VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

NEHAL A K(1BM22CS176)

in partial fulfilment 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 NEHAL A K**(1BM22CS176),** who is Bonafide student of **B.M.S. College of Engineering.** It is in partial fulfilment 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.

Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

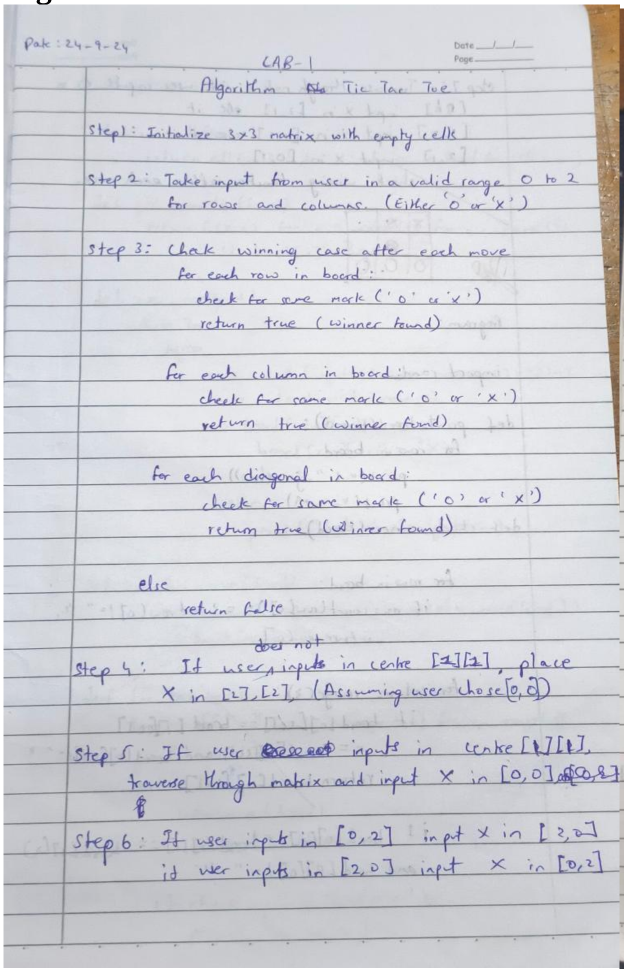
Github Link:

https://github.com/nehxll31/AI_LAB

Program 1

Implement Tic –Tac –Toe Game

Algorithm:



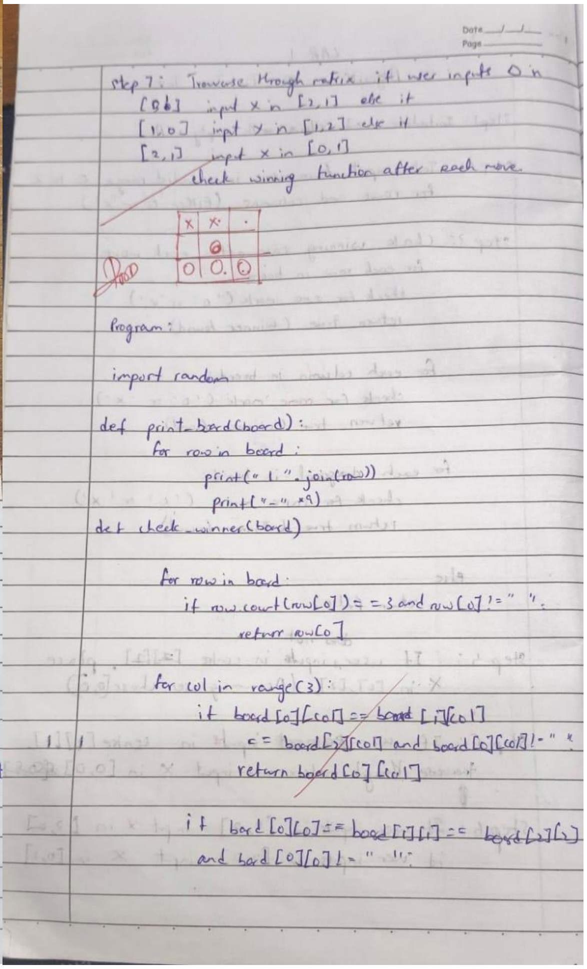
Pak : 24-9-24

Date __/__/__
Page ____

LAB-1

Algorithm ~~for~~ Tic Tac Toe

```
Step1: Initialize 3x3 matrix with empty cells

Step 2: Take input from user in a valid range 0 to 2
        for rows and columns. (Either 'O' or 'X')

Step 3: Check winning case after each move
        for each row in board:
            check for same mark ('O' or 'X')
            return true (winner found)

        for each column in board:
            check for same mark ('O' or 'X')
            return true (winner found)

        for each diagonal in board:
            check for same mark ('O' or 'X')
            return true (winner found)

    else
        return false

Step 4: If user does not inputs in centre [1][1], place
        X in [2],[2], (Assuming user chose [0,0])

Step 5: If user ~~[illegible]~~ inputs in centre [1][1],
        traverse through matrix and input X in [0,0] ~~or~~ [0,2]

Step 6: If user inputs in [0,2] input X in [2,0]
        if user inputs in [2,0] input X in [0,2]
```



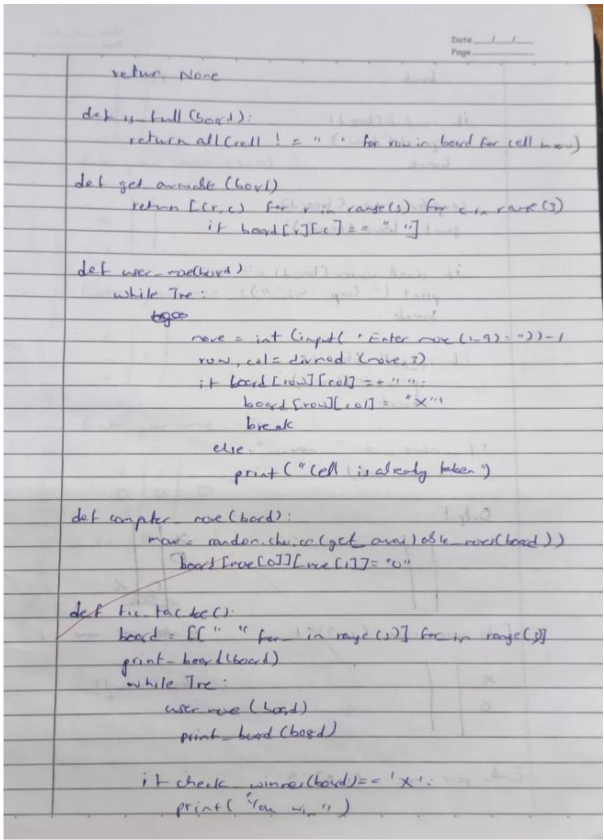
Date __/__/__
Page ____

```
Step 7: Traverse through matrix if user inputs O in
    [0,0] input X in [2,1] else if
    [1,0] input X in [1,2] else if
    [2,1] input X in [0,1]
        check winning function after each move.
```

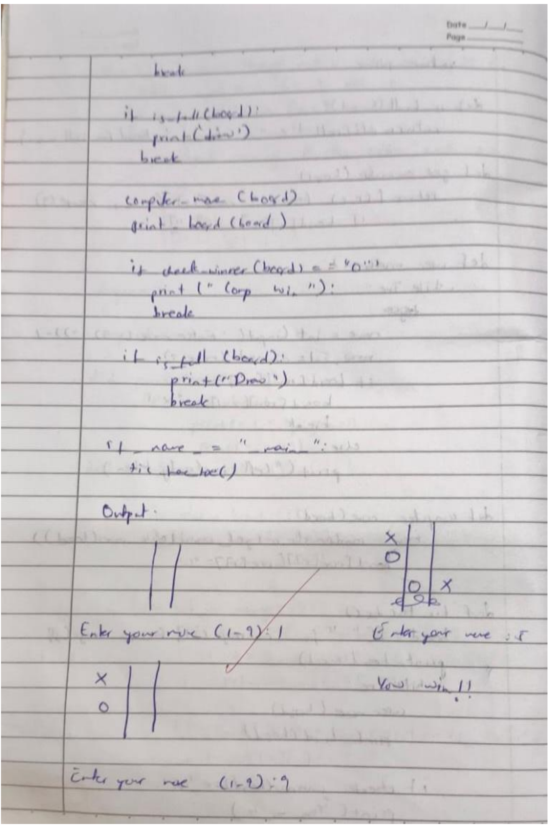
X	X	.
	O	
O	O.	O

Program:

```
import random

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

    for row in board:
        if row.count(row[0]) = = 3 and row[0] != " ":
            return row[0]

    for col in range(3):
        if board[0][col] == board [1][col]
            == board[2][col] and board[0][col] != " ":
        return board[0] [col]

    if bord [0][0] == bood [1][1] == bord[2][2]
        and bord [0][0] != " ":
```



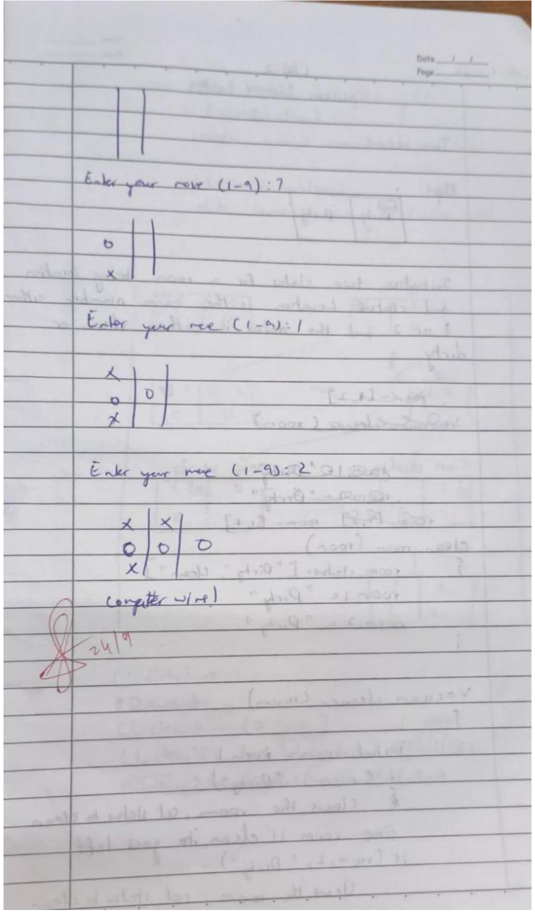
```
    retur None

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

def get avaiable (bord)
        retun [(r,c)  for r in range(3)  for c in range(3)
                if board[r][c] == " "]

def user_move(board)
    while Tre :
        ~~logo~~
            move = int (input( 'Enter move (1-9): "))-1
            row, col= divmod (move, 3)
            if board[row][col] == " ":
                board[row][col] = "X"'
                break
            else:
                print("Cell is already taken")

def computer_move(bord):
        move= random.choice(get_avaiable_moves(board))
            board[move[0]][move[1]]= "O"

def tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]
    print_board(board)
    while Tre :
        user_move (board)
        print_board (board)

        if check_winner(board)== 'X':
            print( "You win")
```



```
            break

        if is_full(board):
            print('draw')
          break

        computer_move (board)
        print_board (board)

        if check_winner (board) == "O":
            print (" Comp wins "):
            break

        if is_full (board):
            print("Draw")
            break

if _name_ == "_main_":
    tic_tac_toe()
```

Output:

```
  | |

Enter your move (1-9): 1

X | |
O | |

Enter your move (1-9): 9

X | |
O | |
  |O| X

Enter your move : 5

You win !!
```



```
  | |

Enter your move (1-9): 7

O | |
X | |

Enter your move (1-9): 1

X | |
O |O|
X | |

Enter your move (1-9): 2

X |X|
O |O| O
X | |

computer wins!
```

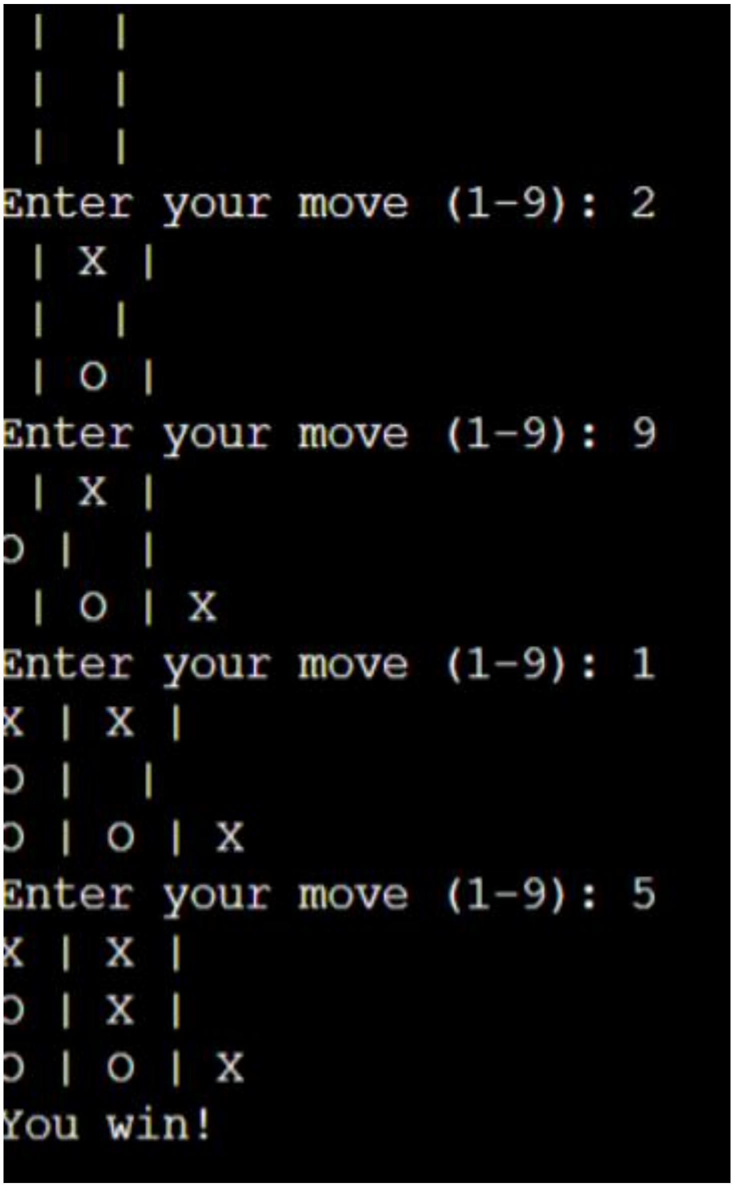
24/9

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:

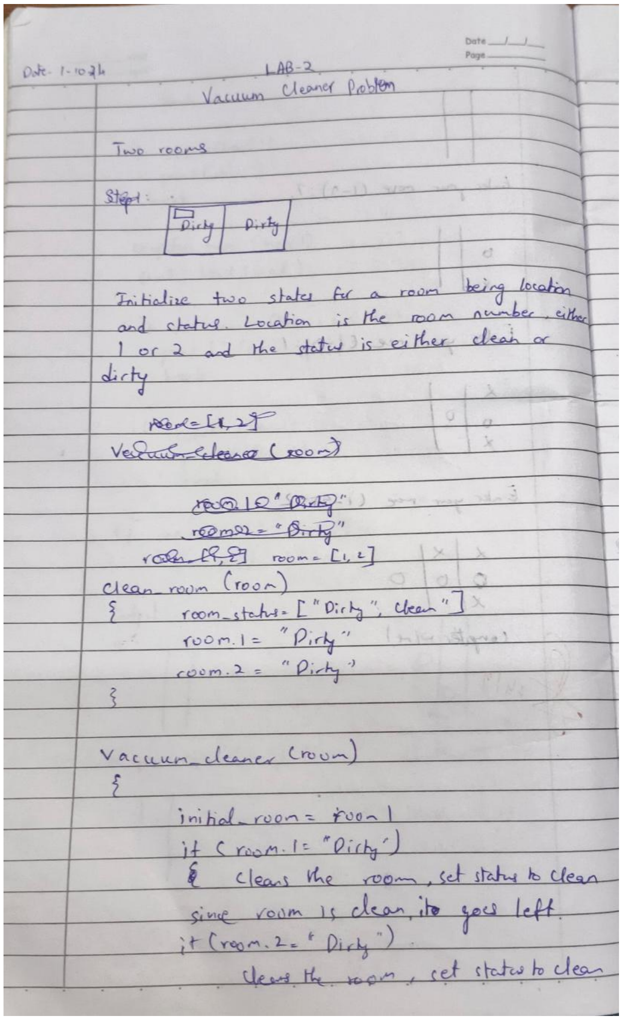


```
  |   |
  |   |
  |   |
Enter your move (1-9): 2
  | X |
  |   |
  | O |
Enter your move (1-9): 9
  | X |
O |   |
  | O | X
Enter your move (1-9): 1
X | X |
O |   |
O | O | X
Enter your move (1-9): 5
X | X |
O | X |
O | O | X
You win!
```

Program 2

Implement vacuum cleaner agent

Algorithm:



Date. 1-10-24

LAB-2

Vacuum Cleaner Problem

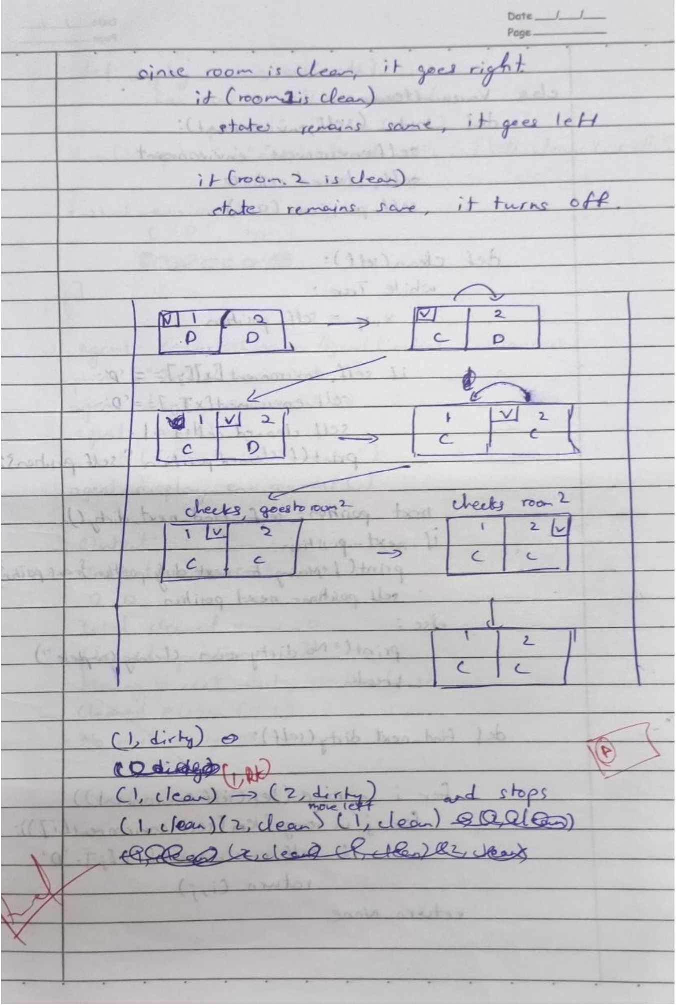
Two rooms

Step1:

Dirty	Dirty

Initialize two states for a room being location and status. Location is the room number, either 1 or 2 and the status is either clean or dirty

```
    ~~room=[1,2]~~
~~Vacuum_cleaner (room)~~

        ~~room.1= "Dirty"~~
        ~~room.2 = "Dirty"~~
    ~~room [1,2]~~ room= [1, 2]
clean_room (room)
{       room_status= ["Dirty", Clean"]
        room.1= "Dirty"
        room.2 = "Dirty"
}

Vacuum_cleaner (room)
{
        initial_room= room1
        if (room.1= "Dirty")
        { Cleans the room, set status to clean
        since room is clean, it goes left.
        if (room.2= "Dirty")
            Cleans the room, set status to clean
```



```
    since room is clean, it goes right.
        if (room.1 is clean)
          states remains same, it goes left

        if (room.2 is clean)
          state remains same, it turns off.
```

☑ 1	2
D	D

→

☑	2
C	D

1	☑ 2
C	D

→

1	☑ 2
C	C

checks, goes to room 2

1 ☑	2
C	C

→

checks room 2

1	2 ☑
C	C

1	2
C	C

(1, dirty)

~~(2, dirty)~~ (1, R)

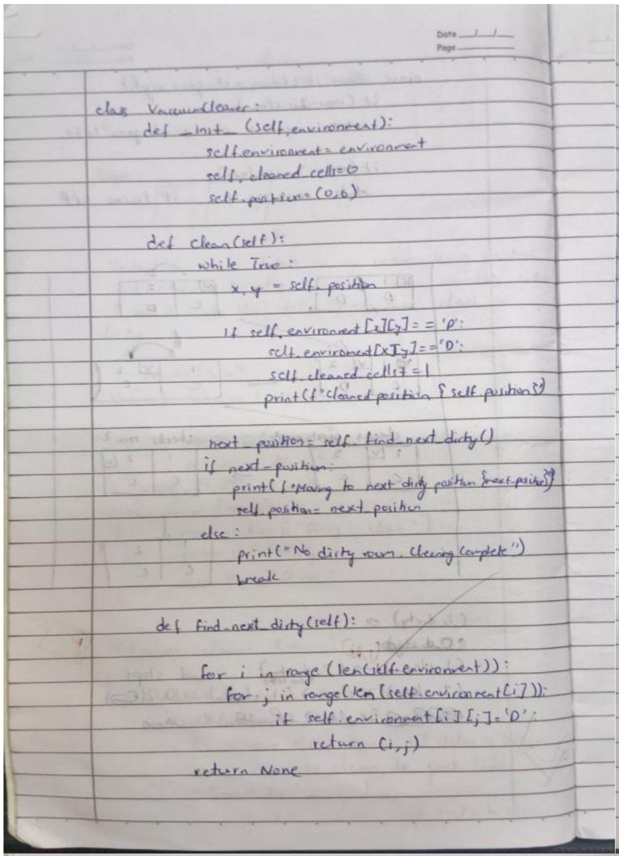
(1, clean) → (2, dirty) and stops

move left

(1, clean)(2, clean) (1, clean) ~~(2, clean)~~

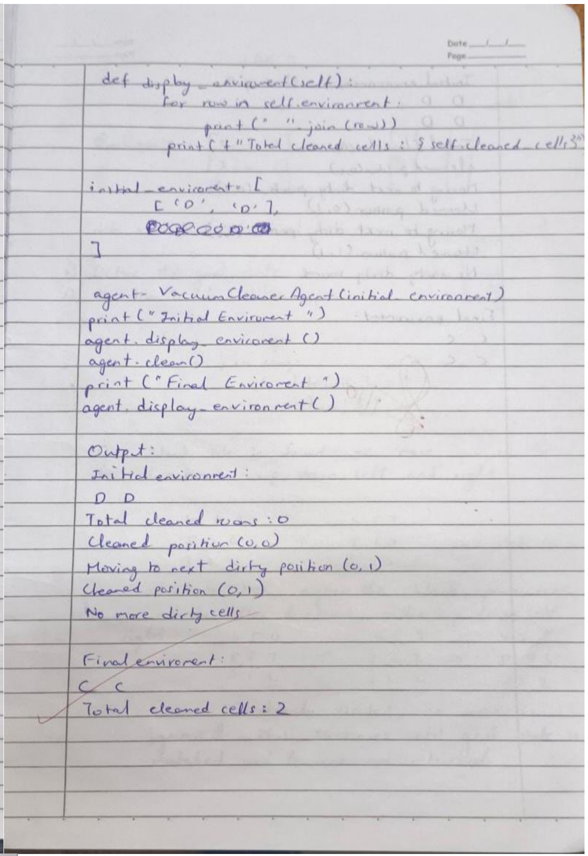
~~(1, clean) (2, clean) (1, clean) (2, clean)~~

A



Date / /
Page

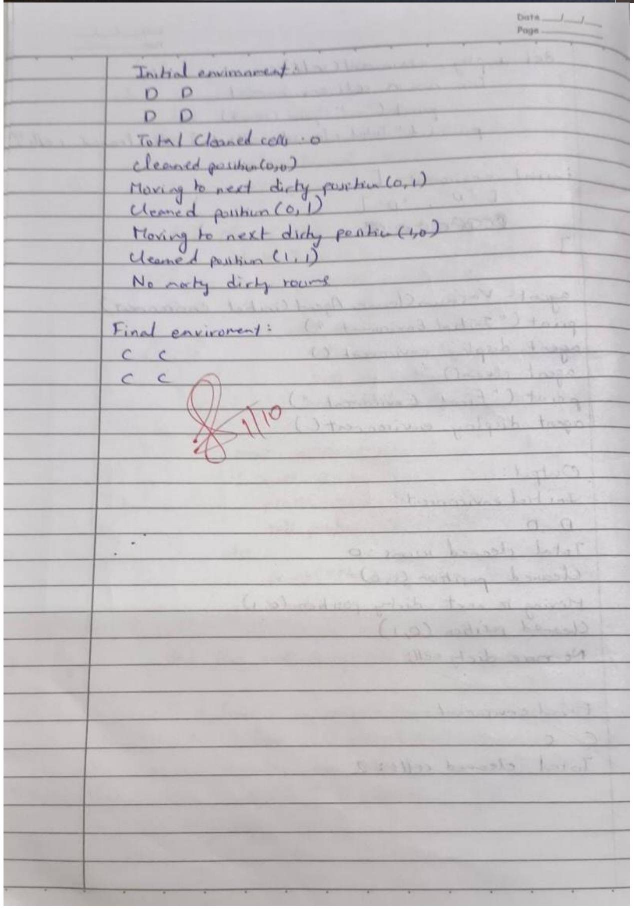
```
class VacuumCleaner:
    def _init_ (self,environment):
            self.environment= environment
            self.cleaned cells=0
            self.position=(0,0)

    def clean(self):
         while True:
              x,y = self.position

              if self.environment[x][y]= = 'D':
                   self.environment[x][y]=='D':
                   self.cleaned cells+=1
                   print(f"cleaned position {self.position}")

            next_position= self.find_next_dirty()
            if next-position:
                 print(f"Moving to next dirty position {next-position}")
                 self.position= next position
            else:
                 print("No dirty room. Cleaning Complete")
                 break

    def find_next_dirty(self):

          for i in range (len(self.environment)):
               for j in range(len(self.environment[i])):
                    if self.environment[i][j]= 'D':
                         return (i,j)
          return None
```



Date / /
Page

```
def display_environment(self):
        for row in self.environment:
             print(" ".join(row))
        print(f"Total cleaned cells : {self.cleaned_cells}")

initial_environment= [
        ['D', 'D'],
        [illegible]
]

agent= VacuumCleanerAgent(initial_environment)
print("Initial Environment ")
agent.display_environment()
agent.clean()
print("Final Environment ")
agent.display_environment()
```

```
Output:
Initial environment:
 D  D
Total cleaned rooms :0
Cleaned position (0,0)
Moving to next dirty position (0,1)
Cleaned position (0,1)
No more dirty cells

Final environment:
C  C
Total cleaned cells: 2
```



Date / /
Page

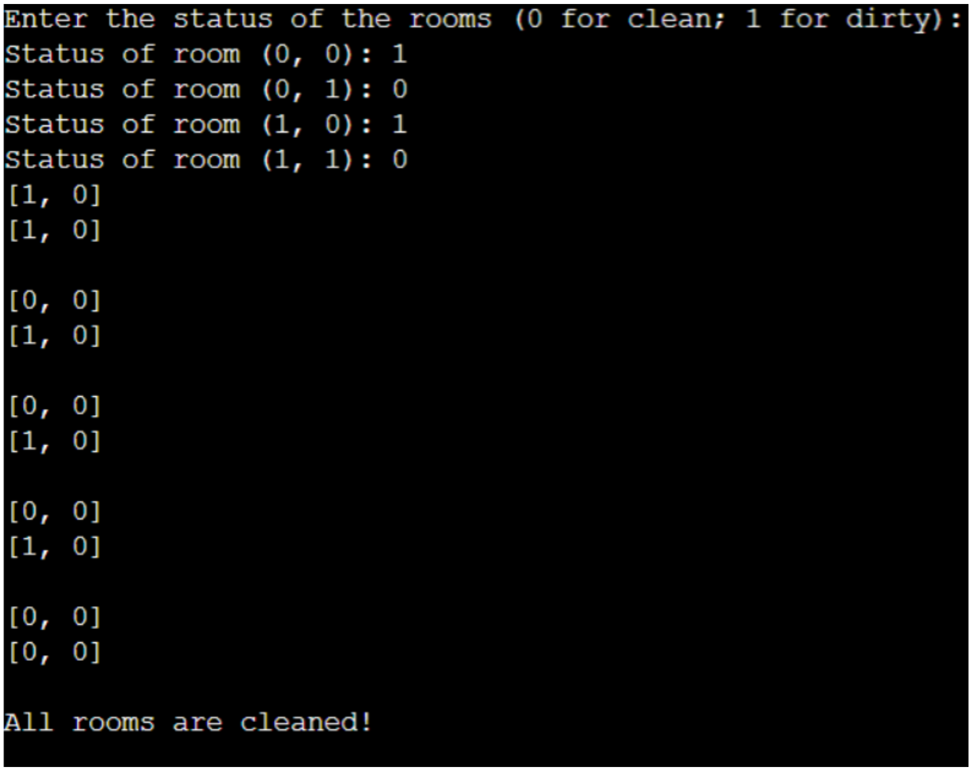
```
Initial environment:
 D  D
 D  D
 Total Cleaned cells : 0
cleaned position(0,0)
Moving to next dirty position (0,1)
Cleaned position (0,1)
Moving to next dirty position (1,0)
Cleaned position (1,1)
No morty dirty rooms

Final environment:
 C  C
 C  C
```

1/10

Code:

```
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:

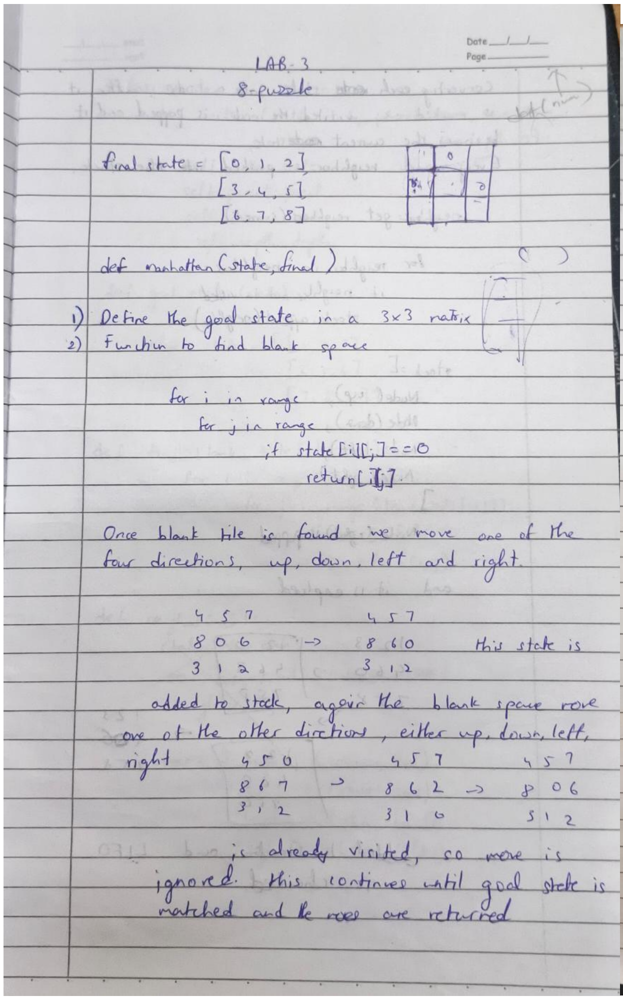


```
Enter the status of the rooms (0 for clean; 1 for dirty):
Status of room (0, 0): 1
Status of room (0, 1): 0
Status of room (1, 0): 1
Status of room (1, 1): 0
[1, 0]
[1, 0]

[0, 0]
[1, 0]

[0, 0]
[1, 0]

[0, 0]
[1, 0]

[0, 0]
[0, 0]

All rooms are cleaned!
```

Program 3

Implement 8 puzzle problems using Depth First Search (DFS)

Algorithm:



LAB-3

8-puzzle

```
final state = [0, 1, 2]
              [3, 4, 5]
              [6, 7, 8]

def manhattan (state, final )
```

1) Define the goal state in a 3x3 matrix
2) Function to find blank space

```
for i in range
    for j in range
        if state[i][j]==0
            return[i][j].
```

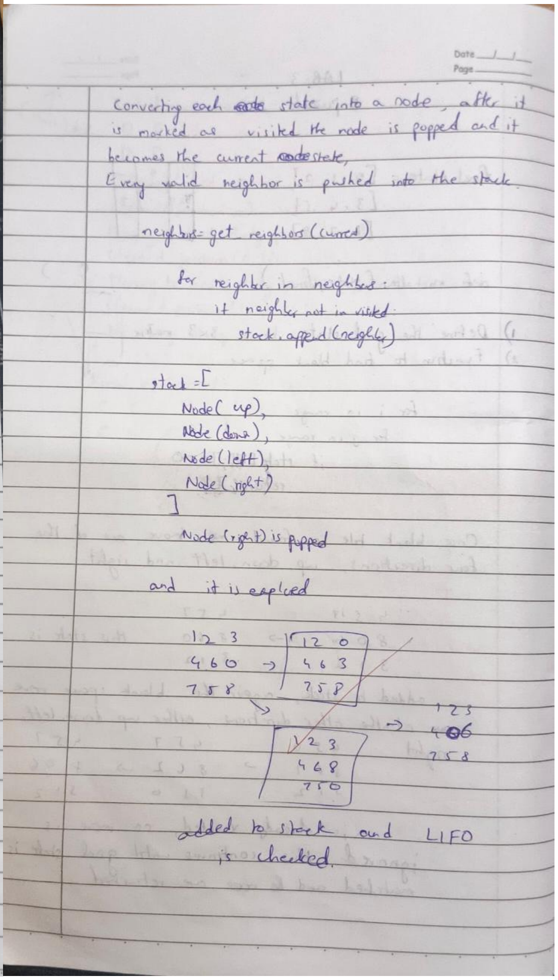
Once blank tile is found we move one of the four directions, up, down, left and right.

```
4 5 7        4 5 7
8 0 6   ->   8 6 0     This state is
3 1 2        3 1 2
```

added to stack, again the blank space move one of the other directions, either up, down, left, right

```
4 5 0        4 5 7        4 5 7
8 6 7   ->   8 6 2   ->   8 0 6
3 1 2        3 1 0        5 1 2
```

is already visited, so move is ignored. This continues until goal state is matched and the moves are returned



Converting each ~~node~~ state into a node, after it is marked as visited the node is popped and it becomes the current ~~node~~ state,
Every valid neighbor is pushed into the stack.

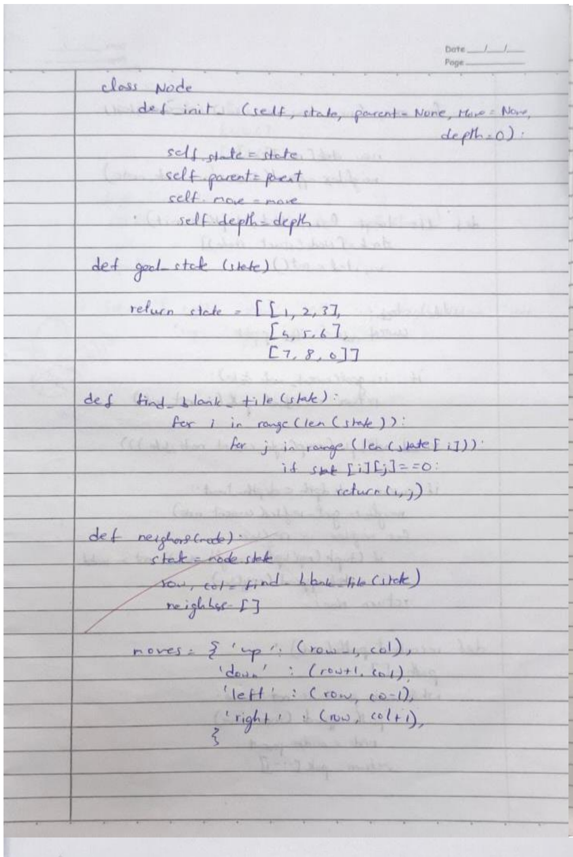
```
neighbors= get_neighbors(current)

for neighbor in neighbors:
    if neighbor not in visited:
        stack.append(neighbor)

stack =[
    Node( up),
    Node (down),
    Node (left),
    Node( right)
]
```

Node (right) is popped

and it is explored

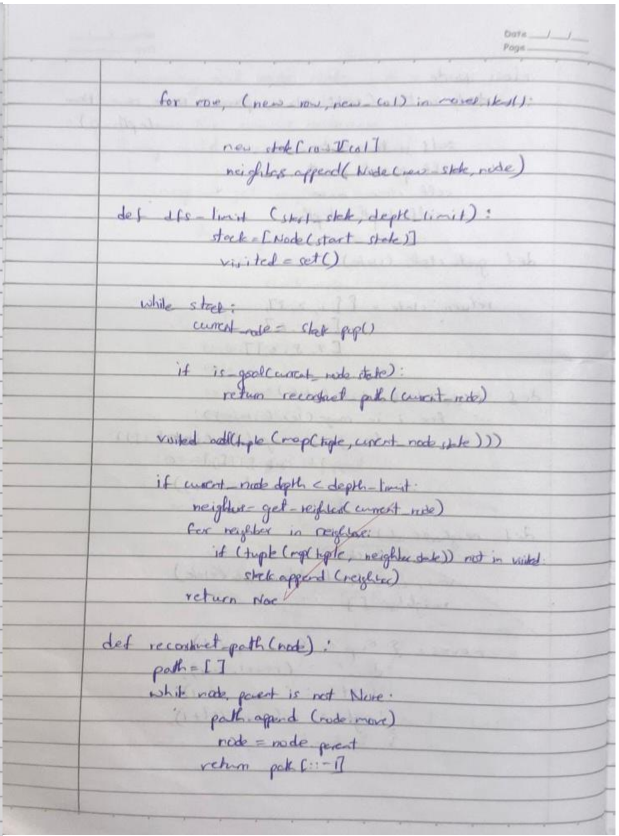
```
1 2 3        1 2 0
4 6 0   ->   4 6 3
7 5 8        7 5 8

             1 2 3          1 2 3
             4 6 8    ->    4 0 6
             7 5 0          7 5 8
```

added to stack and LIFO is checked.



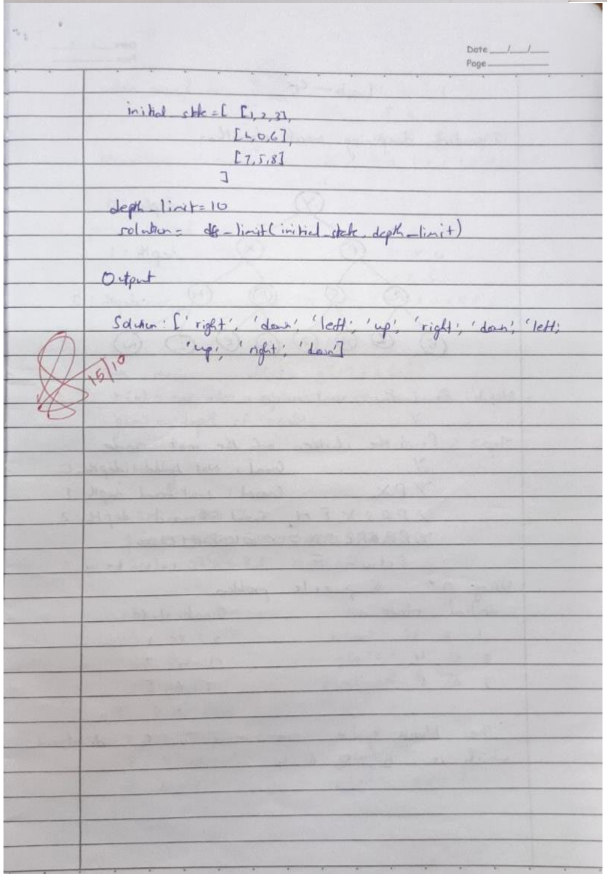
Date ___/___/___
Page ______

```
class Node
    def_init_ (self, state, parent= None, Move= None,
                                            depth=0):
            self.state = state
            self.parent= parent
            self.move = move
            self.depth= depth

def goal_state (state)

    return state = [[1, 2, 3],
                    [4, 5, 6],
                    [7, 8, 0]]

def find_blank_tile(state):
        for i in range(len(state)):
                for j in range (len(state[i])):
                        if stat [i][j]==0:
                                return(i,j)

def neighbors(node):
        state = node.state
        row, col= find_blank_tile(state)
        neighbor= []

    moves= {'up': (row-1, col),
            'down' : (row+1, col),
            'left' : (row, co-1),
            'right' : (row, col+1),
            }
```



Date ___/___/___
Page ______

```
        for row, (new_row, new_col) in moves.items():

                new state[row][col]
                neighbors.append( Node(new_state, node)

def dfs_limit (start_state, depth_limit):
                stack=[Node(start_state)]
                visited = set()

    while stack:
            current_node = stack.pop()

        if  is_goal(current_node.state):
                return reconstruct_path(current_node)

        visited.add(tuple(map(tuple,current_node.state)))

        if current_node.depth < depth_limit:
            neighbors= get_neighbors(current_node)
            for neighbor in neighbors:
                if (tuple(map(tuple, neighbor.state)) not in visited:
                    stack.append (neighbor)
            return None

def reconstruct_path(node):
        path=[]
        while node.parent is not None:
                path.append (node.move)
                node = node.parent
            return path[::-1]
```



Date ___/___/___
Page ______

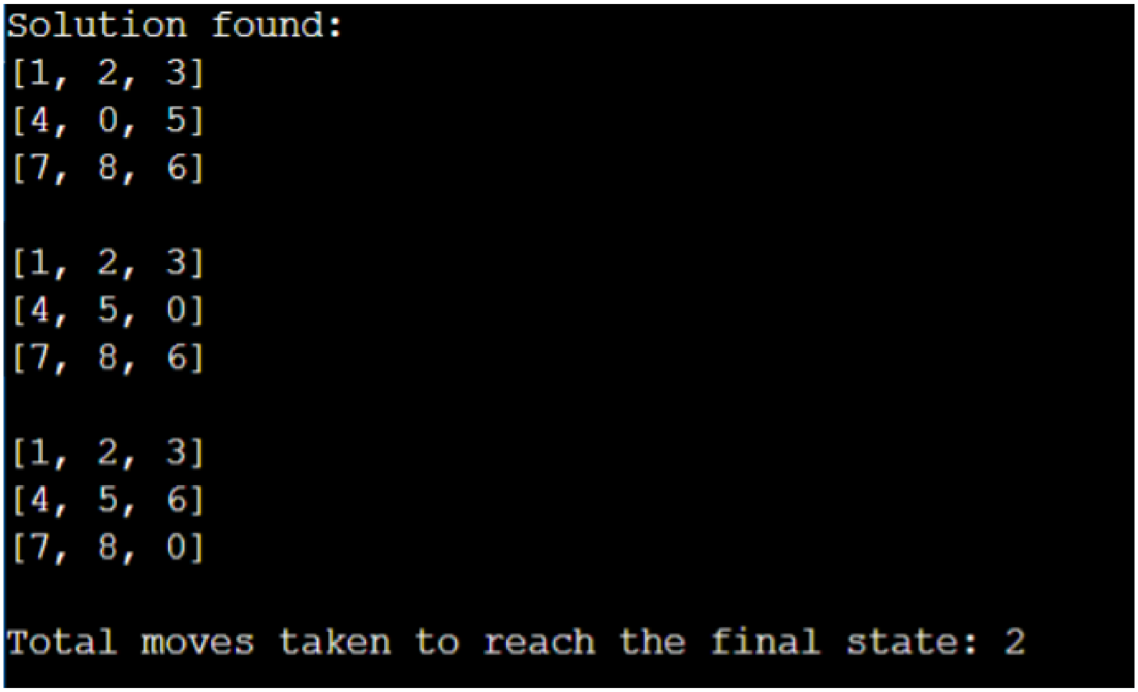
```
    initial_state=[ [1,2,3],
                    [4,0,6],
                    [7,5,8]
                  ]

depth_limit=10
solution= dfs_limit(initial_state, depth_limit)

Output

Solution: ['right', 'down', 'left', 'up', 'right', 'down', 'left',
        'up', 'right', 'down']
```

15/10

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
```

Program 4

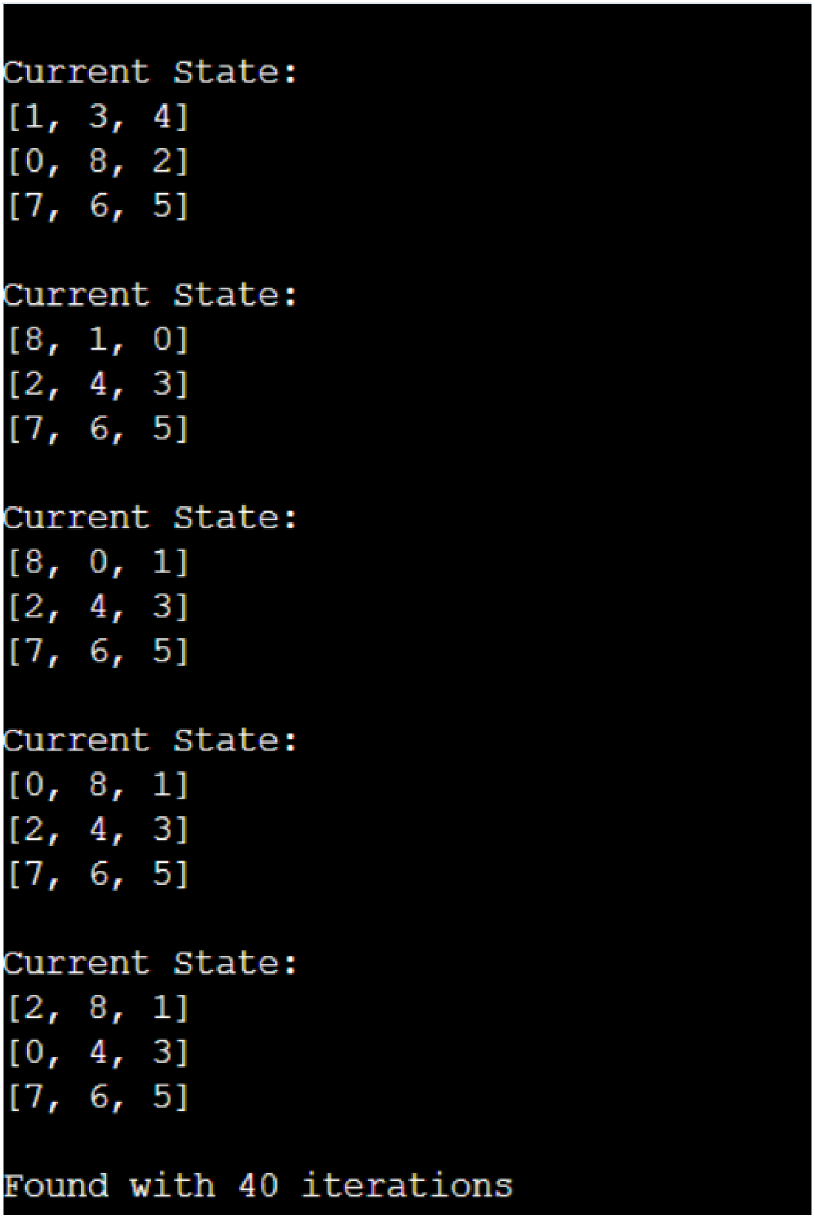
Implement A* search algorithm

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))
```

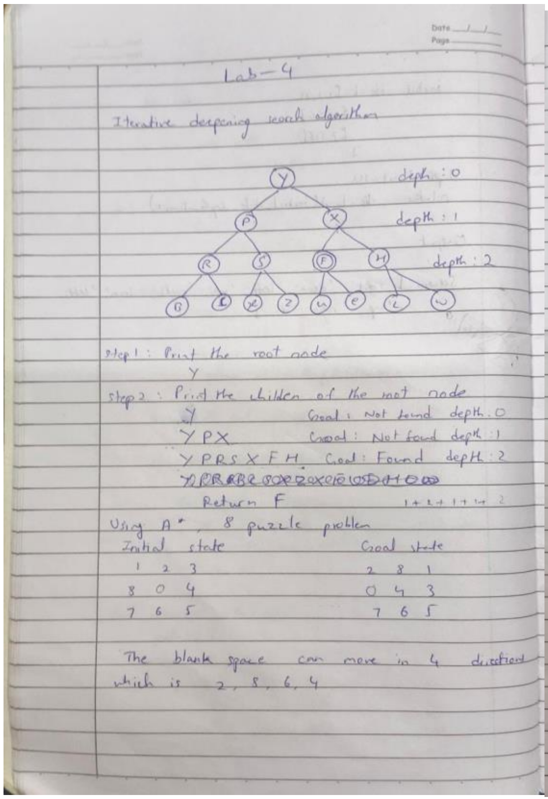
Output:



```
Current State:
[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
```

Implement Iterative deepening search algorithm

Algorithm:



Lab — 4

Iterative deepening search algorithm

Y
P
X
R
S
F
H
B
x
Z
e
depth : 0
depth : 1
depth : 2

step 1 : Print the root node
Y

step 2 : Print the children of the root node
Y Goal : Not found depth . 0
Y P X Goal : Not found depth : 1
Y P R S X F H Goal : Found depth : 2
Return F 1 + 2 + 1 + 1 + 2

Using A*, 8 puzzle problem

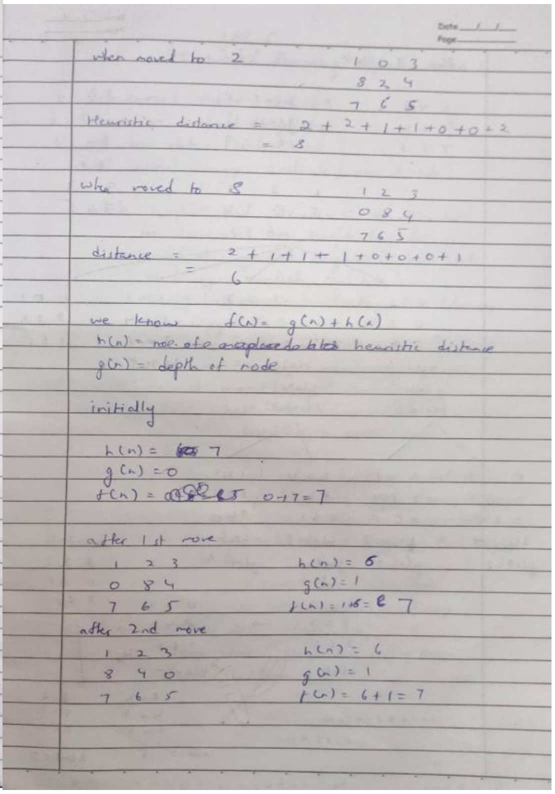
Initial state

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

Goal state

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

The blank space can move in 4 directions which is 2, 8, 6, 4



when moved to 2

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

Heuristic distance = 2 + 2 + 1 + 1 + 0 + 0 + 2
= 8

when moved to 8

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

distance = 2 + 1 + 1 + 1 + 0 + 0 + 0 + 1
= 6

we know $f(n) = g(n) + h(n)$
$h(n)$ = ~~no. of misplaced tiles~~ heuristic distance
$g(n)$ = depth of node

initially

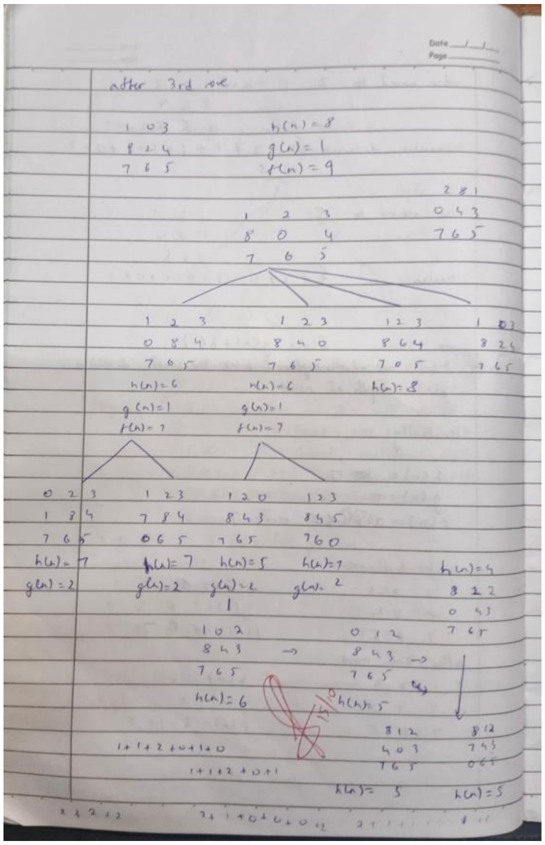
$h(n) = 7$
$g(n) = 0$
$f(n) = 0 + 7 = 7$

after 1st move

```
1 2 3      h(n) = 6
0 8 4      g(n) = 1
7 6 5      f(n) = 1+6 = 7
```

after 2nd move

```
1 2 3      h(n) = 6
8 4 0      g(n) = 1
7 6 5      f(n) = 6 + 1 = 7
```



after 3rd move

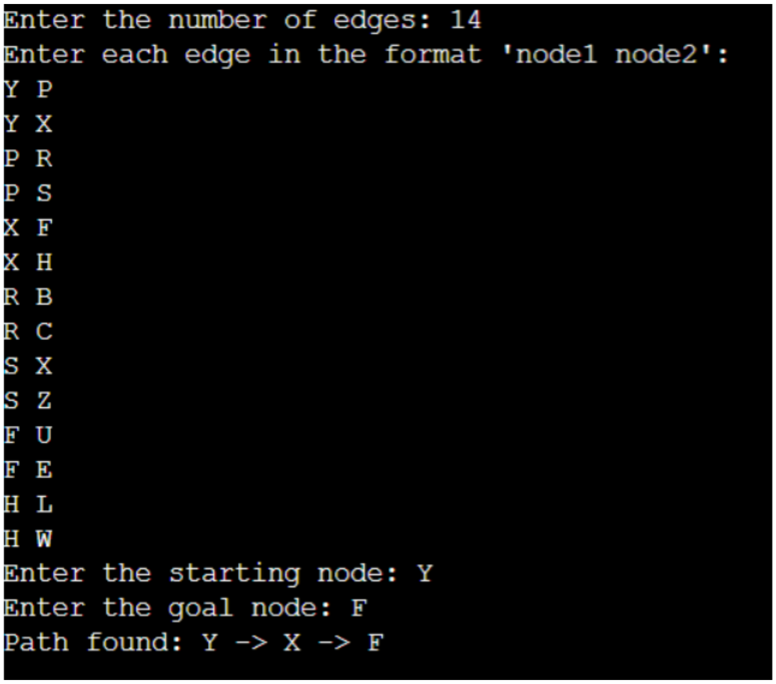
```
1 0 3      h(n) = 8
8 2 4      g(n) = 1
7 6 5      f(n) = 9
```

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

   1 2 3        1 2 3        1 2 3        1 0 3
   0 8 4        8 4 0        8 6 4        8 2 4
   7 0 5        7 6 5        7 0 5        7 6 5
   h(n) = 6     h(n) = 6     h(n) = 8
   g(n) = 1     g(n) = 1
   f(n) = 7     f(n) = 7

0 2 3     1 2 3     1 2 0     1 2 3
1 8 4     7 8 4     8 4 3     8 4 5
7 6 5     0 6 5     7 6 5     7 6 0
h(n) = 7  h(n) = 7  h(n) = 5  h(n) = 7             h(n) = 4
g(n) = 2  g(n) = 2  g(n) = 2  g(n) = 2              8 1 2
                                                    0 4 3
                    1 0 2            0 1 2          7 6 5
                    8 4 3     ->     8 4 3   ->
                    7 6 5            7 6 5
                    h(n) = 6         h(n) = 5

                                     8 1 2          8 1 2
          1 + 1 + 2 + 0 + 1 + 0      4 0 3          7 4 3
                  1 + 1 + 2 + 0 + 1  7 6 5          0 6 5
                                     h(n) = 5       h(n) = 5
   2 + 2 + 2      2 + 1 + 0 + 0 + 0 + 2
```

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()
```

Output:

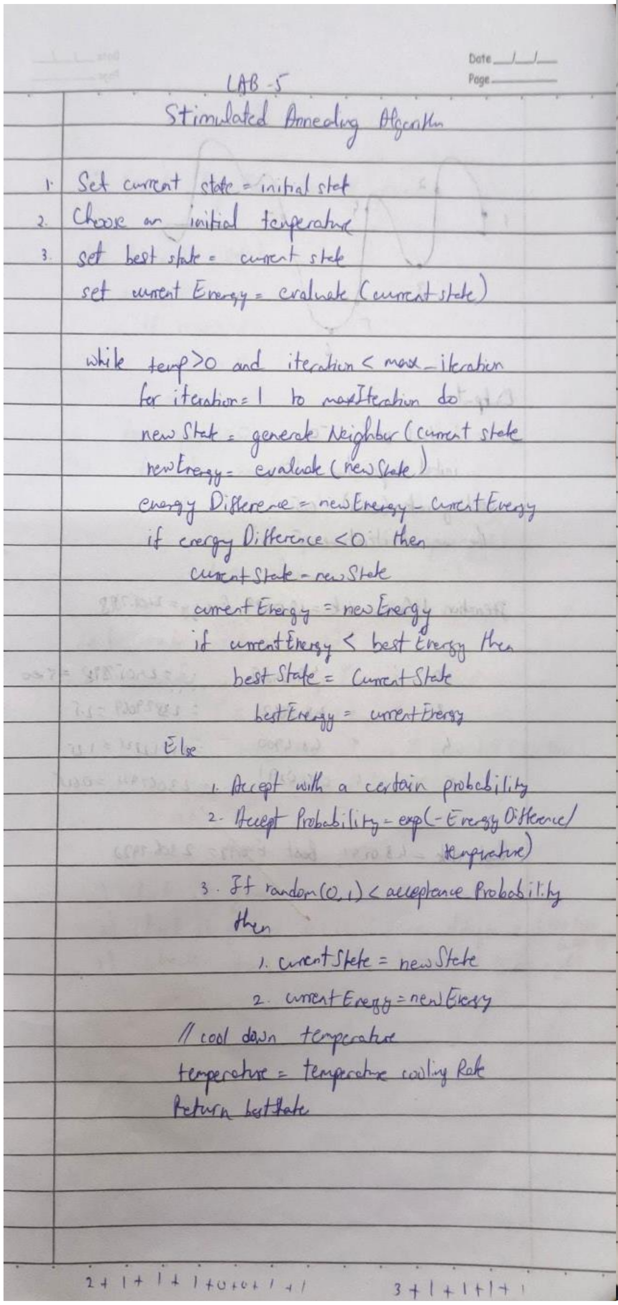


```
Enter the number of edges: 14
Enter each edge in the format 'node1 node2':
Y P
Y X
P R
P S
X F
X H
R B
R C
S X
S Z
F U
F E
H L
H W
Enter the starting node: Y
Enter the goal node: F
Path found: Y -> X -> F
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:

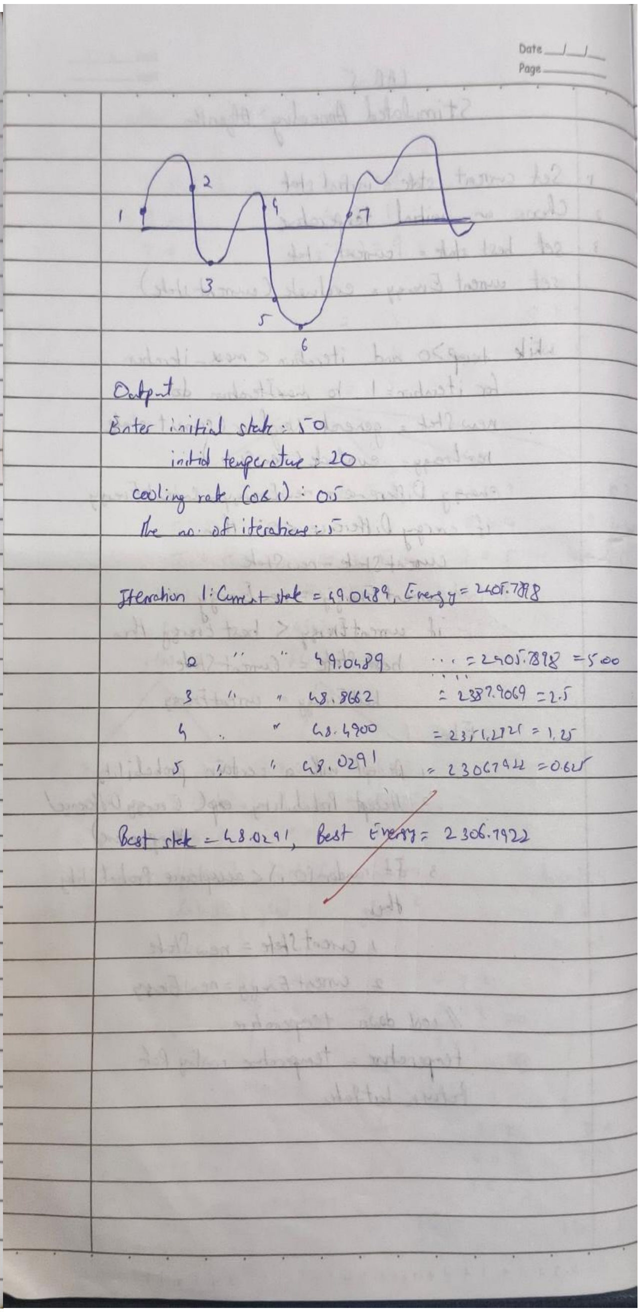


LAB -5

Stimulated Annealing Algorithm

```
1. Set current state = initial stat
2. Choose an initial temperature
3. set best state = current stat
   set current Energy = evaluate (current state)

   while temp>0 and iteration < max_iteration
       for iteration=1 to maxIteration do
       new State = generate Neighbor (current state
       newEnergy = evaluate ( new State )
       energy Difference = newEnergy - current Energy
       if energy Difference <0 then
           current State - new State
           current Energy = new Energy
           if current Energy < best Energy then
               best State = Current State
                 bestEnergy = current Energy
     Else
           1. Accept with a certain probability
           2. Accept Probability = exp(-Energy Difference/
                                        temperature)
         3. If random(0,1) < acceptance Probability
            then
               1. current State = new State
               2. current Energy = new Energy
       // cool down temperature
       temperature = temperature cooling Rate
       Return bestState
```

2+1+1+1+0+0+1+1 3+1+1+1+1

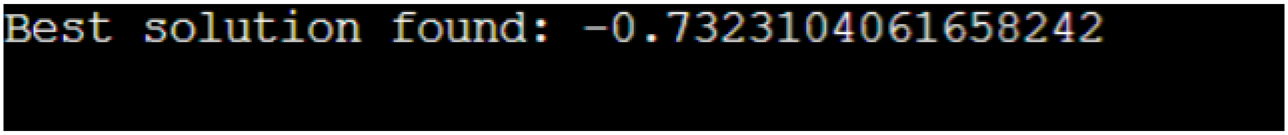


```
Output
Enter initial state : 50
      initial temperature : 20
  cooling rate (0&1) : 0.5
   The no. of iterations : 5

Iteration 1: Current state = 49.0489, Energy = 2405.7898

      2    "      "    49.0489      ... = 2405.7898 = 5.00
      3    "      "    48.8662        = 2387.9069 = 2.5
      4    .      "    48.4900        = 2351.2725 = 1.25
      5    "      "    48.0291        = 2306.7922 = 0.625

Best state = 48.0291, Best Energy = 2306.7922
```

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:

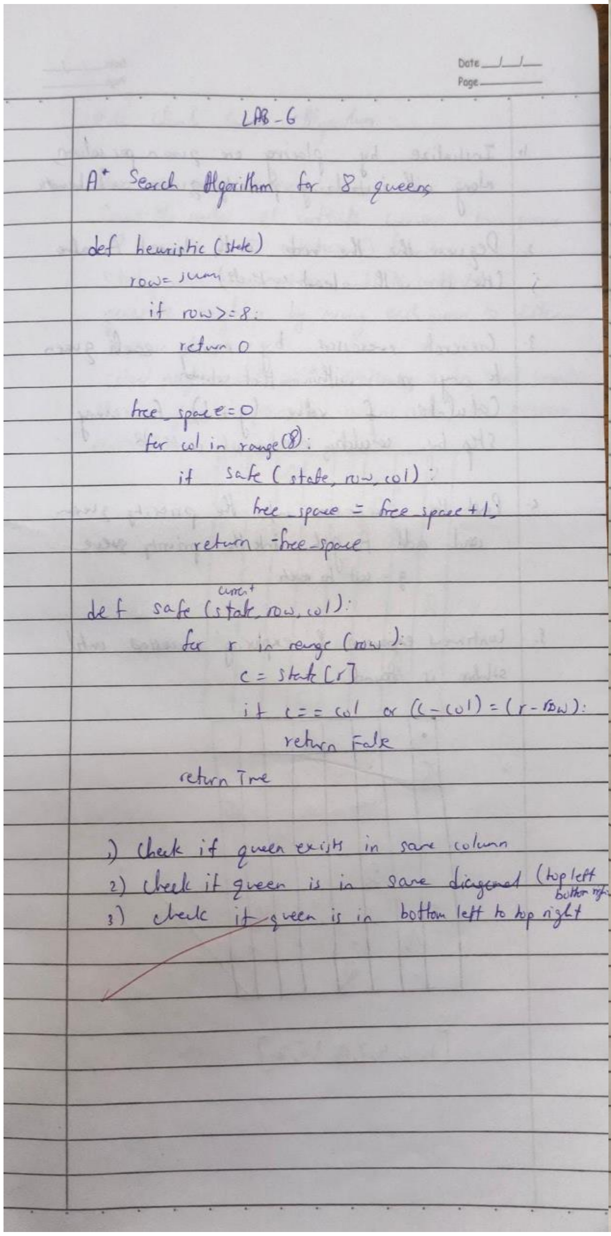


```
Best solution found: -0.7323104061658242
```

Program 6

Implement A* search algorithm for N queens

Algorithm:



Date__/__/__

Page______

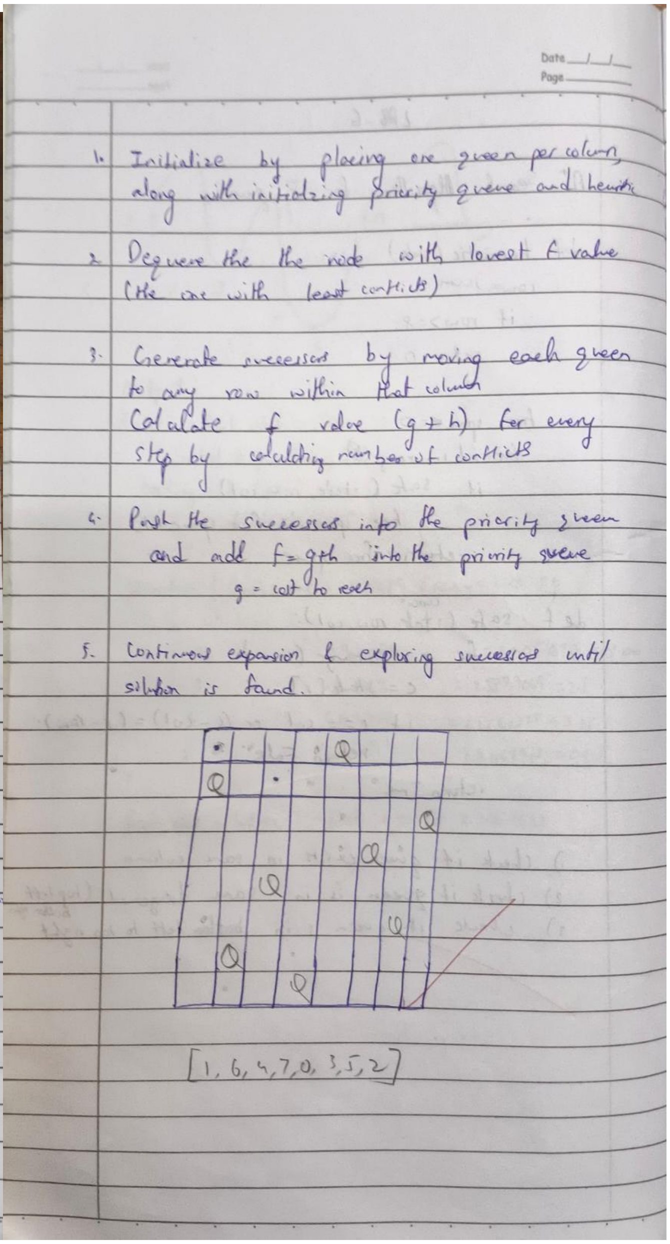
LAB - 6

A* Search Algorithm for 8 queens

```
def heuristic (state)
    row= sum
      if row>=8:
        return 0

    free_space=0
      for col in range(8):
          if  safe ( state, row, col) :
                free_space = free space +1;
          return -free-space

def  safe (state, row, col):
           for  r  in range (row ):
                  c = state [r]
                  if c== col  or ((-col) = (r-row):
                        return False
        return True
```

(current written above "state")

1) Check if queen exists in same column
2) check if queen is in same diagonal (top left bottom right)
3) check if queen is in bottom left to top right



Date__/__/__

Page______

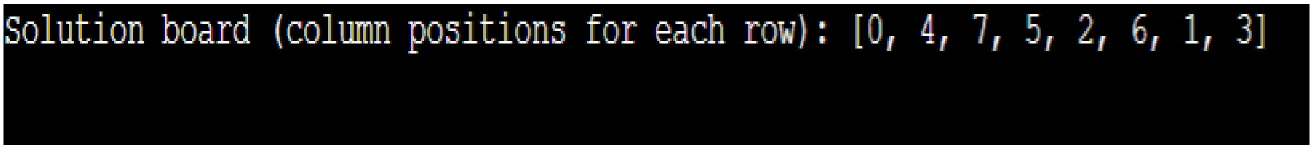
1. Initialize by placing one queen per column, along with initialzing priority queue and heuristic
2. Dequeue the the node with lowest f value (the one with least conflicts)
3. Generate successors by moving each queen to any row within that column
   Calculate f value (g + h) for every step by calculating number of conflicts
4. Push the successors into the priority queue and add f=g+h into the priority queue
   g = cost to reach
5. Continuous expansion & exploring successors until solution is found.

[1, 6, 4, 7, 0, 3, 5, 2]

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)
```

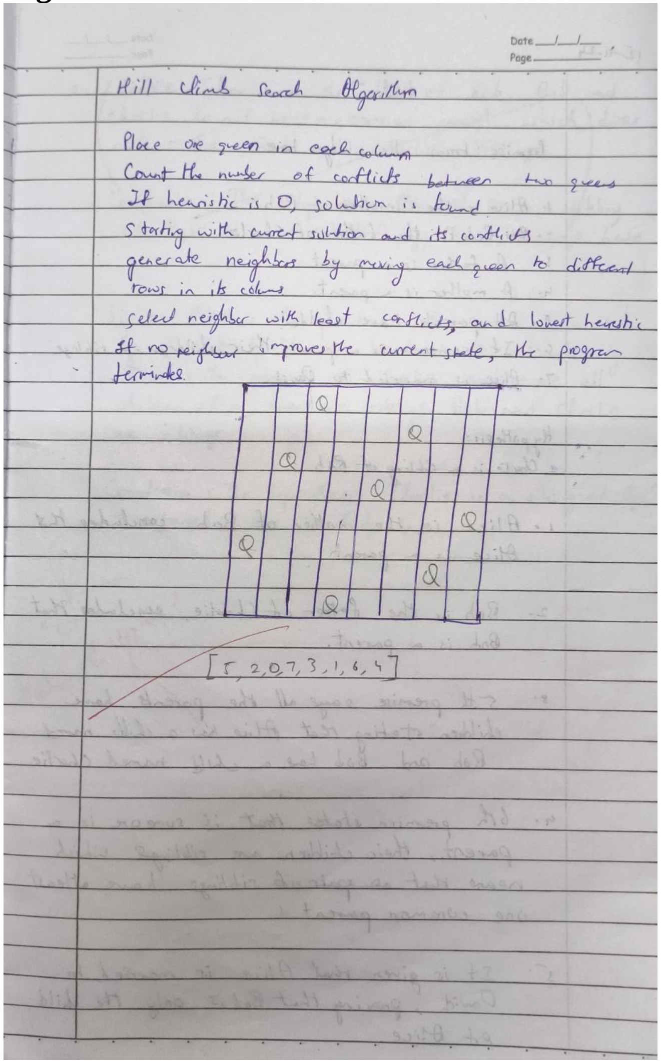
Output:



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

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



Date __/__/__
Page ____

Hill climb Search Algorithm

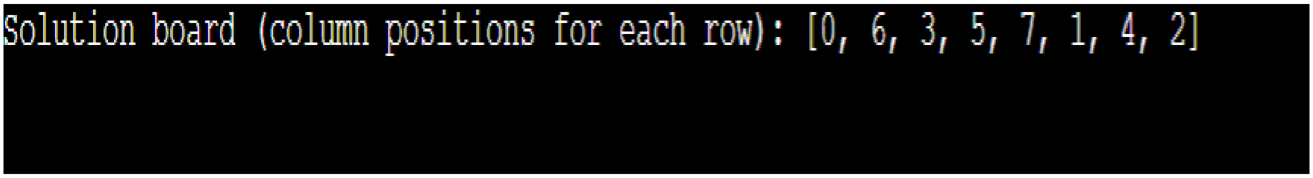
Place one queen in each column
Count the number of conflicts between two queens
If heuristic is 0, solution is found
Starting with current solution and its conflicts
generate neighbors by moving each queen to different rows in its column
select neighbor with least conflicts, and lowest heuristic
If no neighbour improves the current state, the program terminates.

[5, 2, 0, 7, 3, 1, 6, 4]

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:

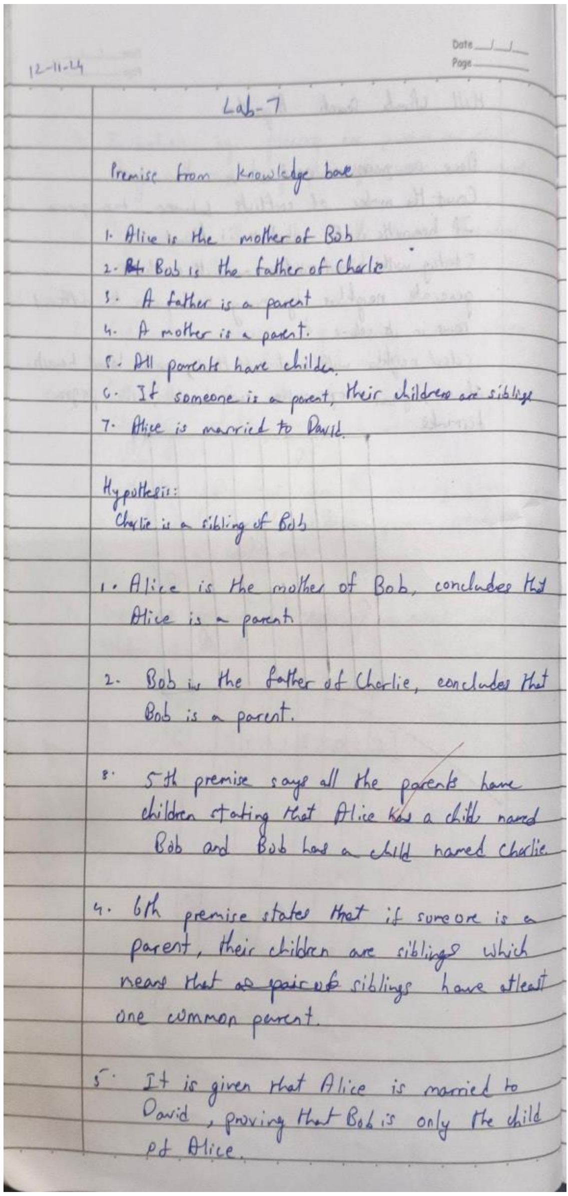


```
Solution board (column positions for each row): [0, 6, 3, 5, 7, 1, 4, 2]
```

Program 7

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

Algorithm:



12-11-24

Date __/__/__
Page ____

Lab-7

Premise from knowledge base

1. Alice is the mother of Bob
2. ~~Bt~~ Bob is the father of Charlie
3. A father is a parent
4. A mother is a parent.
5. All 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

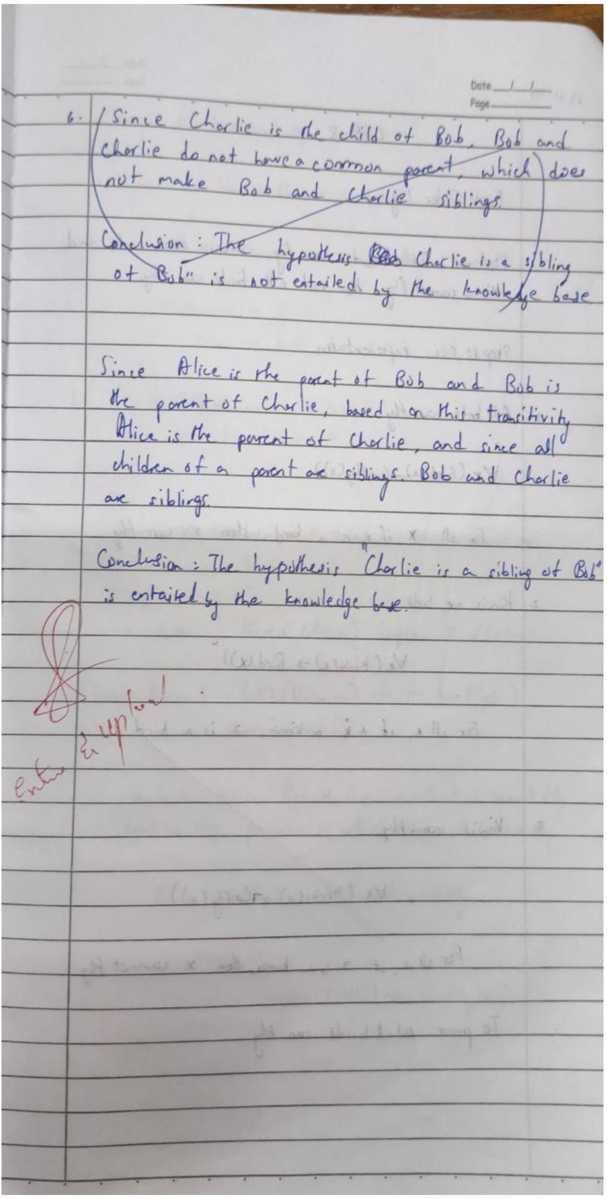
1. Alice is the mother of Bob, concludes that Alice is a parent

2. Bob is the father of Charlie, concludes that Bob is a parent.

3. 5th premise says all the parents have children stating that Alice has a child named Bob and Bob has a child named Charlie.

4. 6th premise states that if someone is a parent, their children are siblings which means that ~~a pair of~~ siblings have atleast one common parent.

5. It is given that Alice is married to David, proving that Bob is only the child of Alice.



Date __/__/__
Page ____

6. ~~Since Charlie is the child of Bob, Bob and Charlie do not have a common parent, which does not make Bob and Charlie siblings.~~

~~Conclusion : The hypothesis "~~Bob~~ Charlie is a sibling of Bob" is not entailed by the knowledge base~~

Since Alice is the parent of Bob and Bob is the parent of Charlie, based on this transitivity Alice is the parent of Charlie, and since all children of a parent are siblings, Bob and Charlie are siblings.

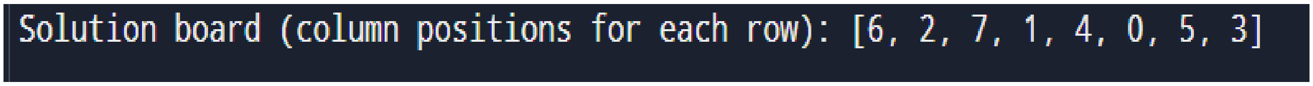
Conclusion : The hypothesis "Charlie is a sibling of Bob" is entailed by the knowledge base.

Entire & upload

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:

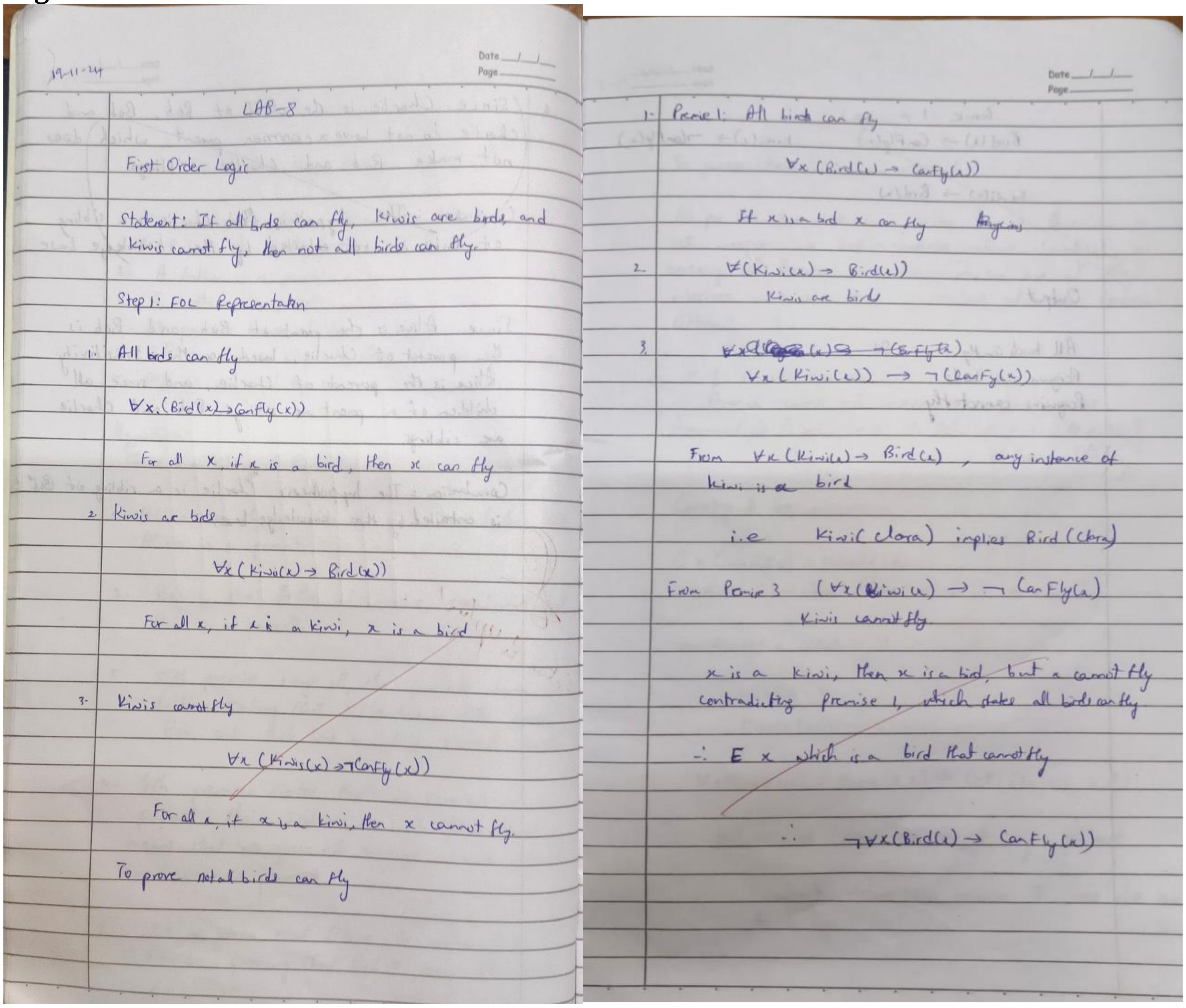


```
Solution board (column positions for each row): [6, 2, 7, 1, 4, 0, 5, 3]
```

Program 8

Implement unification in first order logic

Algorithm:



19-11-24

Date __/__/__
Page ____

LAB-8

First Order Logic

Statement: If all birds can fly, kiwis are birds and kiwis cannot fly, then not all birds can fly.

Step 1: FOL Representation

1. All birds can fly

$\forall x. (Bird(x) \rightarrow CanFly(x))$

For all x, if x is a bird, then x can fly

2. Kiwis are birds

$\forall x (Kiwi(x) \rightarrow Bird(x))$

For all x, if x is a kiwi, x is a bird

3. Kiwis cannot fly

$\forall x (Kiwis(x) \rightarrow \neg CanFly(x))$

For all x, if x is a kiwi, then x cannot fly.

To prove not all birds can fly

Date __/__/__
Page ____

1. Premise 1: All birds can fly

$\forall x (Bird(x) \rightarrow CanFly(x))$

If x is a bird x can fly [illegible]

2. $\forall (Kiwi(x) \rightarrow Bird(x))$

Kiwis are bird

3. ~~$\forall x ($[illegible]$(x) \rightarrow \neg (CanFly(x))$~~

$\forall x (Kiwi(x)) \rightarrow \neg (CanFy(x)))$

From $\forall x (Kiwi(x) \rightarrow Bird(x))$, any instance of kiwi is a bird

i.e Kiwi(clora) implies Bird (Clora)

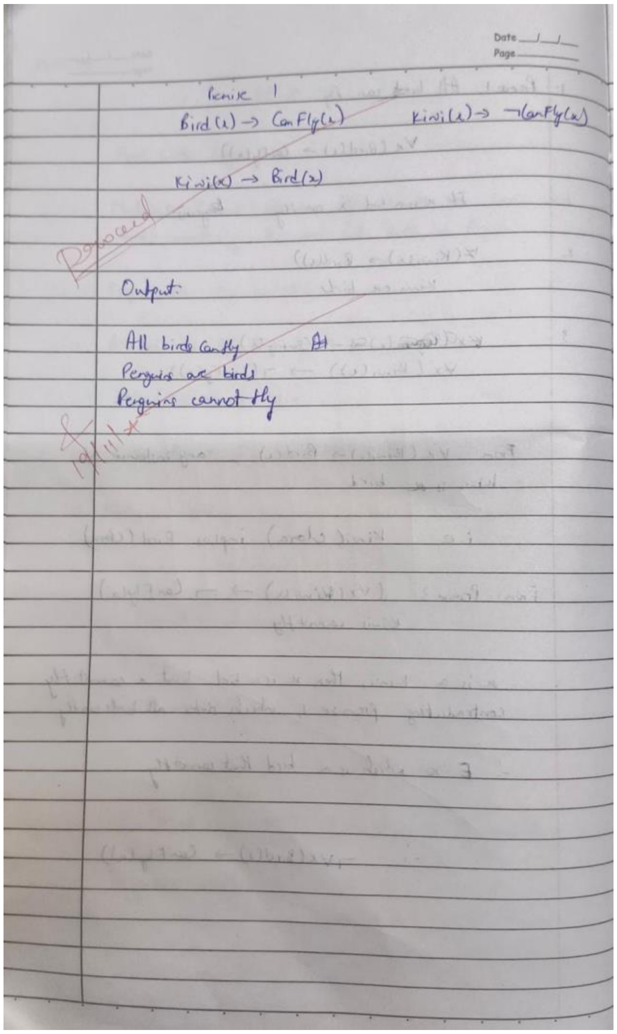
From Premise 3 $(\forall x (Kiwi(x)) \rightarrow \neg CanFly(x))$

Kiwis cannot fly.

x is a kiwi, then x is a bird, but x cannot fly contradicting premise 1, which states all birds can fly

∴ E x which is a bird that cannot fly

∴ $\neg \forall x (Bird(x) \rightarrow CanFly(x))$



Premise 1

$Bird(x) \rightarrow CanFly(x)$ $Kiwi(x) \rightarrow \neg CanFly(x)$

$Kiwi(x) \rightarrow Bird(x)$

Proceed

Output:

All birds can fly

Penguins are birds

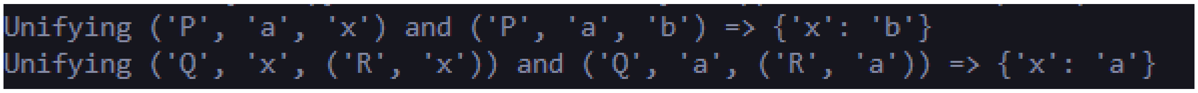
Penguins cannot fly

19/11/24

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:

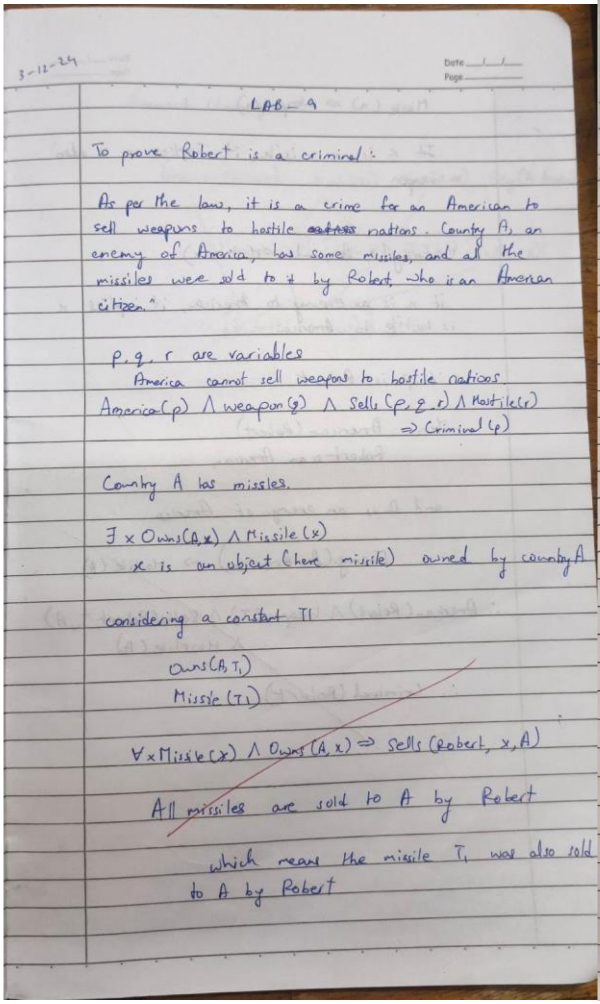


```
Unifying ('P', 'a', 'x') and ('P', 'a', 'b') => {'x': 'b'}
Unifying ('Q', 'x', ('R', 'x')) and ('Q', 'a', ('R', 'a')) => {'x': 'a'}
```

Program 9

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

Algorithm:



3-12-24

Date __/__/__
Page ____

LAB - 9

To prove Robert is a criminal :

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."

p, q, r are variables

America cannot sell weapons to hostile nations.

$America(p) \land weapon(q) \land Sells(p, q, r) \land Hostile(r) \Rightarrow Criminal(p)$

Country A has missles.

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

x is an object (here missile) owned by country A

considering a constant T1

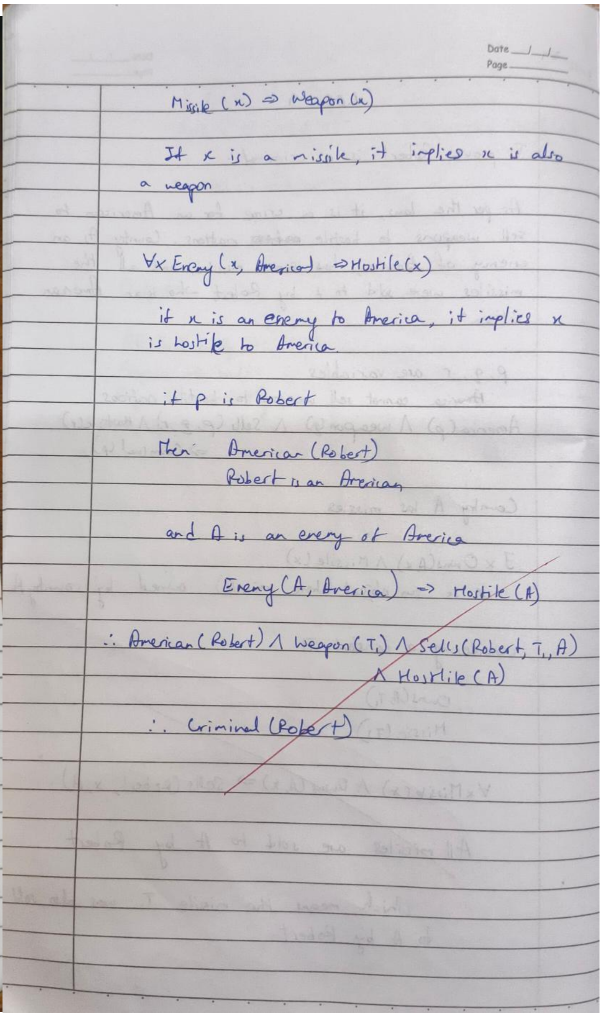
$Owns(A, T_1)$

$Missile(T_1)$

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

All missiles are sold to A by Robert

which means the missile T_1 was also sold to A by Robert



Date __/__/__
Page ____

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

If x is a missile, it implies x is also a weapon

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

if x is an enemy to America, it implies x is hostile to America.

if p is Robert

Then: $American(Robert)$

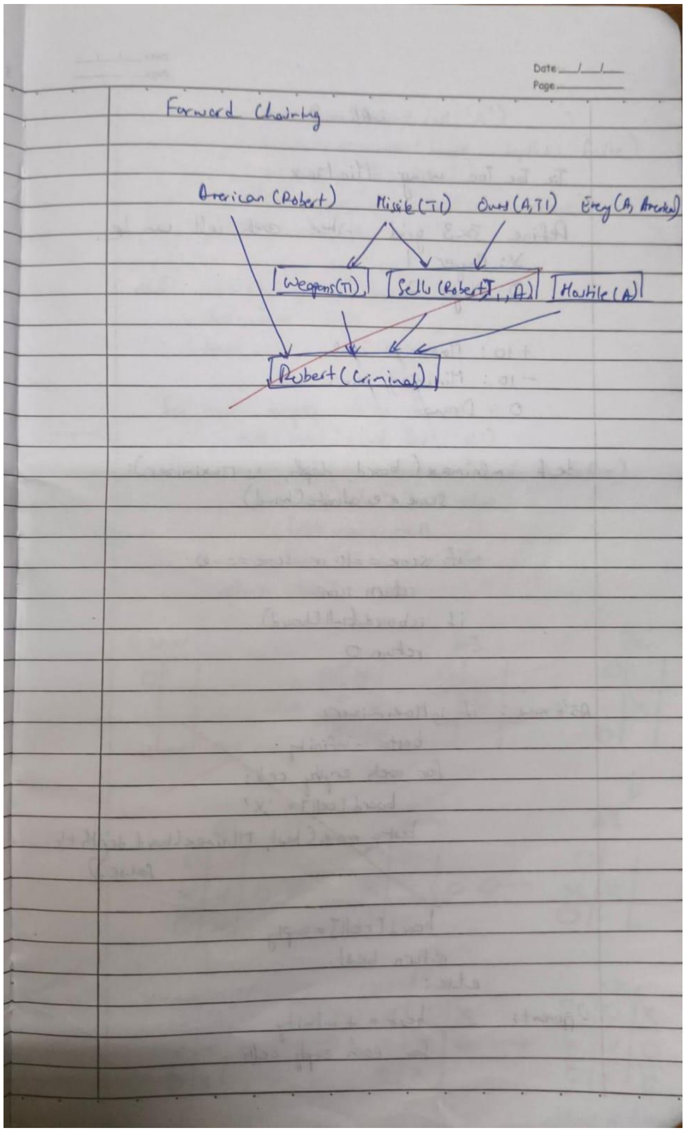
Robert is an American

and A is an enemy of America

$Enemy(A, America) \Rightarrow Hostile(A)$

$\therefore American(Robert) \land Weapon(T_1) \land Sells(Robert, T_1, A) \land Hostile(A)$

$\therefore Criminal(Robert)$

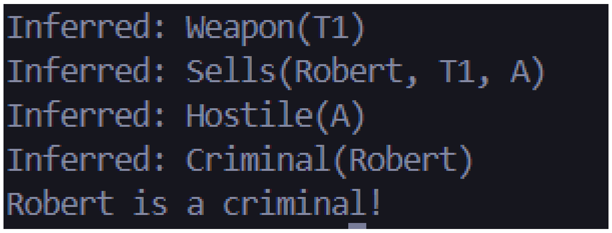


Code:

```
# 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:

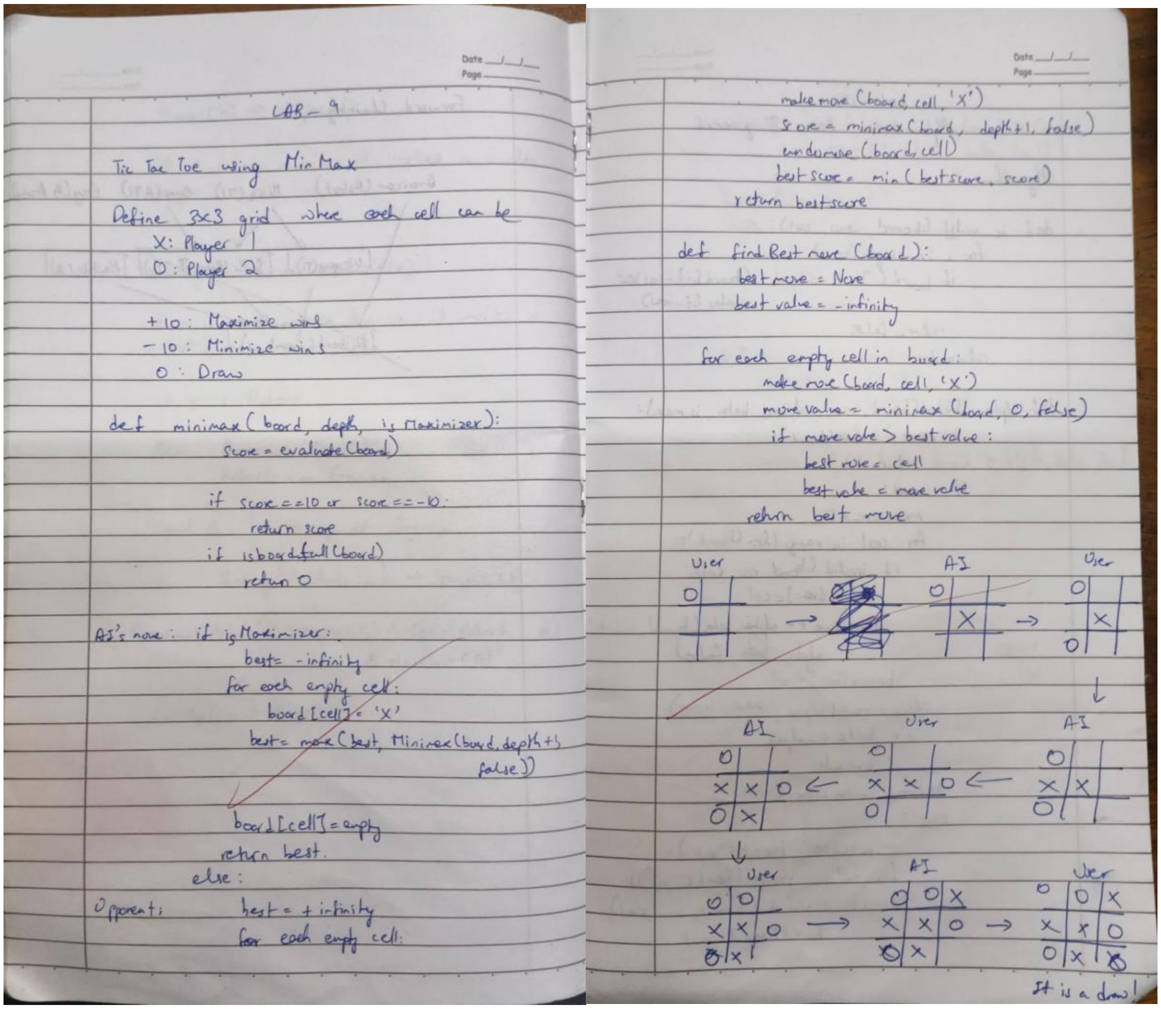


```
Inferred: Weapon(T1)
Inferred: Sells(Robert, T1, A)
Inferred: Hostile(A)
Inferred: Criminal(Robert)
Robert is a criminal!
```

Program 10

Implement Min-Max Algorithm for Tic Tac Toe

Algorithm:



Date __/__/__
Page ____

LAB – 9

Tic Tac Toe using Min Max

```
Define 3x3 grid where each cell can be
    X: Player 1
    O: Player 2

    +10 : Maximize wins
    -10 : Minimize wins
      0 : Draw

def minimax(board, depth, is Maximizer):
        score = evaluate(board)

        if score ==10 or score ==-10:
            return score
        if isboardfull(board)
            return 0

AI's move :  if isMaximizer:
                best= -infinity
                for each empty cell:
                    board[cell]= 'X'
                    best= max(best, Minimax(board, depth+1,
                                                 false))

                board[cell]=empty
              return best.
           else:
Opponent:       best = + infinity
                for each empty cell:
                    make move(board, cell, 'X')
                    score = minimax(board, depth+1, false)
                    undomove(board, cell)
                    best score = min(bestscore, score)
              return bestscore

def find Best move (board):
      best move = None
      best value = -infinity

    for each empty cell in board:
         make move(board, cell, 'X')
         move value = minimax(board, 0, false)
         if move value > best value :
             best move = cell
             best value = move value
      return best move
```

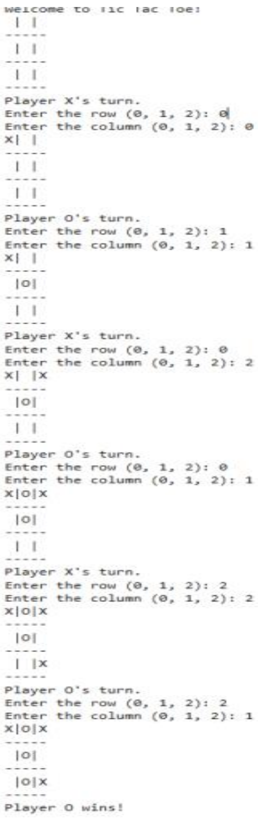
```
User        AI          User
O| |        O| |        O| |
 | |   ->    |X|   ->    |X|
 | |         | |        O| |

AI          User        AI
O| |        O| |        O| |
X|X|O  <-   X|X|O  <-   X|X|
O|X|        O| |        O| |

User        AI          User
O|O|        O|O|X       O|O|X
X|X|O  ->   X|X|O  ->   X|X|O
O|X|        O|X|        O|X|O
```

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}")
```

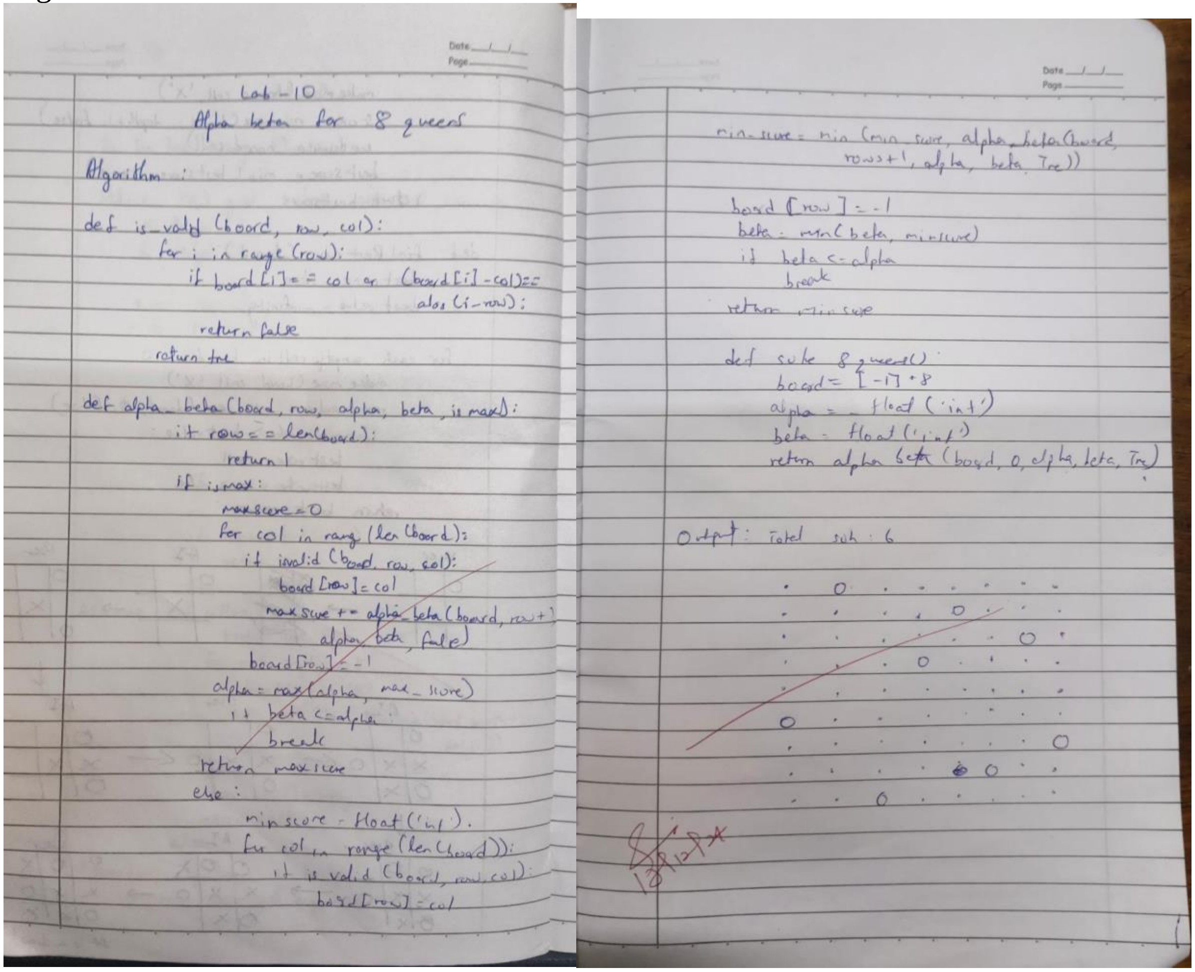
Output:



```
Welcome to Tic Tac Toe!
 | |
-----
 | |
-----
 | |
-----
Player X's turn.
Enter the row (0, 1, 2): 0
Enter the column (0, 1, 2): 0
X| |
-----
 | |
-----
 | |
-----
Player O's turn.
Enter the row (0, 1, 2): 1
Enter the column (0, 1, 2): 1
X| |
-----
 |O|
-----
 | |
-----
Player X's turn.
Enter the row (0, 1, 2): 0
Enter the column (0, 1, 2): 2
X| |X
-----
 |O|
-----
 | |
-----
Player O's turn.
Enter the row (0, 1, 2): 0
Enter the column (0, 1, 2): 1
X|O|X
-----
 |O|
-----
 | |
-----
Player X's turn.
Enter the row (0, 1, 2): 2
Enter the column (0, 1, 2): 2
X|O|X
-----
 |O|
-----
 | |X
-----
Player O's turn.
Enter the row (0, 1, 2): 2
Enter the column (0, 1, 2): 1
X|O|X
-----
 |O|
-----
 |O|X
-----
Player O wins!
```

Implement Alpha-Beta Pruning for 8 queens

Algorithm:

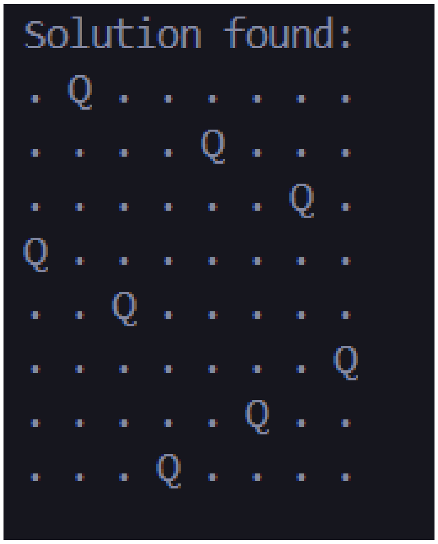


```
Lab - 10
Alpha beta for 8 queens

Algorithm :

def is_valid (board, row, col):
        for i in range (row):
            if board [i] = = col or (board [i] - col)==
                                        abs (i - row) :
                return false
        return true

def alpha_beta (board, row, alpha, beta, is max):
        if row = = len(board):
            return 1
        if is max :
            max score = 0
            for col in rang (len (board):
                if invalid (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,
                    rows+1, alpha, beta, True))

 board [row] = -1
 beta = min (beta, minscore)
 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)

Output : Total soln : 6

    .  O  .  .  .  .  .  .
    .  .  .  .  O  .  .  .
    .  .  .  .  .  .  O  .
    .  .  .  O  .  .  .  .
    .  .  .  .  .  .  .  .
    O  .  .  .  .  .  .  .
    .  .  .  .  .  .  .  O
    .  .  .  .  ⊙  O  .  .
    .  .  O  .  .  .  .  .
```

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 . . . .
```