n(state, target) + lvl
def possible_moves(state_with_lvl, visited_states):
    state, lvl = state_with_lvl
    b = state.index(0)
    directions = []
    pos_moves = []
    if b <= 5: directions.append('d')
    if b >= 3: directions.append('u')
    if b % 3 > 0: directions.append('l')
    if b % 3 < 2: directions.append('r')
    for move in directions:
        temp = gen(state, move, b)
        if temp not in visited_states:
            pos_moves.append([temp, lvl + 1])
    return pos_moves
def gen(state, move, b):
    temp = state.copy()
    if move == 'l': temp[b], temp[b - 1] = temp[b - 1], temp[b]
    if move == 'r': temp[b], temp[b + 1] = temp[b + 1], temp[b]
    if move == 'u': temp[b], temp[b - 3] = temp[b - 3], temp[b]
    if move == 'd': temp[b], temp[b + 3] = temp[b + 3], temp[b]
    return temp
def display_state(state):
    print("Current State:")
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()
def astar(src, target):
    arr = [[src, 0]]
    visited_states = []
    iterations = 0
    while arr:
        iterations += 1
        current = min(arr, key=lambda x: F_n(x, target))
        arr.remove(current)
        display_state(current[0])
        if current[0] == target:
            return f'Found with {iterations} iterations'
        visited_states.append(current[0])
        arr.extend(possible_moves(current, visited_states))
    return 'Not found'
src = [1, 2, 3, 8, 0, 4, 7, 6, 5]
target = [2, 8, 1, 0, 4, 3, 7, 6, 5]
print(astar(src, target))
```

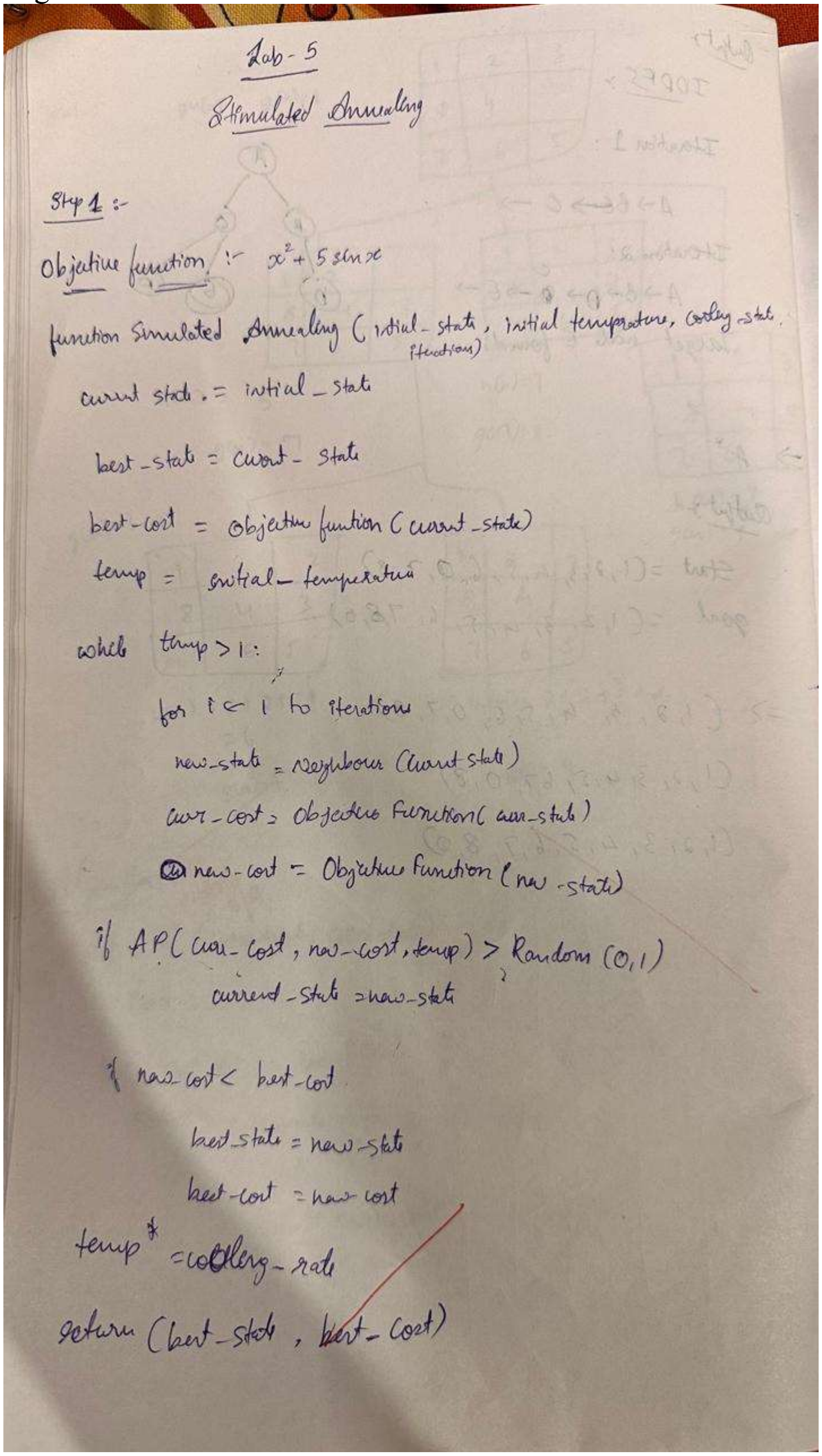
OUTPUT:



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

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

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

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

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

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

Found with 40 iterations
```

LAB 5: Simulated Annealing

Algorithm:



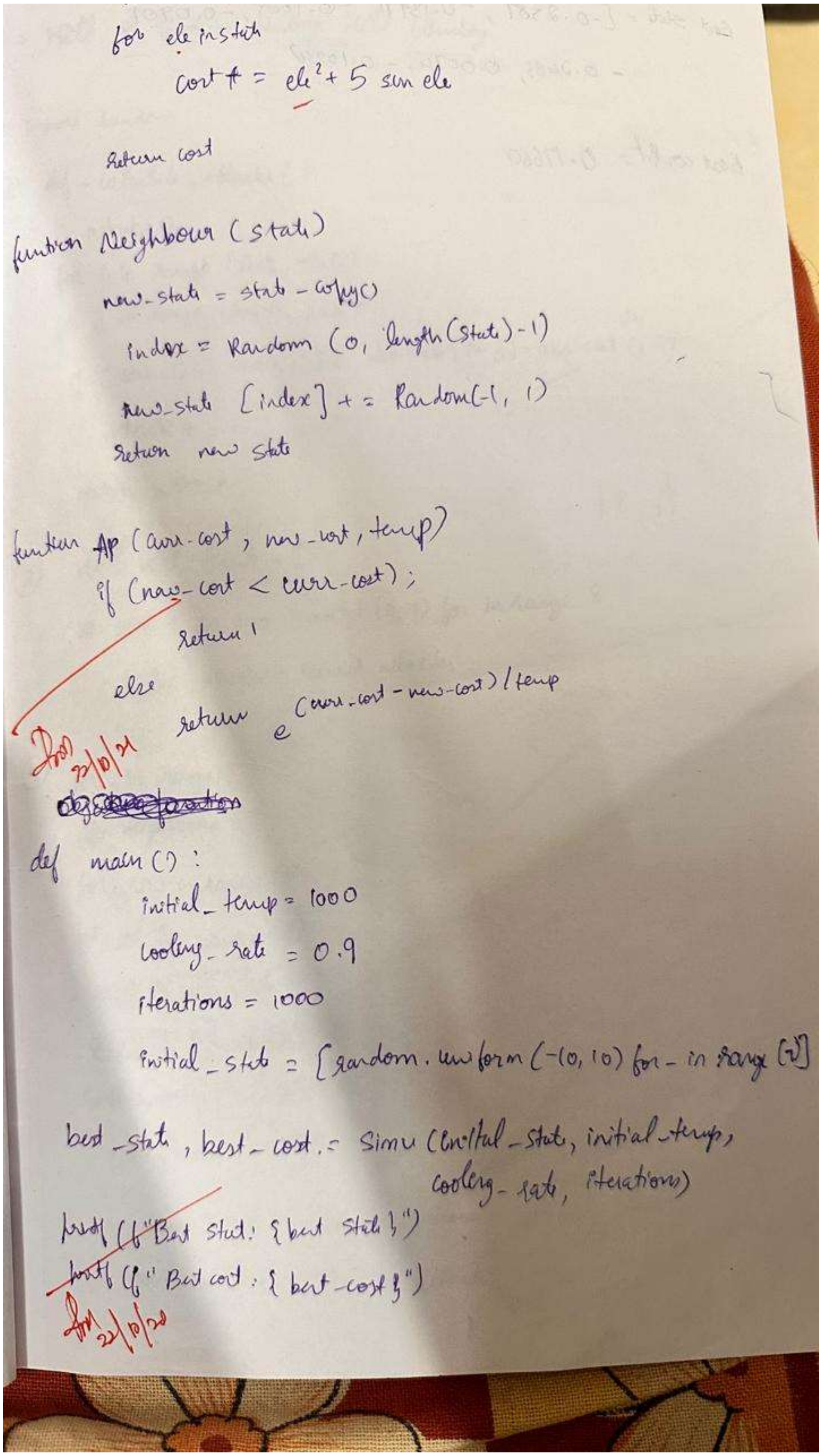
Lab - 5

Simulated Annealing

Step 1 :-

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

```
function Simulated Annealing ( intial_state, initial temprature, cooling_state, iterations)
    current state .= intial_state
    best_state = curent_ state
    best-cost = objective funtion (current_state)
    temp = initial_ temperature
    while temp > 1 :
        for i <- 1 to iterations
            new_state = Neighbour (Current state)
            curr_cost = objective Function( curr_state )
            new-cost = Objective Function (new -state)
    if AP( curr_cost , new_cost, temp) > Random (0,1)
            current_state = new_state
    if new_cost < best_cost
            best_state = new_state
            best-cost = new cost
    temp* = coolling_rate
    return (best_state , best_ cost)
```



```
for ele in state
        cost + = $ele^2$ + 5 sin ele

    Return cost

function Neighbour (state)
    new_state = state - copy()
    index = Random (0, length(State) - 1)
    new_state [index] + = Random(-1, 1)
    return new state

function Ap (curr-cost, new_cost, temp)
    if (new-cost < curr-cost);
        return 1
    else
        return $e^{(curr-cost - new-cost)/temp}$
```

22/10/24

~~objective function~~

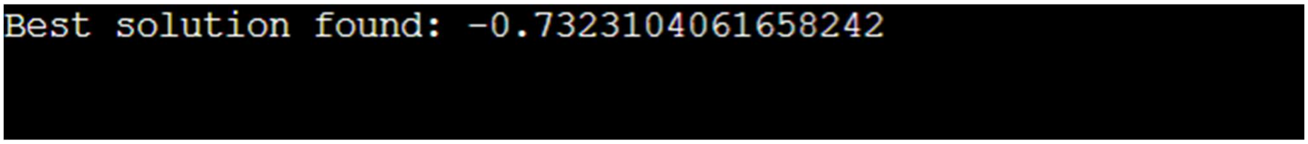
```
def main():
    initial_temp = 1000
    cooling_rate = 0.9
    iterations = 1000
    initial_state = [random.uniform(-10, 10) for _ in range [2]]

best_state, best_cost = Simu (initial_state, initial_temp,
                              cooling_rate, iterations)
print (f"Best State: {best State}")
print (f" Best cost : {best_cost}")
```

22/10/24

Code:

```
import random
import math

def energy(x):
    return x ** 2 + 5 * math.sin(x) + math.exp(-x)

def adaptive_simulated_annealing(start, temp, cooling_rate, lower_limit, upper_limit):
    current = start
    current_energy = energy(current)

    while temp > 1:
        # Adaptive step size based on temperature (larger steps when hot)
        step_size = random.uniform(-1, 1) * temp
        new = current + step_size

        # Ensure new solution is within bounds
        if new < lower_limit or new > upper_limit:
            continue

        new_energy = energy(new)

        # If the new spot is better, move there
        if new_energy < current_energy:
            current = new
            current_energy = new_energy
        else:
            # Acceptance probability (explore worse spots)
            probability = math.exp((current_energy - new_energy) / temp)
            if random.uniform(0, 1) < probability:
                current = new
                current_energy = new_energy

        # Adaptive cooling based on progress
        if abs(new_energy - current_energy) < 0.01:
            temp *= 0.98  # Slow cooling near solution
        else:
            temp *= cooling_rate

    return current

# Run the simulation multiple times from different starting points
best_solution = None
for _ in range(10):  # 10 runs
    result = adaptive_simulated_annealing(start=random.uniform(-10, 10), temp=100, cooling_rate=0.99, lower_limit=-10,
upper_limit=10)
    if best_solution is None or energy(result) < energy(best_solution):
        best_solution = result

print(f"Best solution found: {best_solution}")
```

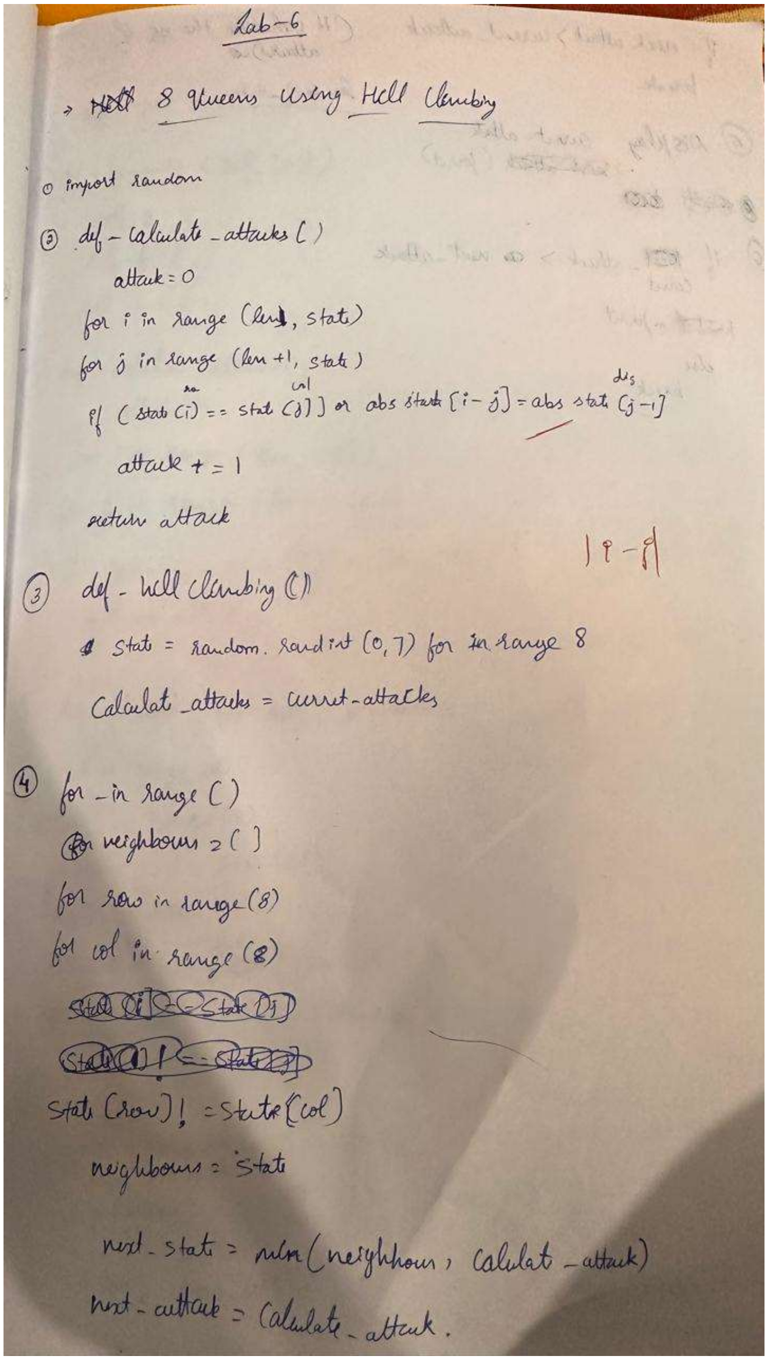
OUTPUT:



```
Best solution found: -0.7323104061658242
```

LAB 6: Implement Hill Climbing

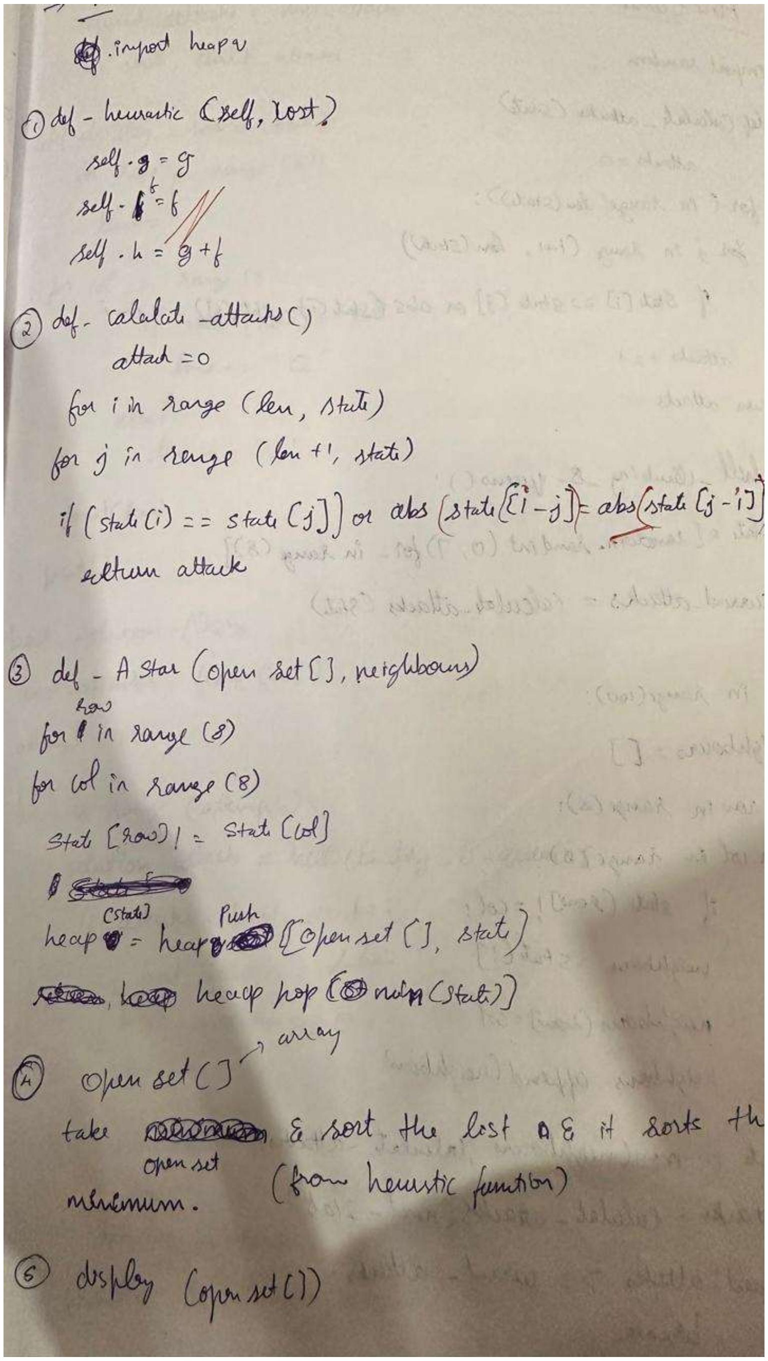
Algorithm:



Lab-6

→ 8 queens using Hill Climbing

```
① import random

② def - calculate _attacks ( )
      attack = 0
   for i in range (len, state)
   for j in range (len +1, state )
                ro          col                                          dig
   if ( state (i) = = state (j) ] or abs state [i - j] = abs state (j -1]
      attack + = 1
   return attack
                                                            |i - j|

③ def - hill climbing ()
     state = random. randint (0, 7) for in range 8
     Calculat _attacks = current-attacks

④ for _in range ( )
   neighbours = ( ]
  for row in range (8)
  for col in range (8)
  state (row)! = state [col)
     neighbours = state

     next _ state = min (neighbour , Calulat _attack)
     next - attack = Calulate _ attack .
```



.import heapq

① def - heuristic (self, lost)
    self . g = g
    self . f = f
    self . h = g + f

② def - calculate -attacks ( )
        attack = 0
    for i in range (len, state)
    for j in range (len +1, state)
    if (state (i) == state [j]) or abs (state [i - j] = abs (state [j - i]
    return attack

③ def - A star (open set [ ], neighbours)
    for row in range (8)
    for col in range (8)
    state [row] != state [col]
    heap (state) = heap push [open set [ ], state)
    heap pop (nilp (state)]

④ open set [ ] → array
    take open set & sort the list & it sorts the
    minimum. (from heuristic function)

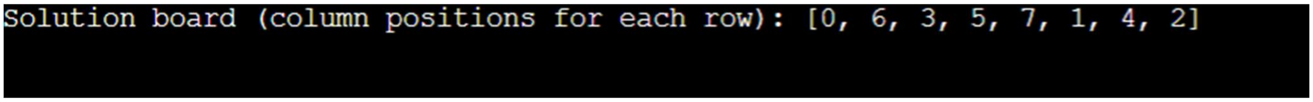
⑤ display (open set [ ])

Code:

```
import random

# Helper function to calculate the heuristic (number of conflicts)
def heuristic(board):
    conflicts = 0
    for i in range(len(board)):
        for j in range(i + 1, len(board)):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                conflicts += 1
    return conflicts

# Hill climbing for 8-queens
```

```
def hill_climbing_8_queens():
    n = 8
    # Generate a random initial state
    board = [random.randint(0, n - 1) for _ in range(n)]

    while True:
        current_h = heuristic(board)
        if current_h == 0:
            return board  # Solution found

        # Find the best neighbor by moving each queen to every other column in its row
        best_board = board[:]
        best_h = current_h
        for row in range(n):
            for col in range(n):
                if col == board[row]:
                    continue
                new_board = board[:]
                new_board[row] = col
                new_h = heuristic(new_board)

                # If the new board has fewer conflicts, update the best board
                if new_h < best_h:
                    best_h = new_h
                    best_board = new_board

        # If no improvement, we're stuck in a local minimum; restart
        if best_h >= current_h:
            board = [random.randint(0, n - 1) for _ in range(n)]
        else:
            board = best_board

# Run hill climbing search
solution = hill_climbing_8_queens()
print("Solution board (column positions for each row):", solution)
```

OUTPUT:



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

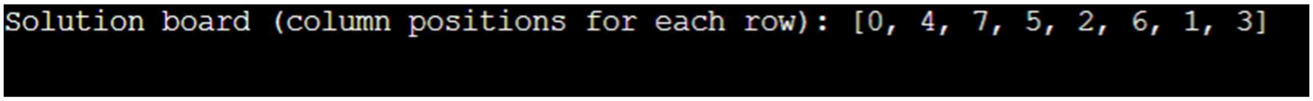
PART 2: Implement A* search algorithm

Code:

```
import heapq

# Helper function to calculate the heuristic (number of conflicts)
def heuristic(board):
    conflicts = 0
    for i in range(len(board)):
        for j in range(i + 1, len(board)):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                conflicts += 1
    return conflicts

# A* Search for 8-queens
def a_star_8_queens():
    n = 8
    open_set = []
    # Initial state: empty board
    heapq.heappush(open_set, (0, []))  # (f, board)

    while open_set:
        f, board = heapq.heappop(open_set)

        # Goal check
        if len(board) == n and heuristic(board) == 0:
            return board

        # Generate successors
        row = len(board)
        for col in range(n):
            new_board = board + [col]
            if heuristic(new_board) == 0:  # No conflicts so far
                g = row + 1
                h = heuristic(new_board)
                heapq.heappush(open_set, (g + h, new_board))


    return None  # No solution found

# Run A* search
```

```
solution = a_star_8_queens()
print("Solution board (column positions for each row):", solution)
```

OUTPUT:

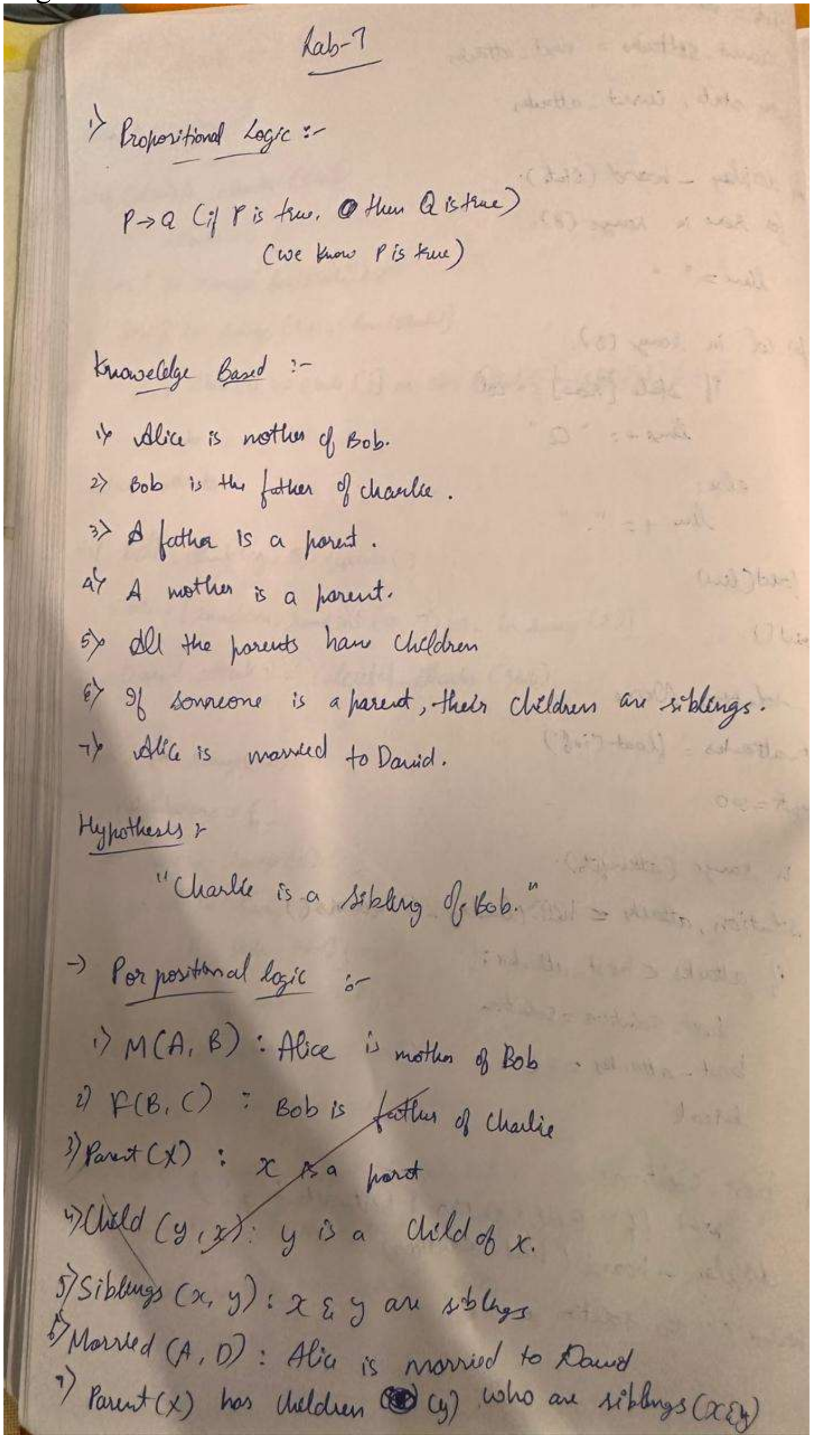


```
Solution board (column positions for each row): [0, 4, 7, 5, 2, 6, 1, 3]
```

LAB 7: Propositional Logic

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

Algorithm:



Lab-7

1) Propositional Logic :-

P→Q (if P is true, then Q is true)
(we know P is true)

Knowelelge Based :-

1) Alice is mother of Bob.
2) Bob is the father of charlie.
3) A father is a parent.
4) A mother is a parent.
5) All the parents have children
6) If someone is a parent, their children are siblings.
7) Alice is married to David.

Hypothesis :-

"Charlie is a sibling of Bob."

→ Porpositional logic :-

1) M(A, B) : Alice is mother of Bob
2) F(B, C) : Bob is father of Charlie
3) Parent (X) : X is a parent
4) Child (y, x) : y is a child of x.
5) Siblings (x, y) : x & y are siblings
6) Married (A, D) : Alice is married to David
7) Parent (X) has children (y) who are siblings (x&y)

Logical Reasoning :-

1) From statement 1 & 4

$M(A,B)$ & (y,x) → Alice is parent

2) From 2 & 4 :

$F(B,C)$ & (y,x) → Bob is a parent

3) From 1 & 2 & 7 :

$M(A,B)$ & $F(B,C)$ & $(x \,\&\, (x, y)) \rightarrow$

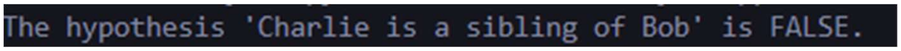
Bob & Charlie are siblings.

11/11/24

Code:

```
import random

# Helper function to calculate the heuristic (number of conflicts)
def heuristic(board):
    conflicts = 0
    for i in range(len(board)):
        for j in range(i + 1, len(board)):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                conflicts += 1
    return conflicts

# Hill climbing for 8-queens
def hill_climbing_8_queens():
    n = 8
    # Generate a random initial state
    board = [random.randint(0, n - 1) for _ in range(n)]

    while True:
        current_h = heuristic(board)
        if current_h == 0:
            return board  # Solution found

        # Find the best neighbor by moving each queen to every other column in its row
        best_board = board[:]
        best_h = current_h
        for row in range(n):
            for col in range(n):
                if col == board[row]:
                    continue
                new_board = board[:]
                new_board[row] = col
                new_h = heuristic(new_board)

                # If the new board has fewer conflicts, update the best board
                if new_h < best_h:
                    best_h = new_h
                    best_board = new_board

        # If no improvement, we're stuck in a local minimum; restart
        if best_h >= current_h:
            board = [random.randint(0, n - 1) for _ in range(n)]
        else:
            board = best_board

# Run hill climbing search
solution = hill_climbing_8_queens()
print("Solution board (column positions for each row):", solution)
```

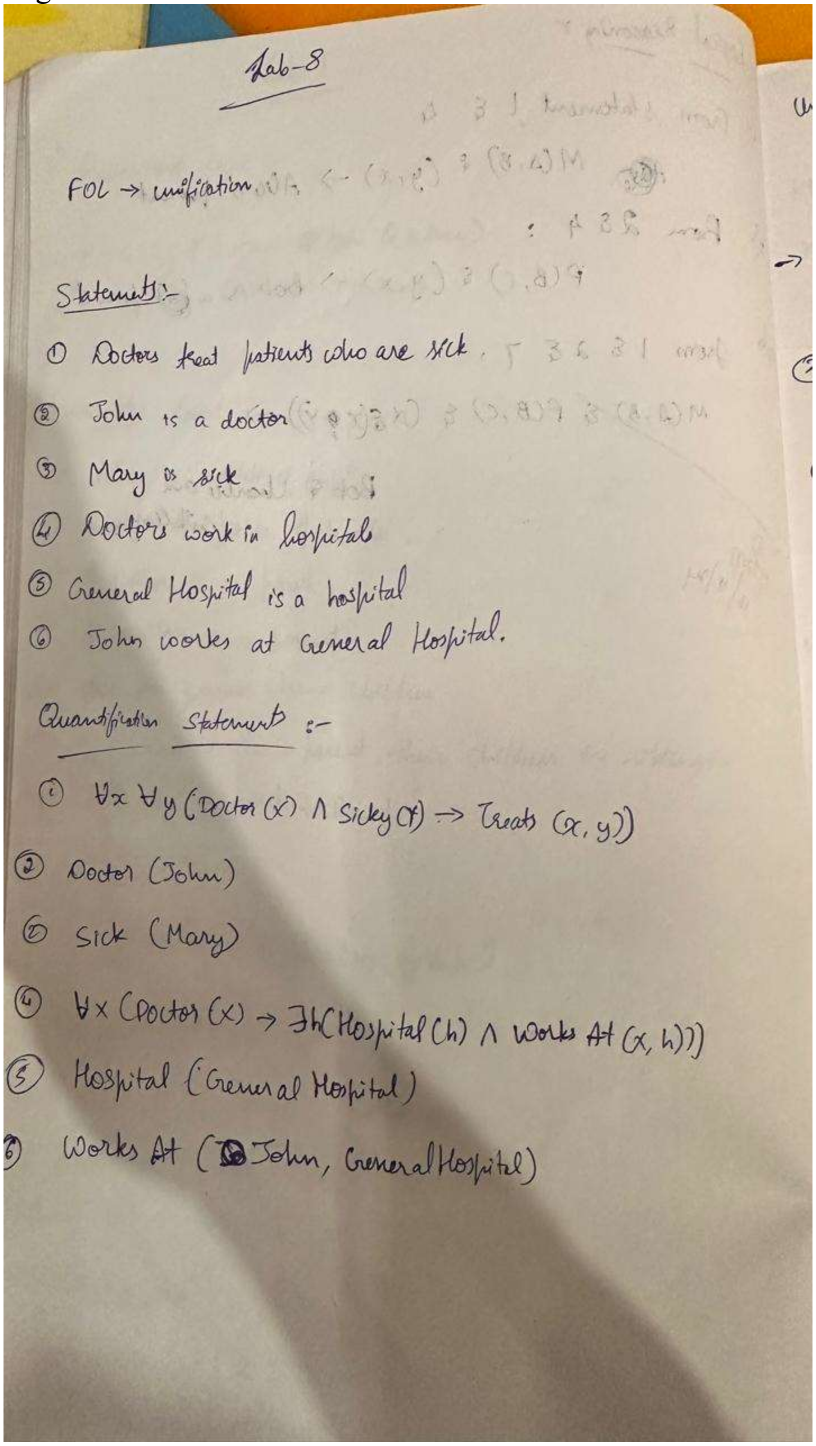
OUTPUT:



```
The hypothesis 'Charlie is a sibling of Bob' is FALSE.
```

LAB 8: Unification in first order logic

Algorithm:



Lab-8

FOL → unification

Statements:-

① Doctors treat patients who are sick.

② John is a doctor

③ Mary is sick

④ Doctors work in hospitals

⑤ General Hospital is a hospital

⑥ John works at General Hospital.

Quantification Statements :-

① $\forall x \forall y (Doctor(x) \land Sicky(y) \rightarrow Treats(x, y))$

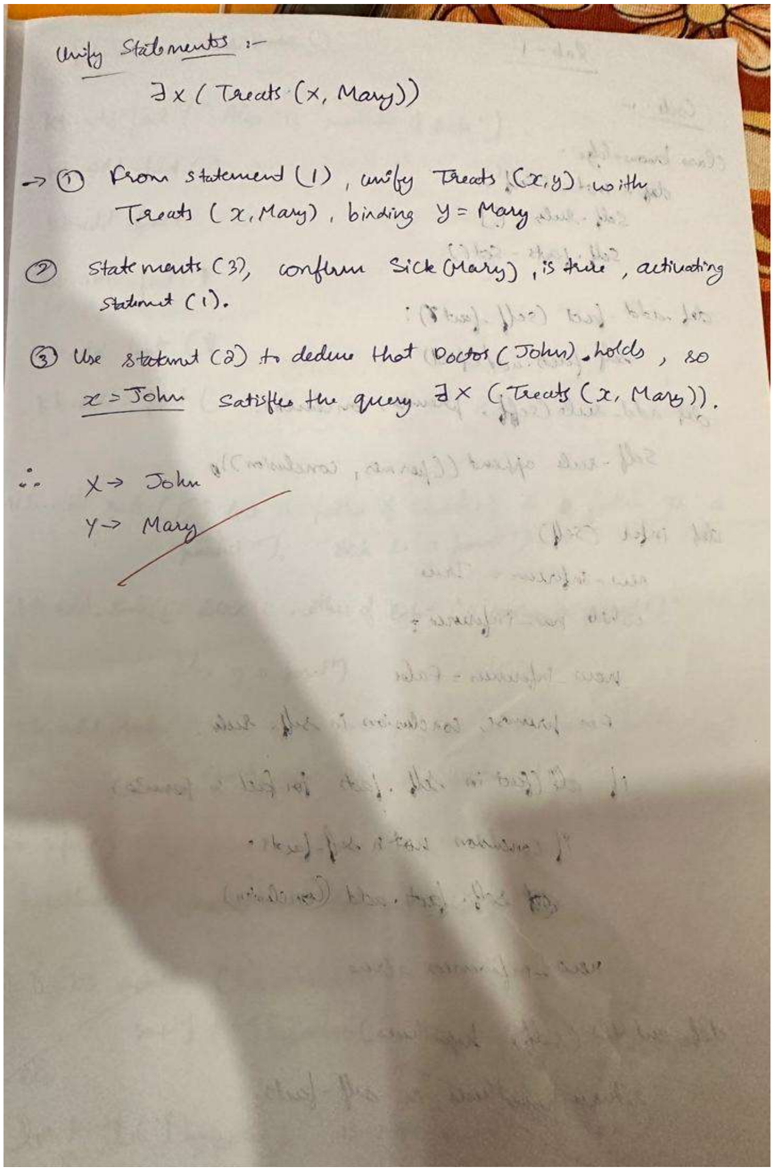
② Doctor (John)

③ Sick (Mary)

④ $\forall x (Doctor(x) \rightarrow \exists h (Hospital(h) \land WorksAt(x, h)))$

⑤ Hospital (General Hospital)

⑥ Works At (John, GeneralHospital)



Unify Statements :-

∃x (Treats (x, Mary))

→ ① From statement (1) , unify Treats (x,y) with Treats (x,Mary) , binding y = Mary

② Statements (3), confirm Sick (Mary) , is true , activating statement (1).

③ Use statement (2) to deduce that Doctor (John) holds , so x = John satisfies the query ∃x (Treats (x, Mary)).

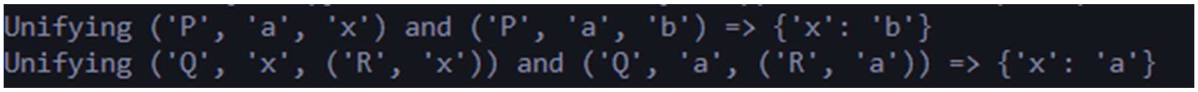
∴ X → John
Y → Mary

Code:

```
def unify(x, y, subst=None):
    """
    Unification Algorithm: Unifies two terms, X and Y.
    """
    if subst is None:
        subst = {}

    if x == y:  # Step 1(a): If X and Y are identical
        return subst
    elif isinstance(x, str) and x.islower():  # Step 1(b): If X is a variable
        return unify_variable(x, y, subst)
    elif isinstance(y, str) and y.islower():  # Step 1(c): If Y is a variable
        return unify_variable(y, x, subst)
    elif isinstance(x, tuple) and isinstance(y, tuple):  # Step 2: Check predicates and arguments
        if x[0] != y[0] or len(x) != len(y):  # Predicate symbol or argument count mismatch
            return None
        for x_i, y_i in zip(x[1:], y[1:]):  # Step 5: Recurse through arguments
```

```
            subst = unify(x_i, y_i, subst)
            if subst is None:
                return None
        return subst
    else:
        return None  # Step 1(d): Failure case


def unify_variable(var, x, subst):
    """
    Unify variable with another term.
    """
    if var in subst:
        return unify(subst[var], x, subst)
    elif occurs_check(var, x, subst):  # Check if var occurs in x
        return None
    else:
        subst[var] = x
        return subst


def occurs_check(var, x, subst):
    """
    Check if a variable occurs in a term.
    """
    if var == x:
        return True
    elif isinstance(x, tuple):
        return any(occurs_check(var, xi, subst) for xi in x)
    elif isinstance(x, str) and x in subst:
        return occurs_check(var, subst[x], subst)
    return False


# Test cases for unification
x1 = ("P", "a", "x")
y1 = ("P", "a", "b")

x2 = ("Q", "x", ("R", "x"))
y2 = ("Q", "a", ("R", "a"))

print("Unifying", x1, "and", y1, "=>", unify(x1, y1))  #
print("Unifying", x2, "and", y2, "=>", unify(x2, y2))
```

OUTPUT:

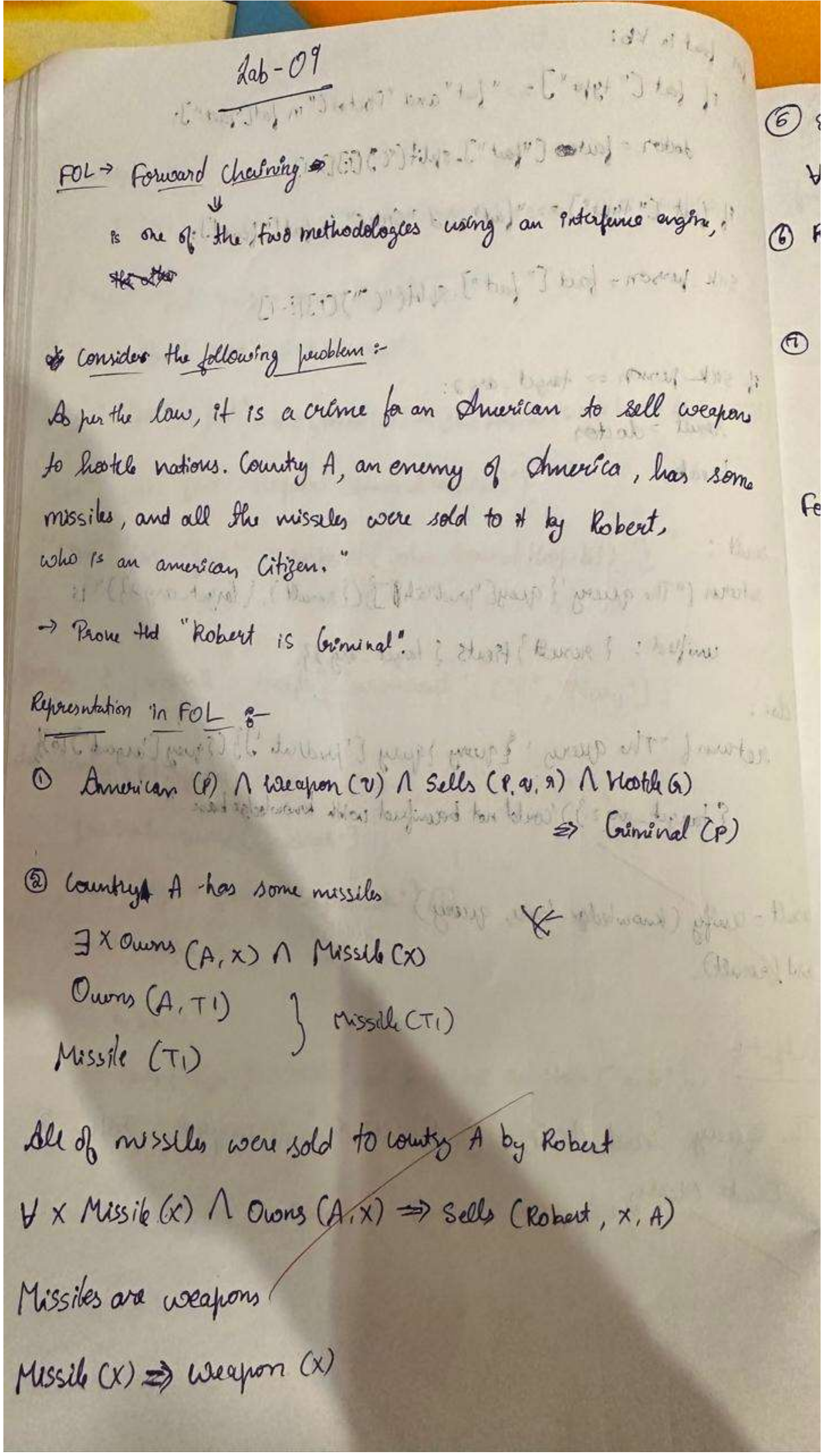


```
Unifying ('P', 'a', 'x') and ('P', 'a', 'b') => {'x': 'b'}
Unifying ('Q', 'x', ('R', 'x')) and ('Q', 'a', ('R', 'a')) => {'x': 'a'}
```

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

FOL → Forward Chaining

is one of the two methodologies using an interfence engine, ~~the other~~

Consider the following problem :-

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".

Representation in FOL :-

① $American(p) \land Weapon(q) \land Sells(p, q, r) \land Hostile(r) \Rightarrow Criminal(p)$

② Country A has some missiles

$\exists x \; Owns(A, x) \land Missile(x)$

$Owns(A, T1)$ } $Missile(T1)$

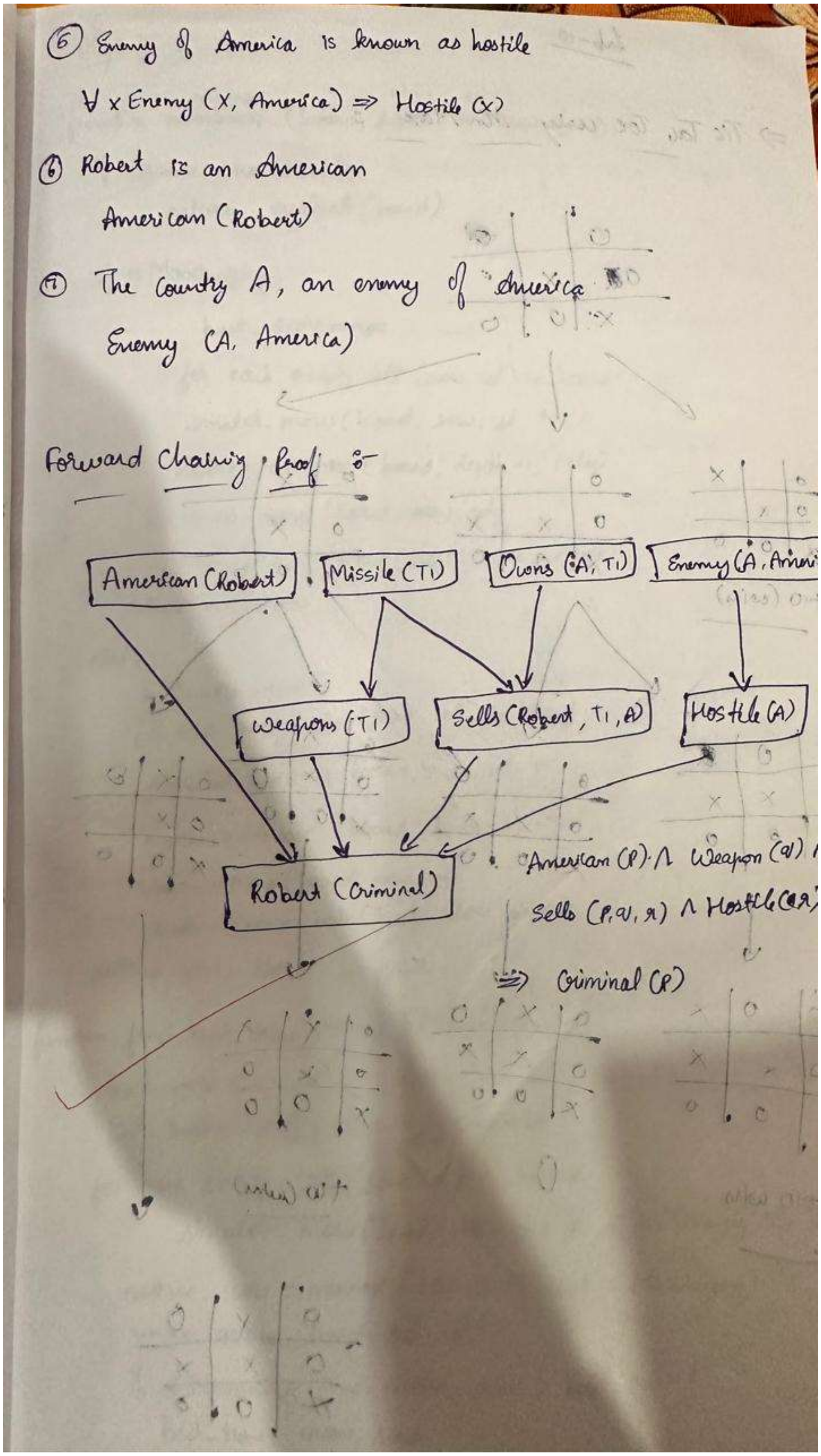
$Missile(T1)$

All of missiles were sold to country A by Robert

$\forall x \; Missile(x) \land Owns(A, x) \Rightarrow Sells(Robert, x, A)$

Missiles are weapons

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



(5) Enemy of America is known as hostile

$\forall x$ Enemy (X, America) ⇒ Hostile (x)

(6) Robert is an American

American (Robert)

(7) The country A, an enemy of America

Enemy (A, America)

Forward Chaining Proof :-

American (Robert)

Missile (T1)

Owns (A, T1)

Enemy (A, Ameri

Weapons (T1)

Sells (Robert, T1, A)

Hostile (A)

Robert (Criminal)

American (P) ∧ Weapon (q) ∧

Sells (P, q, r) ∧ Hostile (r)

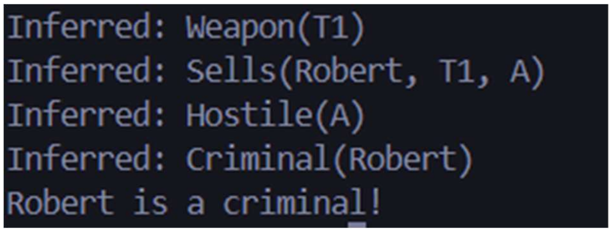
⇒ Criminal (P)

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()
```

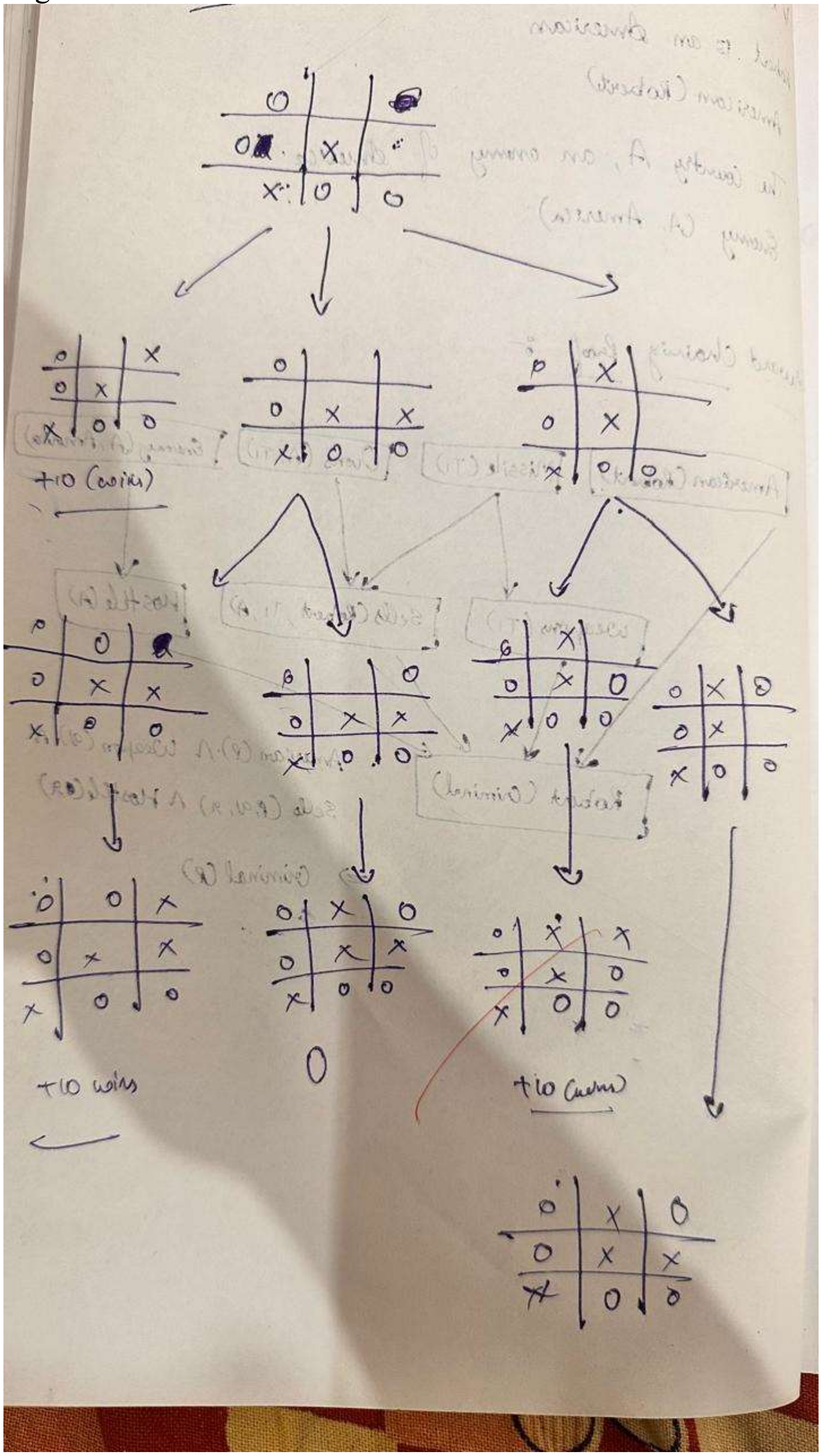
OUTPUT:

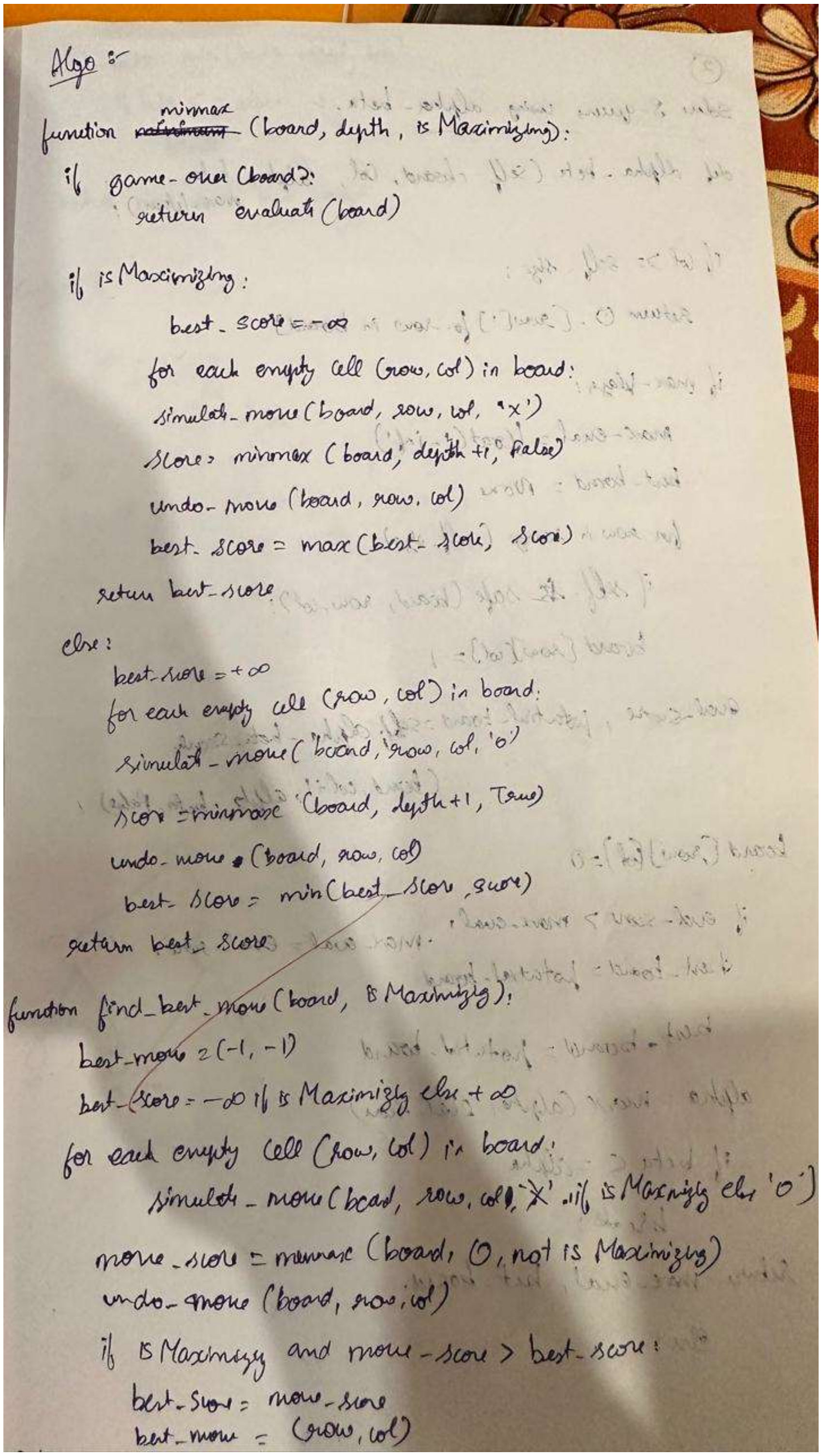


```
Inferred: Weapon(T1)
Inferred: Sells(Robert, T1, A)
Inferred: Hostile(A)
Inferred: Criminal(Robert)
Robert is a criminal!
```

LAB 10: Implement Tic Tac Toe using Min Max

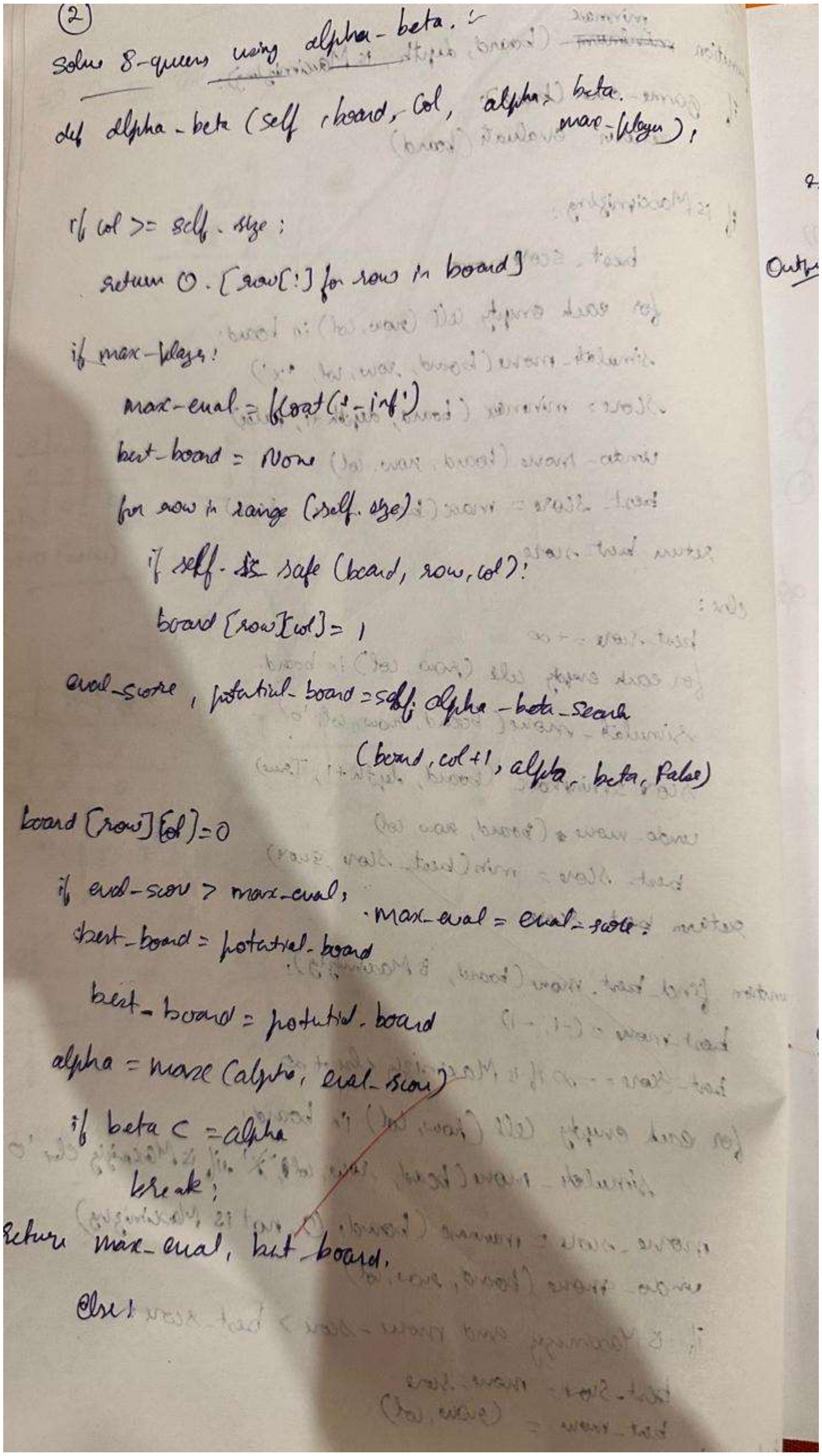
Algorithm:





Algo :-

```
function ~~[illegible]~~ minmax (board, depth, is Maximizing):
    if game-over (board):
        return evaluate (board)

    if is Maximizing:
            best_score = -∞
        for each empty cell (row, col) in board:
        simulate_move (board, row, col, 'x')
        score = minmax (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 = minmax (board, depth+1, True)
        undo_move (board, row, col)
        best_score = min (best_score, score)
    return best_score

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



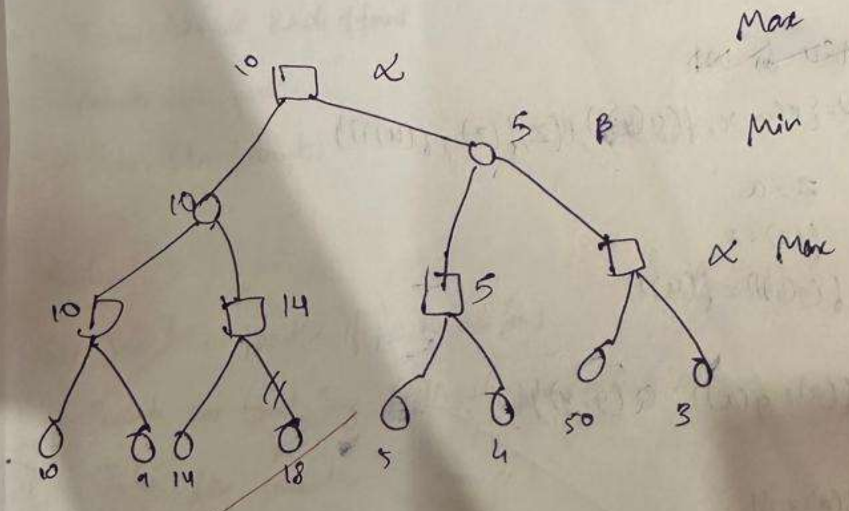
(2)

Solve 8-queens using alpha-beta. :-

```
def alpha_beta (self ,board, col, alpha, beta, max_player);

  if col >= self. size :
    return 0. [row[:] for row in board]

  if max_player:
     max_eval = float('-inf')
     best_board = None
     for row in range (self. size):
        if self. ~~is~~ safe (board, row, col):
        board [row][col] = 1
 eval_score , potential_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 = potential_board
   best_board = potential_board
 alpha = max (alpha, eval_score)
  if beta <= alpha
     break;
Return max_eval, best_board.
   else:
        beta = min(beta, eval_son)
        if beta <= alpha
            break
    return min_eval, best board
```

__Output:__

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

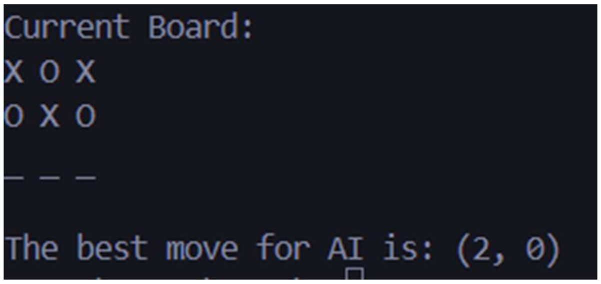


3/12/21

Code:

```
import math

# Constants for the players
AI = 'X'
HUMAN = 'O'
EMPTY = '_'

# Function to print the board
def print_board(board):
    for row in board:
        print(" ".join(row))
    print()

# Function to check if a player has won
def check_winner(board, player):
    # Check rows, columns, and diagonals
    for row in board:
        if all(cell == player for cell in row):
            return True
    for col in range(3):
        if all(row[col] == player for row in board):
            return True
    if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in range(3)):
        return True
    return False

# Function to check if the game is a draw
def is_draw(board):
    return all(cell != EMPTY for row in board for cell in row)

# Minimax algorithm
def minimax(board, depth, is_maximizing):
    if check_winner(board, AI):
        return 10 - depth
    if check_winner(board, HUMAN):
        return depth - 10
    if is_draw(board):
        return 0

    if is_maximizing:
        best_score = -math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == EMPTY:
                    board[i][j] = AI
                    score = minimax(board, depth + 1, False)
                    board[i][j] = EMPTY
                    best_score = max(best_score, score)
        return best_score
    else:
        best_score = math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == EMPTY:
                    board[i][j] = HUMAN
                    score = minimax(board, depth + 1, True)
                    board[i][j] = EMPTY
                    best_score = min(best_score, score)
        return best_score

# Function to find the best move for AI
def find_best_move(board):
    best_score = -math.inf
    move = (-1, -1)
    for i in range(3):
        for j in range(3):
            if board[i][j] == EMPTY:
                board[i][j] = AI
                score = minimax(board, 0, False)
                board[i][j] = EMPTY
                if score > best_score:
                    best_score = score
                    move = (i, j)
    return move

# Example usage
if __name__ == "__main__":
    # Initialize a sample board
    board = [
        ['X', 'O', 'X'],
        ['O', 'X', 'O'],
        ['_', '_', '_']
    ]
    print("Current Board:")
    print_board(board)

    best_move = find_best_move(board)
    print(f"The best move for AI is: {best_move}")
```

OUTPUT:



```
Current Board:
X O X
O X O
_ _ _

The best move for AI is: (2, 0)
```

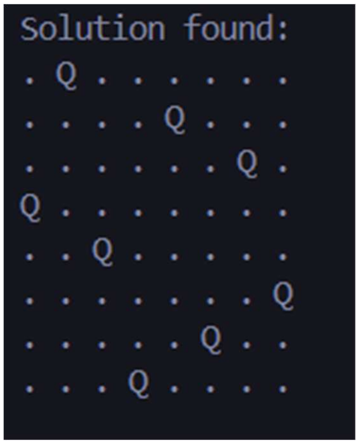
PART 2: Implement Alpha-Beta Pruning

Code:

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

    def is_safe(self, board, row, col):
        """Check if placing a queen at board[row][col] is safe."""
        for i in range(col):
            if board[row][i] == 1:  # Check this row on the left
                return False

        for i, j in zip(range(row, -1, -1), range(col, -1, -1)):  # Check upper diagonal
            if board[i][j] == 1:
                return False

        for i, j in zip(range(row, self.size), range(col, -1, -1)):  # Check lower diagonal
            if board[i][j] == 1:
                return False

        return True

    def alpha_beta_search(self, board, col, alpha, beta, maximizing_player):
        """Alpha-Beta Pruning Search."""
        if col >= self.size:  # If all queens are placed
            return 0, [row[:] for row in board]  # Return 0 as heuristic since it's a valid solution

        if maximizing_player:
            max_eval = float('-inf')
            best_board = None
            for row in range(self.size):
                if self.is_safe(board, row, col):
                    board[row][col] = 1
                    eval_score, potential_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 = potential_board
                    alpha = max(alpha, eval_score)
                    if beta <= alpha:  # Beta cutoff
                        break
            return max_eval, best_board
        else:
            min_eval = float('inf')
            best_board = None
            for row in range(self.size):
                if self.is_safe(board, row, col):
                    board[row][col] = 1
                    eval_score, potential_board = self.alpha_beta_search(board, col + 1, alpha, beta, True)
                    board[row][col] = 0
                    if eval_score < min_eval:
                        min_eval = eval_score
                        best_board = potential_board
                    beta = min(beta, eval_score)
                    if beta <= alpha:  # Alpha cutoff
```

```
                        break
            return min_eval, best_board

    def solve(self):
        """Solve the 8-Queens problem."""
        board = [[0] * self.size for _ in range(self.size)]
        _, solution = self.alpha_beta_search(board, 0, float('-inf'), float('inf'), True)
        return solution

    def print_board(self, board):
        """Print the chessboard."""
        for row in board:
            print(" ".join("Q" if col else "." for col in row))
        print()


if __name__ == "__main__":
    game = EightQueens()
    solution = game.solve()
    if solution:
        print("Solution found:")
        game.print_board(solution)
    else:
        print("No solution exists.")
```

OUTPUT:



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