Artificial Intelligence Lab Report

Submitted by

Rushi Hundiwala(1BM21CS224)

Course: Artificial Intelligence
Course Code: 24CS5PCAIN
Sem & Section: 5D

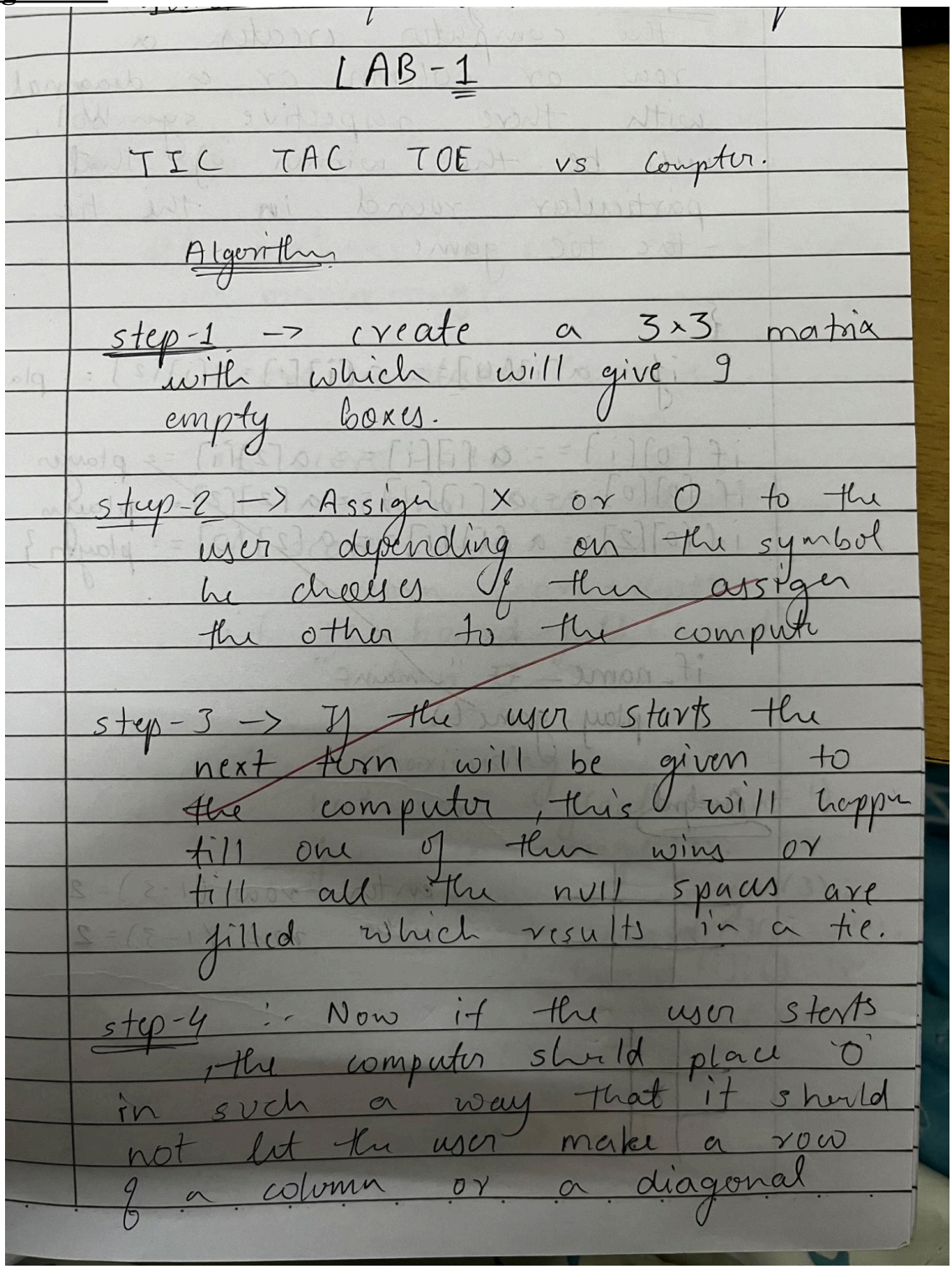
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B. M. S. COLLEGE OF ENGINEERING
(Autonomous Institution under
VTU) BENGALURU-560019
2023-2024

Table of contents

Program 1 - Tic Tac toe

Algorithm



LAB - 1

TIC TAC TOE vs Computer.

Algorithm

step-1 -> create a 3×3 matrix with which will give 9 empty boxes.

step-2 -> Assign X or O to the user depending on the symbol he chooses & then assign the other to the computer

step-3 -> If the user starts the next turn will be given to the computer, this will happen till one of them wins or till all the null spaces are filled which results in a tie.

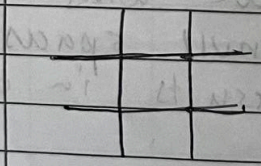
step-4 :- Now if the user starts, the computer should place 'O' in such a way that it should not let the user make a row & a column or a diagonal.

with X's. (basically should not let the user win).

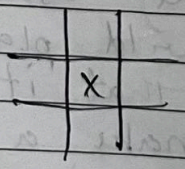
step-5 :- If the user or the computer creates a row or column or a diagonal. with there respective symbol, will be the winner of that particular round in the tic-tac-toe game.

```
{
  if a[i][0] == a[i][1] = a[i][2] == player

if [0][i] == a [i][i] == a[2][i] == player
if [0][0] == a [1][1] == a[2][2] == player
if [0][2] == a [1][1] == a[2][0] == player }

if_name_ == "_main_"
   play game()
```

Output



enter row (1-3) = 2
enter row (1-3) = 2



code:

```
import random
import math


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


def check_winner(board,
    mark):
    # Check rows
    for row in board:
        if all(cell == mark for
    cell in row):
            return True

    # Check columns
    for col in range(3):
        if all(board[row][col] ==
    mark for row in range(3)):
            return True

    # Check diagonals
    if all(board[i][i] == mark
    for i in range(3)) or
    all(board[i][2 - i] == mark
    for i in range(3)):
        return True

    return False


def
    get_available_moves(boa
    rd):
    return [(r, c) for r in
    range(3) for c in range(3)
    if board[r][c] == " "]


def minimax(board, depth,
```

```
    is_maximizing):
    if check_winner(board,
    "O"):
        return 10 - depth
    if check_winner(board,
    "X"):
        return depth - 10
    if not
    get_available_moves(boa
    rd):
        return 0

    if is_maximizing:
        best_score = -math.inf
        for (row, col) in
    get_available_moves(boa
    rd):
            board[row][col] =
    "O"
            score =
    minimax(board, depth +
    1, False)
            board[row][col] = " "
            best_score =
    max(best_score, score)
        return best_score
    else:
        best_score = math.inf
        for (row, col) in
    get_available_moves(boa
    rd):
            board[row][col] =
    "X"
            score =
    minimax(board, depth +
    1, True)
            board[row][col] = " "
            best_score =
    min(best_score, score)
        return best_score


def computer_move(board):
    best_score = -math.inf
```

```
    best_move = None
    for (row, col) in
    get_available_moves(boa
    rd):
        board[row][col] = "O"
        score =
    minimax(board, 0, False)
        board[row][col] = " "
        if score > best_score:
            best_score = score
            best_move = (row,
    col)
    return best_move


def main():
    print("Yashraj Sinha
    (1BM22CS335)")
    print("Welcome to Tic
    Tac Toe!")
    board = [[" " for _ in
    range(3)] for _ in range(3)]

    print_board(board)

    for turn in range(9):
        if turn % 2 == 0:
            # Player's turn
            while True:
                try:
                    row =
    int(input("Enter the row
    (0, 1, 2): "))
                    col =
    int(input("Enter the
    column (0, 1, 2): "))
                    if (row, col) not
    in
    get_available_moves(boa
    rd):
                        print("This
    spot is already taken or
    invalid. Try again.")
                    else:
```

```
    board[row][col] = "X"
                    break
            except
    ValueError:
                print("Invalid
    input. Please enter
    numbers 0, 1, or 2.")
        else:
            # Computer's turn
            row, col =
    computer_move(board)
            board[row][col] =
    "O"
            print(f"Computer
    chose: ({row}, {col})")

        print_board(board)

        # Check for a winner
        if check_winner(board,
    "X"):

    print("Congratulations!
    You win!")
            return
        elif
    check_winner(board,
    "O"):
            print("Computer
    wins! Better luck next
    time.")
            return

    print("It's a tie!")


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

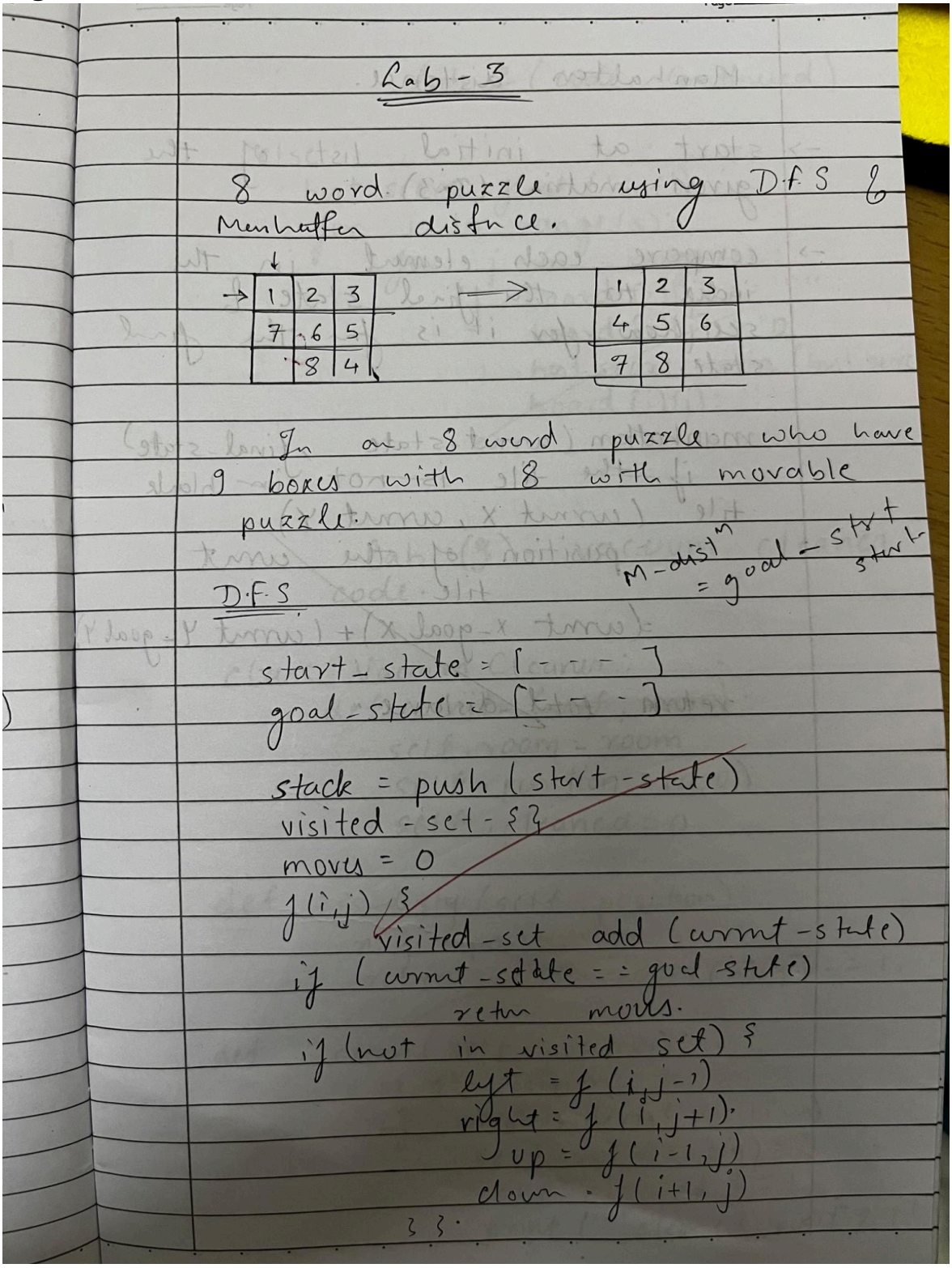
Output

```
Yashraj Sinha (1BM22CS335)
Welcome to Tic Tac Toe!
  |   |
---------
  |   |
---------
  |   |
---------
Enter the row (0, 1, 2): 0
Enter the column (0, 1, 2): 0
X |   |
---------
  |   |
---------
  |   |
---------
Computer chose: (1, 1)
X |   |
---------
  | O |
---------
  |   |
---------
Enter the row (0, 1, 2): 2
Enter the column (0, 1, 2): 1
X |   |
---------
  | O |
---------
  | X |
---------
Computer chose: (1, 0)
X |   |
---------
O | O |
---------
  | X |
---------
Enter the row (0, 1, 2): 1
Enter the column (0, 1, 2): 2
X |   |
---------
O | O | X
---------
  | X |
---------
Computer chose: (0, 2)
X |   | O
---------
O | O | X
---------
  | X |
---------
Enter the row (0, 1, 2): 2
```

```
Enter the row (0, 1, 2): 2
Enter the column (0, 1, 2): 0
X |   | O
---------
O | O | X
---------
X | X |
---------
Computer chose: (2, 2)
X |   | O
---------
O | O | X
---------
X | X | O
---------
Enter the row (0, 1, 2): 0
Enter the column (0, 1, 2): 1
X | X | O
---------
O | O | X
---------
X | X | O
---------
It's a tie!
```

Program 2 - 8 Puzzle BFS and DFS

Algorithm



Lab - 3

8 word puzzle using D.F.S & Manhattan distance.

1	2	3
7	6	5
	8	4

→

1	2	3
4	5	6
7	8	

In an 8 word puzzle who have 9 boxes with 8 with movable puzzle.

M-dist^n = goal − start

D.F.S

```
start_state = [- - - ]
goal_state = [- - - ]

stack = push (start_state)
visited_set = {}
moves = 0
f(i,j) {
    visited_set add (curnt_state)
    if (curnt_state == goal_state)
        return moves.
    if (not in visited set) {
        left = f(i,j-1)
        right = f(i,j+1).
        up = f(i-1,j)
        down = f(i+1,j)
    }}.
```

Date __/__/__
Page _____

Manhattan distance.

-> start at initial list of the given matrix (3×3).

-> compare each element in the index to the final state & see how far it is from the final state.

manhattan (curnt_state, final_state)
    if the tile is not in blank tile (curnt x, curnt y)
        = position of the curnt tile.
        = (curnt x - goal x) + (curnt y - goal y)

return total distance.

Code (BFS)

```
from collections import
deque
def is_solvable(state):
    inversions = 0
    flattened = [num for row
in state for num in row if
num != 0]
    for i in
range(len(flattened)):
        for j in range(i + 1,
len(flattened)):
            if flattened[i] >
flattened[j]:
                inversions += 1

    return inversions % 2
== 0


def print_state(state,
label=None):
    if label:
        print(label)
    for row in state:
        print(" ".join(str(num)
if num != 0 else " " for num
in row))
    print()


def get_neighbors(state):
    rows, cols = len(state),
len(state[0])
    zero_pos = None
    for r in range(rows):
        for c in range(cols):
            if state[r][c] == 0:
                zero_pos = (r, c)
                break
        if zero_pos:
            break

    directions = [(-1, 0), (1,
0), (0, -1), (0, 1)]
    neighbors = []
```

```
    for dr, dc in directions:
        nr, nc = zero_pos[0] +
dr, zero_pos[1] + dc
        if 0 <= nr < rows and
0 <= nc < cols:
            new_state = [row[:]
for row in state]

new_state[zero_pos[0]][zer
o_pos[1]],
new_state[nr][nc] = (

new_state[nr][nc],

new_state[zero_pos[0]][zer
o_pos[1]],
            )

neighbors.append(new_st
ate)

    return neighbors


def bfs(initial, goal):
    queue = deque([(initial,
[])])
    visited = set()

visited.add(tuple(tuple(row)
for row in initial))

    while queue:
        current, path =
queue.popleft()
        if current == goal:
            return path +
[current]

        for neighbor in
get_neighbors(current):
            neighbor_tuple =
tuple(tuple(row) for row in
neighbor)
            if neighbor_tuple
not in visited:

visited.add(neighbor_tuple)
```

```
queue.append((neighbor,
path + [current]))
    return None


def main():
    print("Yashraj Sinha
(1BM22CS335)")
    print("8-Puzzle Solver
Using BFS")
    initial_state = [[1, 2, 3],
[4, 0, 5], [7, 8, 6]]  #
Example initial state
    goal_state = [[1, 2, 3], [4,
5, 6], [7, 8, 0]]  # Goal state

    print_state(initial_state,
label="Initial State:")
    print_state(goal_state,
label="Goal State:")

    if not
is_solvable(initial_state):
        print("This puzzle is
not solvable.")
        return

    solution =
bfs(initial_state, goal_state)
    if solution:
        print("Solution found
in {}
steps:\n".format(len(solutio
n) - 1))
        for i, step in
enumerate(solution):
            if i == 0:
                print_state(step,
label="Initial State:")
            elif i ==
len(solution) - 1:
                print_state(step,
label="Final State:")
            else:
                print_state(step,
label=f"Step {i}:")
    else:
```

```
        print("No solution
exists.")


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

Code (DFS)

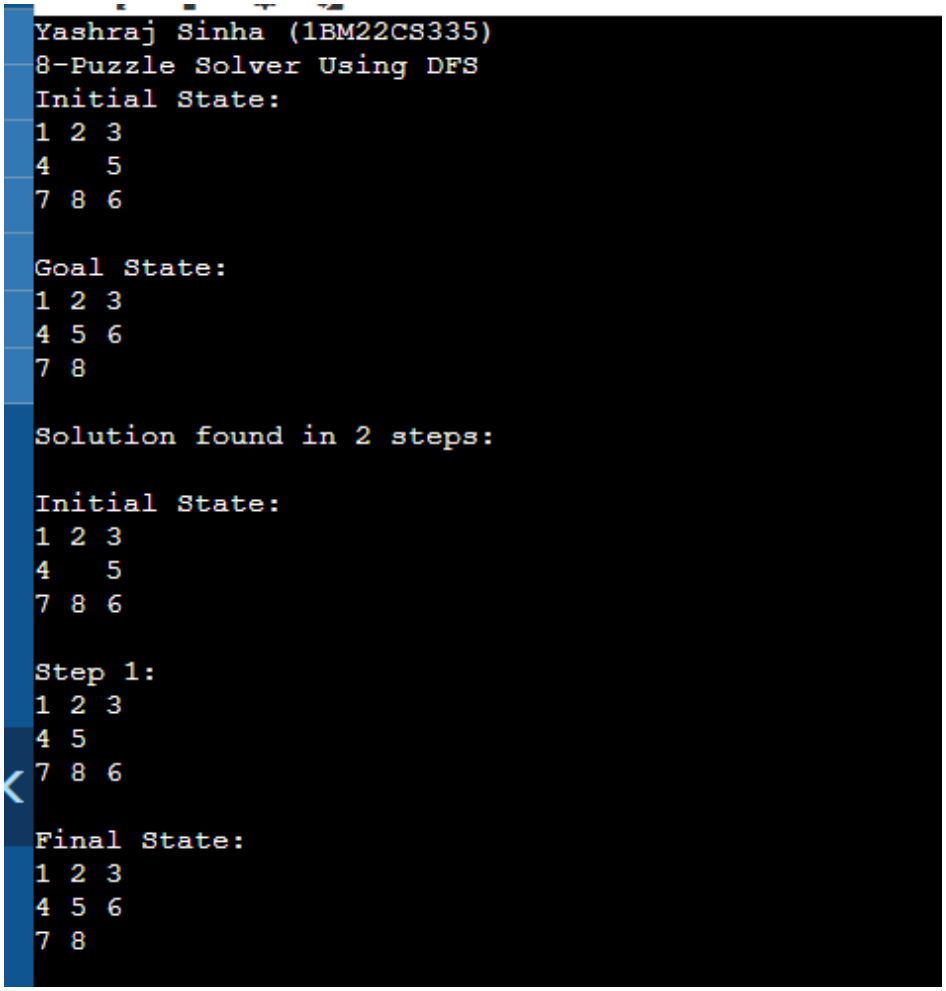
```
def is_solvable(state):
    inversions = 0
    flattened = [num for row in state for num in row if num != 0]
    for i in range(len(flattened)):
        for j in range(i + 1, len(flattened)):
            if flattened[i] > flattened[j]:
                inversions += 1

    return inversions % 2 == 0


def print_state(state, label=None):
    if label:
        print(label)
    for row in state:
        print(" ".join(str(num) if num != 0 else " " for num in row))
    print()


def get_neighbors(state):
    rows, cols = len(state), len(state[0])
    zero_pos = None
    for r in range(rows):
        for c in range(cols):
            if state[r][c] == 0:
                zero_pos = (r, c)
                break
        if zero_pos:
            break

    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    neighbors = []

    for dr, dc in directions:
        nr, nc = zero_pos[0] + dr, zero_pos[1] + dc
        if 0 <= nr < rows and 0 <= nc < cols:
            new_state = [row[:] for row in state]
            new_state[zero_pos[0]][zero_pos[1]], new_state[nr][nc] = (
                new_state[nr][nc],
                new_state[zero_pos[0]][zero_pos[1]],
            )
            neighbors.append(new_state)

    return neighbors
```

```
def dfs(initial, goal):
    stack = [(initial, [])]
    visited = set()
    visited.add(tuple(tuple(row) for row in initial))

    while stack:
        current, path = stack.pop()
        if current == goal:
            return path + [current]

        for neighbor in get_neighbors(current):
            neighbor_tuple = tuple(tuple(row) for row in neighbor)
            if neighbor_tuple not in visited:
                visited.add(neighbor_tuple)
                stack.append((neighbor, path + [current]))
    return None


def main():
    print("Yashraj Sinha (1BM22CS335)")
    print("8-Puzzle Solver Using DFS")
    initial_state = [[1, 2, 3], [4, 0, 5], [7, 8, 6]]  # Example initial state
    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]  # Goal state

    print_state(initial_state, label="Initial State:")
    print_state(goal_state, label="Goal State:")

    if not is_solvable(initial_state):
        print("This puzzle is not solvable.")
        return

    solution = dfs(initial_state, goal_state)
    if solution:
        print("Solution found in {} steps:\n".format(len(solution) - 1))
        for i, step in enumerate(solution):
            if i == 0:
                print_state(step, label="Initial State:")
            elif i == len(solution) - 1:
                print_state(step, label="Final State:")
            else:
                print_state(step, label=f"Step {i}:")
    else:
        print("No solution exists.")


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

Output (BFS)

```
Yashraj Sinha (1BM22CS335)
8-Puzzle Solver Using BFS
Initial State:
1 2 3
4   5
7 8 6

Goal State:
1 2 3
4 5 6
7 8

Solution found in 2 steps:

Initial State:
1 2 3
4   5
7 8 6

Step 1:
1 2 3
4 5
7 8 6

Final State:
1 2 3
4 5 6
7 8
```

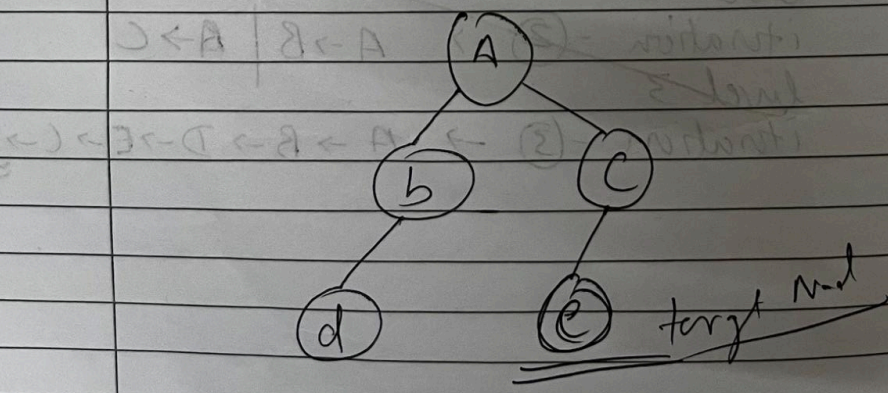
Output (DFS)


Yashraj Sinha (1BM22CS335)
8-Puzzle Solver Using DFS
Initial State:
1 2 3
4 5
7 8 6
Goal State:
1 2 3
4 5 6
7 8
Solution found in 2 steps:
Initial State:
1 2 3
4 5
7 8 6
Step 1:
1 2 3
4 5
7 8 6
Final State:
1 2 3
4 5 6
7 8

Program 3 - Iterative Deepening Search

Algorithm

```
bool IDDFS (src, target, max_depth)

for limit from 0 to max_depth
    if DLS (src, target, limit) == true
        return true
    return false.

bool DLS (src, target, limit)
    if (src == target) {
        return true;
    }

    if (limit <= 0) {
        return false;
    }

    for each adjacent i of SRC
        if DLS (i, target, limit-1)
        return true;

return false
```



Code

```
print("Yashraj Sinha (1BM22CS335)")
def is_solvable(state):
    inversions = 0
    flattened = [num for row in state for num in row if num !=
    0] for i in range(len(flattened)):
        for j in range(i + 1, len(flattened)):
            if flattened[i] > flattened[j]:
                inversions += 1
    return inversions % 2 == 0

def print_state(state, label=None):
    if label:
        print(label)
    for row in
    state:
        print(" ".join(str(num) if num != 0 else " " for num in
    row)) print()

def get_neighbors(state):
    rows, cols = len(state), len(state[0])
    for r in range(rows):
        for c in range(cols):
            if state[r][c] == 0:
                zero_pos = (r, c)
                break
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    neighbors = []
    for dr, dc in directions:
        nr, nc = zero_pos[0] + dr, zero_pos[1] + dc
        if 0 <= nr < rows and 0 <= nc < cols:
            new_state = [row[:] for row in state]
            new_state[zero_pos[0]][zero_pos[1]], new_state[nr][nc] = new_state[nr][nc],
new_state[zero_pos[0]][zero_pos[1]]
            neighbors.append(new_state)
    return neighbors
```

```
def ids(initial, goal, depth_limit):
    def dls(state, path, depth):
        if state == goal:
```

```
            return path + [state]
        if depth == 0:
            return None
        for neighbor in get_neighbors(state):
            if tuple(tuple(row) for row in neighbor) not in visited:
                visited.add(tuple(tuple(row) for row in neighbor))
                result = dls(neighbor, path + [state], depth - 1)
                if result:
                    return result
        return None

    for depth in range(depth_limit):
        visited = set()
        visited.add(tuple(tuple(row) for row in initial))
        result = dls(initial, [], depth)
        if result:
            return result
    return None

def main():
    print("8-Puzzle Solver Using Iterative Deepening Search")
    initial_state = [[1, 2, 3], [4, 0, 5], [7, 8, 6]] # Example initial state
    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]       # Goal state

    print_state(initial_state, label="Initial State:")
    print_state(goal_state, label="Goal State:")

    if not is_solvable(initial_state):
        print("This puzzle is not solvable.")
        return

    depth_limit = 20
    solution = ids(initial_state, goal_state, depth_limit)
    if solution:
        print("Solution found in {} steps:\n".format(len(solution) - 1))
        for i, step in enumerate(solution):
            if i == 0:
                print_state(step, label="Initial State:")
```

```
elif i == len(solution) - 1:
    print_state(step, label="Final State:")
else:
```

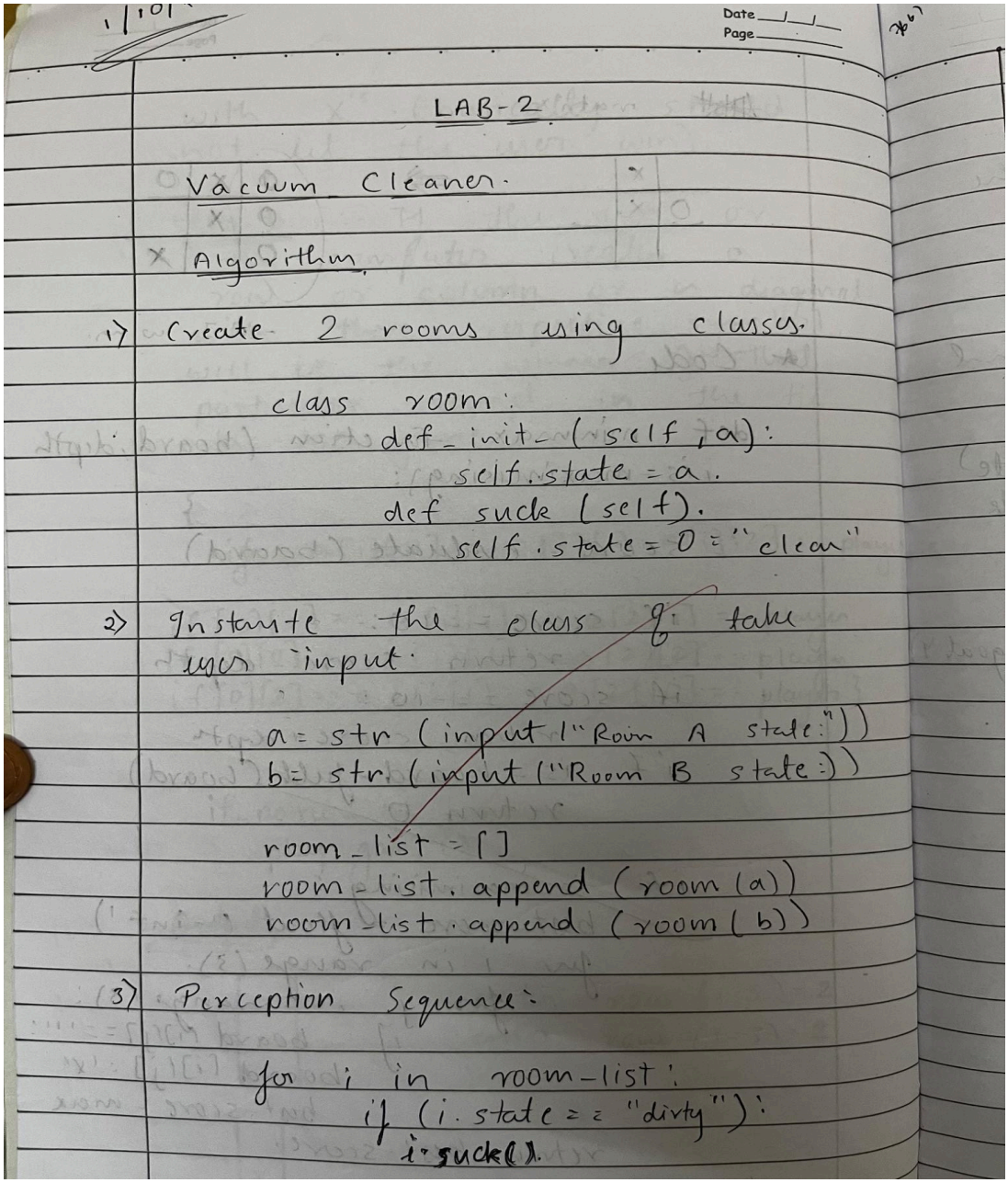
```
            print_state(step, label=f"Step {i}:")
    else:
        print("No solution exists within depth limit {}.".format(depth_limit))

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

Output

```
Yashraj Sinha (1BM22CS335)
8-Puzzle Solver Using Iterative Deepening Search
Initial State:
1 2 3
4   5
7 8 6

Goal State:
1 2 3
4 5 6
7 8

Solution found in 2 steps:

Initial State:
1 2 3
4   5
7 8 6

Step 1:
1 2 3
4 5
7 8 6

Final State:
1 2 3
4 5 6
7 8
```

Program 4 - Vacuum Cleaner Agent

Algorithm



LAB-2

Vacuum Cleaner.

Algorithm.

1) Create 2 rooms using classes.

```
class room:
    def __init__(self, a):
        self.state = a.
    def suck(self).
        self.state = 0 = "clean"
```

2) Instante the class & take user input.

```
a = str(input("Room A state:"))
b = str(input("Room B state:"))

room_list = []
room_list.append(room(a))
room_list.append(room(b))
```

3) Perception Sequence:

```
for i in room_list:
    if (i.state == "dirty"):
        i.suck().
```

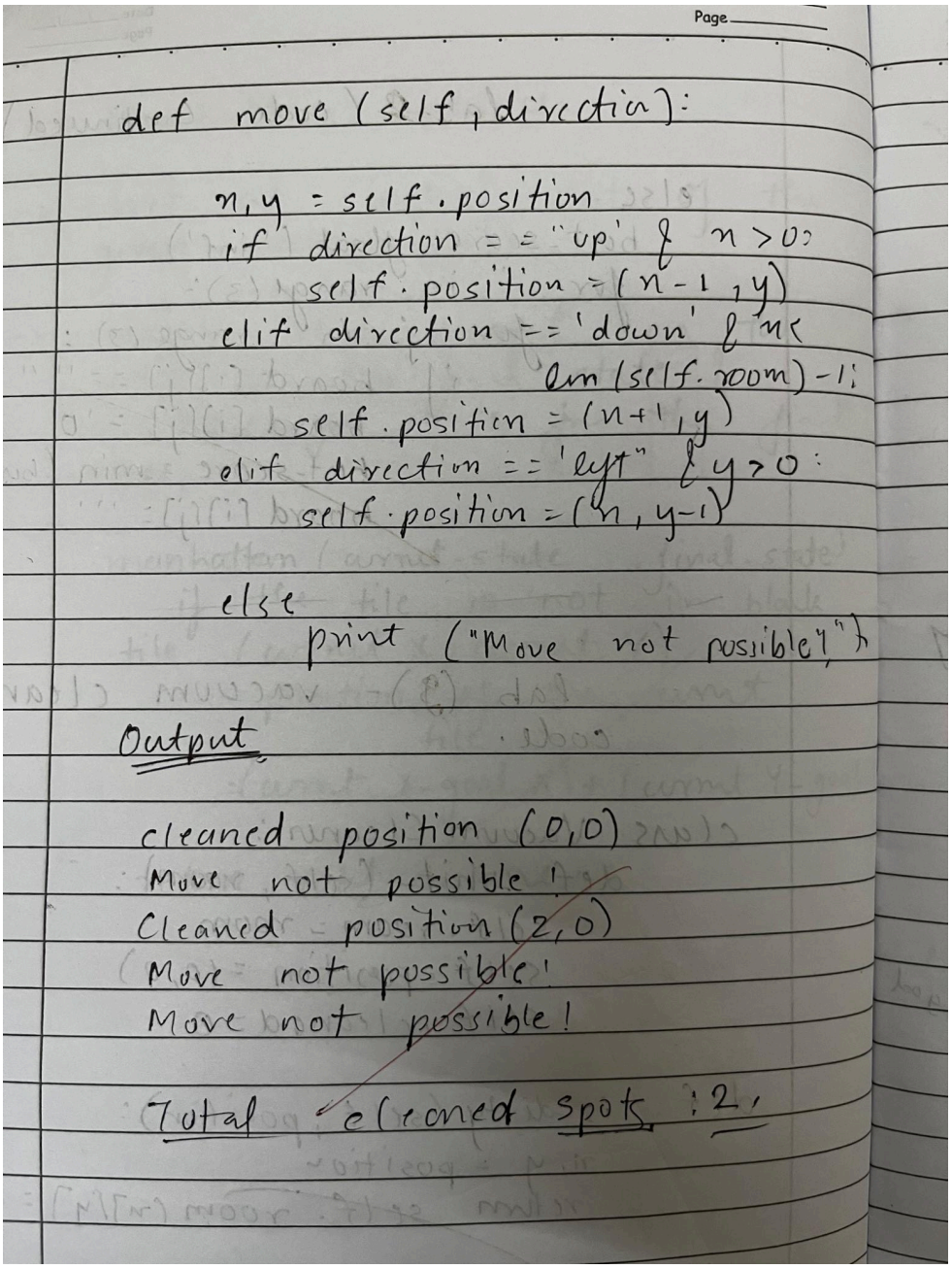
Date __/__/__
Page _____

lab-2 (code continued)

```
else:
    best_score = float('inf')
    for i in range(3):
        for j in range(3):
            if board[i][j] == '':
                board[i][j] = 'O'
                best_score = min(best_score
                board[i][j] = ''
```

lab (3)- vacuum cleaner code.

```
class VacuumCleaner:
    def __init__(self, room):
        self.room = room
        self.position = (0,0)
        self.cleaned = 0

def is_dirty(self, position):
    x, y = position
    return self.room[x][y] == 1.

def clean(self, position):
    x, y = position
    if self.is_dirty(position):
        self.room[x][y] = 0
        self.cleaned += 1
        print("cleaned position")
```



Code

print("Yashraj Sinha (1BM22CS335)")

def vacuum_cleaner(initial_state):

```
# Initial states of rooms A and

B room_A, room_B =

initial_state



# Trace of actions

actions = []



# Start in Room A

actions.append("Starting in Room

A.")



# Check room A

if room_A == 1:

    actions.append("Room A is dirty. Cleaning Room A.")

    room_A = 0

else:

    actions.append("Room A is already clean.")
```

```
# Move to Room B

actions.append("Moving to Room

B.")



# Check room B

if room_B == 1:

    actions.append("Room B is dirty. Cleaning Room B.")

    room_B = 0

else:

    actions.append("Room B is already clean.")



# Move back to Room A

actions.append("Returning to Room A.")



# Final state

final_state = (room_A, room_B)

actions.append("Both rooms are now clean.")
```

```
return final_state, actions
```

```
def main():
    print("Vacuum Cleaner AI")

    # Input initial states of Room A and Room B
    room_A = int(input("Enter the state of Room A (0 for clean, 1 for dirty): "))
    room_B = int(input("Enter the state of Room B (0 for clean, 1 for dirty): "))

    # Validate input
    if room_A not in (0, 1) or room_B not in (0, 1):
        print("Invalid input. Please enter 0 or 1.")
        return

    # Solve using vacuum cleaner AI
    final_state, actions = vacuum_cleaner((room_A, room_B))

    # Output actions and final state
    print("\nActions:")
```

```
for action in actions:
```

```
        print(action)



    print("\nFinal State:")
    print(f"Room A: {'Clean' if final_state[0] == 0 else 'Dirty'}")
    print(f"Room B: {'Clean' if final_state[1] == 0 else 'Dirty'}")



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

Output

```
Yashraj Sinha (1BM22CS335)
Vacuum Cleaner AI
Enter the state of Room A (0 for clean, 1 for dirty): 1
Enter the state of Room B (0 for clean, 1 for dirty): 1

Actions:
Starting in Room A.
Room A is dirty. Cleaning Room A.
Moving to Room B.
Room B is dirty. Cleaning Room B.
Returning to Room A.
Both rooms are now clean.

Final State:
Room A: Clean
Room B: Clean
```

Program 5 - A* Search Algorithm and Hill Climbing Algorithm

Algorithm

4) Implementing A* algorithm using python.

→ A*

function A* search (problem) return a solution or failure

node ← a node n with n state = problem.initial-state, neg = 0

frontier ← a priority queue ordered by ascending g+h, only element n.

loop do

if empty? (frontier) then return failure

n ← pop (frontier)

if problem.goalTest (n.state) then return solution(n)

for each action a in problem. actions (n.state) do

n' ← ChildNode (problem, n, a)

insert (n', g(n') + h(n'), frontier)

(i) Using no. of misplaced tiles as heuristic function

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

2	3	
4	5	6
7	8	

Initial State

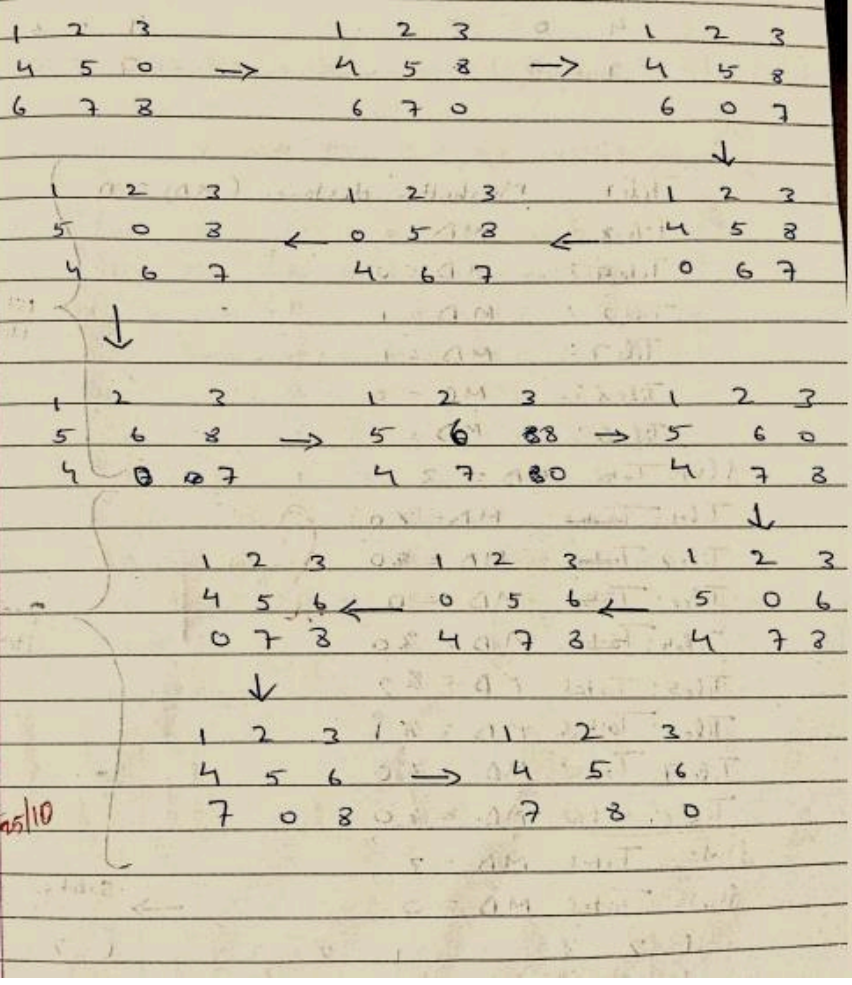
1	2	3
4	5	6
7	8	0

Goal state

Output

1	2	3
4	0	5
6	7	8

Initial state



25/10

ii) Using Manhattan distance

→ Initial state

1	2	3
4	0	5
7	8	6

Goal state

1	2	3
4	5	6
7	8	0

Tile 1 : Manhattan distance (MD) = 0
Tile 3 : MD = 0
Tile 4 : MD = 0
Tile 2 : MD = 1
Tile 7 : MD = 1
Tile 8 : MD = 0
Tile 6 : MD = 1
f(n) = Total MD = 3

1st iteration

Tile 1 : ~~Total~~ MD = 0
Tile 2 : Total MD = 0
Tile 3 : Total MD = 0
Tile 4 : Total MD = 0
Tile 5 : Total MD = 2
Tile 6 : Total MD = 1
Tile 7 : Total MD = 0
Tile 8 : Total MD = 0
f(n) = Total MD = 3
f(n) = Total MD = 0

2nd iteration

→ Solution in 3rd iteration

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

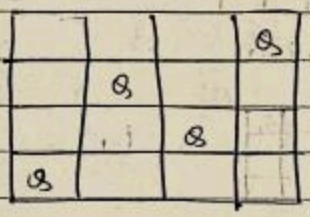
5) Implementing Hill Climbing search algorithm to solve N-Queens problem.

→ function Hill_Climbing (problem) returns a state that is local maximum

```
current ← Make-Node (problem, Initial-State)
loop do
    neighbour ← a highest value successor of current
    if neighbour.value ≤ current.value
        then return current.State
    current ← neighbour.
```

Execute

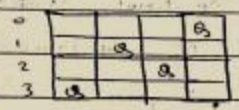
8) Show how the cost calculation of current state and neighbour nodes. And continue until you reach goal configuration of 4-Queens board.



→ State h(score)

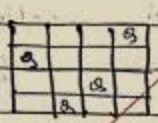
State	h(score)	
3,1,2,0	2	
1,3,2,0	1	← this
2,1,3,0	1	
0,1,2,3	6	
3,2,1,0	6	
3,0,2,1	1	
3,1,0,2	1	

State space :



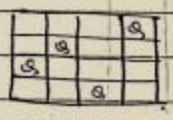
h=(2)

2,1,2,0



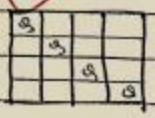
h=(1)

1,3,2,0



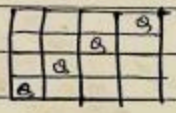
h=(1)

2,1,3,0



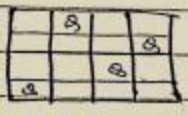
h=(6)

0,1,2,3



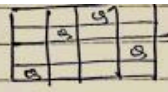
h=(6)

3,2,1,0



h=(1)

3,0,2,1

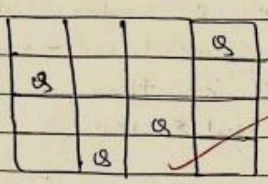


h = (1)

3, 1, 0, 2

• Next step

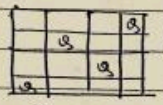
1, 3, 2, 0



h = (1)

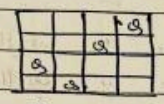
state	h (score)
1, 3, 2, 0	1
3, 1, 2, 0	2
2, 3, 1, 0	2
0, 3, 2, 1	3
1, 2, 3, 0	3
1, 3, 0, 2	0 ← return solution.

state space :



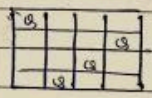
h = (2)

3, 1, 2, 0



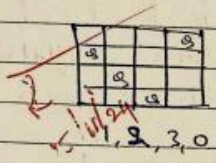
h = (2)

2, 3, 1, 0



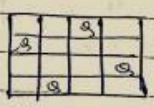
h = (3)

0, 3, 2, 1



h = (3)

1, 2, 3, 0



h = (0)

1, 3, 0, 2

Code (A* algorithm using N – displaced Tiles)

```
import heapq

 # Goal state

goal_state = (

    (1, 2, 3),

    (4, 5, 6),

    (7, 8, 0)

)



# Function to compute the heuristic (misplaced

tiles) def misplaced_tiles(state):

    misplaced = 0

    for i in range(3):

        for j in range(3):

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

                misplaced += 1

    return misplaced
```

```
# Function to get possible moves (neighbors)
```

```
def get_neighbors(state):
    neighbors = []
    zero_pos = [(i, j) for i in range(3) for j in range(3) if state[i][j] == 0][0]
    i, j = zero_pos

    # Possible moves: up, down, left, right
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for move in moves:
        new_i, new_j = i + move[0], j + move[1]
        if 0 <= new_i < 3 and 0 <= new_j < 3:
            new_state = list(list(row) for row in state) # Create a copy of the state
            new_state[i][j], new_state[new_i][new_j] = new_state[new_i][new_j], new_state[i][j]
            neighbors.append(tuple(tuple(row) for row in new_state)) # Convert back to tuple
    return neighbors

# Function to count the number of inversions in the
puzzle def count_inversions(state):
    one_d_state = [tile for row in state for tile in row if tile != 0]
```

inversions = 0

```
    for i in range(len(one_d_state)):
        for j in range(i + 1, len(one_d_state)):
            if one_d_state[i] > one_d_state[j]:
                inversions += 1
    return inversions


# Check if the puzzle is solvable
def is_solvable(state):
    inversions = count_inversions(state)
    return inversions % 2 == 0


# A* Algorithm
def a_star(initial_state):
    if not is_solvable(initial_state):
        print("This puzzle is not solvable.")
        return None

    open_list = []
```

```
    heapq.heappush(open_list, (0 + misplaced_tiles(initial_state), 0, initial_state, [])) # (f(n), g(n), state, path)
```

```
closed_list = set()



while open_list:
    f, g, current_state, path =
    heapq.heappop(open_list)
    closed_list.add(current_state)



    # If goal state is reached
    if current_state == goal_state:
        return path + [current_state]



    # Generate neighbors
    for neighbor in get_neighbors(current_state):
        if neighbor not in closed_list:
            heapq.heappush(open_list, (
                g + 1 + misplaced_tiles(neighbor), # f(n) = g(n) + h(n)
                g + 1, # Increment g(n) by 1 for each move
                neighbor,
```

```
    path + [current_state]

))
```

```
    return None # No solution found



# Function to display the puzzle state

def display_state(state, label):

    print(f"{label}

    state:") for row in

    state:

        print(" ".join(str(x) for x in row))

    print()



# Example initial state (this one is solvable)

initial_state = (

    (1, 2, 3),

    (5, 6, 4),

    (7, 8, 0)

)
```

```
# Solving the puzzle

solution = a_star(initial_state)
```

```
# Displaying the result
if solution:
    # Print Yashraj's information
    print("Yashraj Sinha (1BM22CS335)\n")

    # Print the initial state
    display_state(initial_state, "Initial")

    # Print the final state
    display_state(goal_state, "Goal")

    # Displaying the solution path
    print("Solution path:")
    for step in solution:
        display_state(step,
        "Step")
else:
```

```
print("No solution found.")
```

Code (A* algorithm using Manhattan distance)

```
import
heapq



# Goal state

goal_state = (

    (1, 2, 3),

    (4, 5, 6),

    (7, 8, 0)

)



# Function to compute the Manhattan distance heuristic

def manhattan_distance(state):

    distance = 0

    for i in range(3):

        for j in range(3):

            tile = state[i][j]

            if tile != 0:

                goal_i, goal_j = divmod(tile - 1, 3)
```

```
distance += abs(goal_i - i) + abs(goal_j - j)
```

```
    return distance



# Function to get possible moves (neighbors)

def get_neighbors(state):

    neighbors = []

    zero_pos = [(i, j) for i in range(3) for j in range(3) if state[i][j] == 0][0]

    i, j = zero_pos



    # Possible moves: up, down, left, right

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

    for move in moves:

        new_i, new_j = i + move[0], j + move[1]

        if 0 <= new_i < 3 and 0 <= new_j < 3:

            new_state = list(list(row) for row in state) # Create a copy of the state

            new_state[i][j], new_state[new_i][new_j] = new_state[new_i][new_j], new_state[i][j]

            neighbors.append(tuple(tuple(row) for row in new_state)) # Convert back to tuple

    return neighbors
```

```
# Function to count the number of inversions in the puzzle
```

```
def count_inversions(state):
    one_d_state = [tile for row in state for tile in row if tile != 0]
    inversions = 0
    for i in range(len(one_d_state)):
        for j in range(i + 1, len(one_d_state)):
            if one_d_state[i] > one_d_state[j]:
                inversions += 1
    return inversions


# Check if the puzzle is solvable
def is_solvable(state):
    inversions = count_inversions(state)
    return inversions % 2 == 0


# A* Algorithm
def a_star(initial_state):
    if not is_solvable(initial_state):
        print("This puzzle is not solvable.")
```

```
return None
```

```
    open_list = []

    heapq.heappush(open_list, (0 + manhattan_distance(initial_state), 0, initial_state, [])) # (f(n), g(n),
state, path)

    closed_list = set()



    while open_list:

        f, g, current_state, path =

        heapq.heappop(open_list)

        closed_list.add(current_state)



        # Print the current state and its f(n) value

        print(f"State: {current_state}")

        print(f"f(n) = g(n) + h(n) = {g} + {manhattan_distance(current_state)} = {f}")

        print()



        # If goal state is reached

        if current_state == goal_state:

            return path + [current_state]
```

Generate neighbors

```
        for neighbor in get_neighbors(current_state):
            if neighbor not in closed_list:
                heapq.heappush(open_list, (
                    g + 1 + manhattan_distance(neighbor), # f(n) = g(n) + h(n) g
                    + 1, # Increment g(n) by 1 for each move
                    neighbor,
                    path + [current_state]
                ))

    return None # No solution found

# Function to display the puzzle state
def display_state(state, label):
    print(f"{label}
    state:") for row in
    state:
        print(" ".join(str(x) for x in row))
    print()
```

```
# Example initial state (this one is solvable)
```

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

# Solving the puzzle
solution = a_star(initial_state)

# Displaying the result
if solution:
    # Print Yashraj's information
    print("Yashraj Sinha (1BM22CS335)\n")

    # Print the initial state
    display_state(initial_state, "Initial")

    # Print the final state
```

```
display_state(goal_state, "Goal")
```

```
    # Displaying the solution path
    print("Solution path:")
    for step in solution:
        display_state(step,
        "Step")
else:
    print("No solution found.")
```

Code (Hill Climbing algorithm)

```
import random
print("Yashraj Sinha (1BM22CS335)")
# Function to calculate the number of attacking pairs of
queens def calculate_attacks(board):
    attacks = 0
    n = len(board)
    for i in range(n):
```

```
for j in range(i + 1, n):

    if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
```

```
            attacks +=

    1 return attacks



# Function to generate a random initial state

def generate_initial_state(n):

    return [random.randint(0, n - 1) for _ in range(n)]



# Function to generate neighbors by moving one queen to a different row

def generate_neighbors(board):

    neighbors = []

    n = len(board)

    for col in range(n): for

        row in range(n):

            if row != board[col]: # Make sure we are not moving the queen to its current row

                neighbor = board[:]

                neighbor[col] = row

                neighbors.append(neighbor)

    return neighbors
```

```
# Hill Climbing algorithm with random restarts

def hill_climbing(n, max_restarts=100):

    for restart in range(max_restarts):

        current_state = generate_initial_state(n)

        current_attacks = calculate_attacks(current_state)



        while True:

            # Generate all neighbors

            neighbors = generate_neighbors(current_state)



            # Find the neighbor with the minimum number of attacks

            next_state = None

            next_attacks = current_attacks



            for neighbor in neighbors:

                attacks = calculate_attacks(neighbor)

                if attacks < next_attacks:

                    next_state = neighbor
```

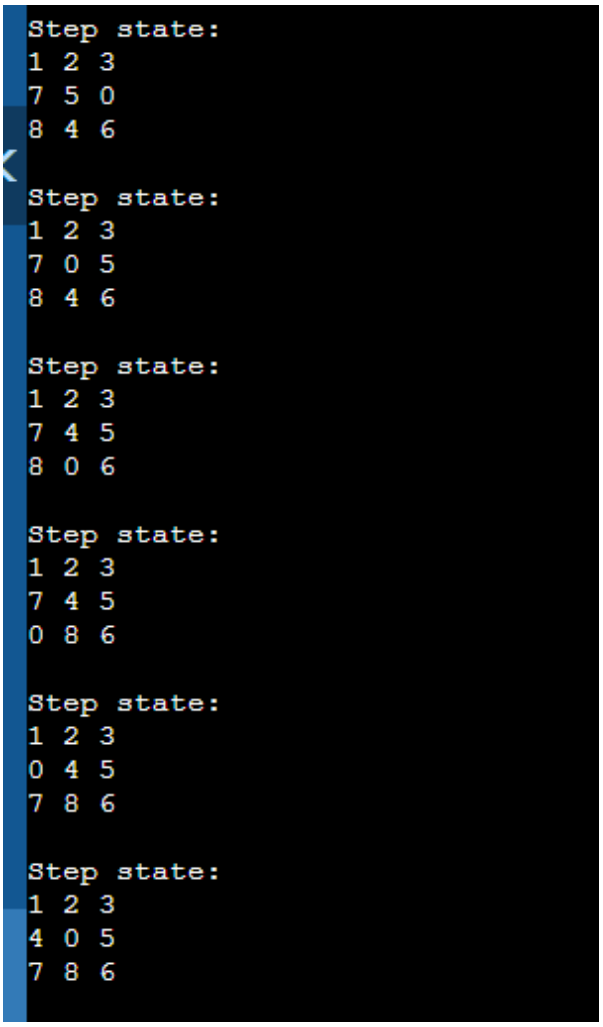
```
next_attacks = attacks
```

```
            # If no improvement, return the solution or
            terminate if next_attacks == current_attacks:
                break

            current_state = next_state
            current_attacks = next_attacks

        # If a solution is found, return the current
        state if current_attacks == 0:
            return current_state

    # If no solution found after max_restarts, return None
    return None

# Function to display the board
def display_board(board):
    n = len(board)
```

```
for i in range(n):
```

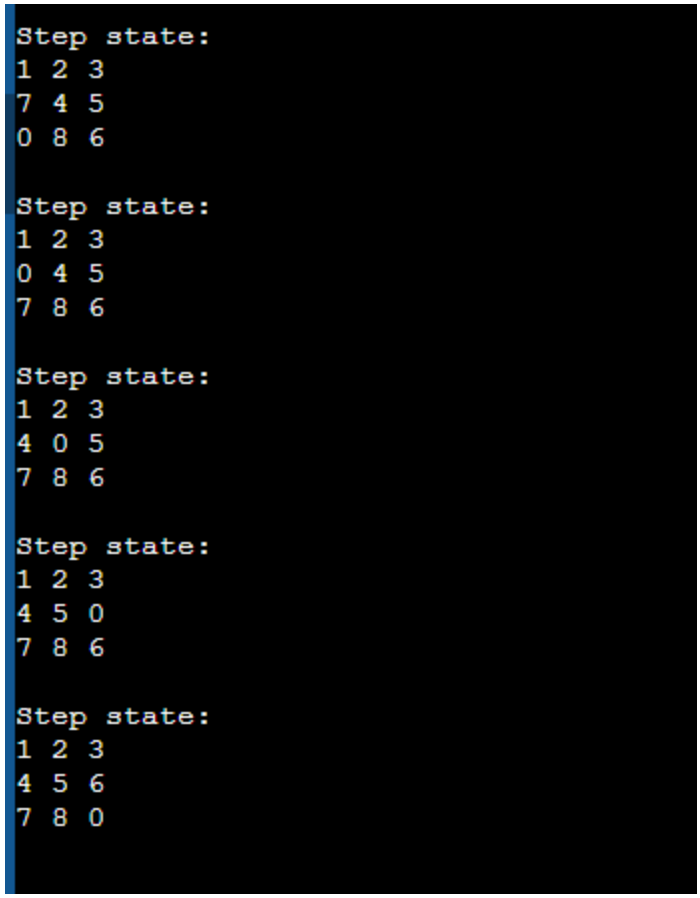
```
        row = ['Q' if i == board[col] else '.' for col in

        range(n)] print(' '.join(row))

    print()



# Set the size of the board (N)

N = 8



# Solve the N-Queens problem with random restarts

solution = hill_climbing(N)



# Display the result

if solution:

    print(f"Solution for {N}-Queens:")

    display_board(solution)

else:

    print(f"No solution found for {N}-Queens.")
```

```
Yashraj Sinha (1BM22CS335)

Initial state:
1 2 3
5 6 4
7 8 0

Goal state:
1 2 3
4 5 6
7 8 0

Solution path:
Step state:
1 2 3
5 6 4
7 8 0

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

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

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

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

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

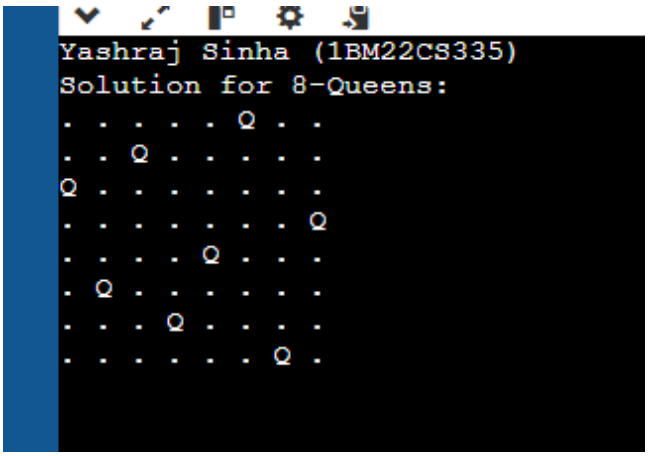

Step state:
1 2 3
7 5 0
8 4 6
Step state:
1 2 3
7 0 5
8 4 6
Step state:
1 2 3
7 4 5
8 0 6
Step state:
1 2 3
7 4 5
0 8 6
Step state:
1 2 3
0 4 5
7 8 6
Step state:
1 2 3
4 0 5
7 8 6

Output (Manhattan Distance)

```
Yashraj Sinha (1BM22CS335)

Initial state:
1 2 3
5 6 4
7 8 0

Goal state:
1 2 3
4 5 6
7 8 0

Solution path:
Step state:
1 2 3
5 6 4
7 8 0

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

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

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

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

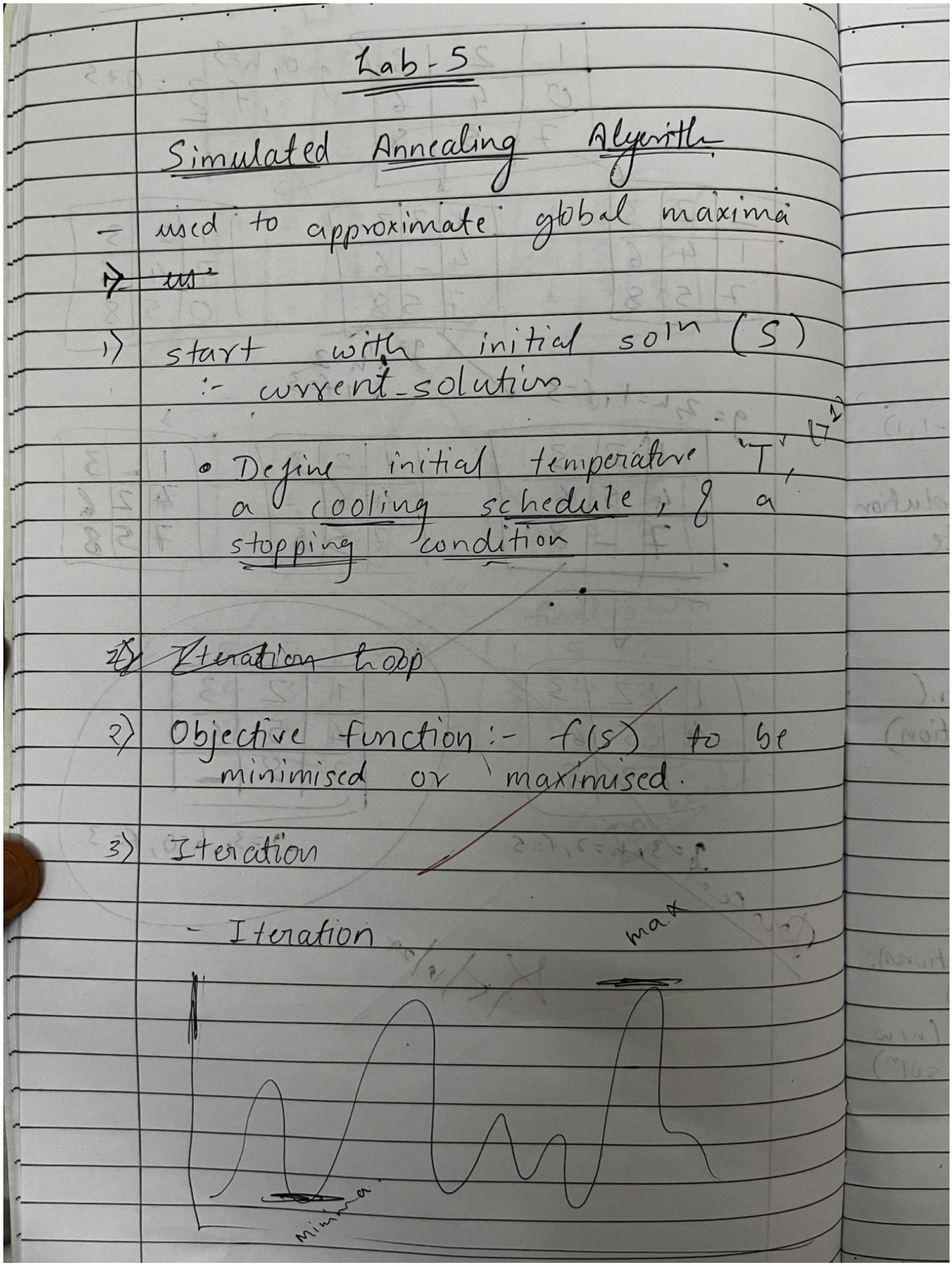

Step state:
1 2 3
7 4 5
0 8 6
Step state:
1 2 3
0 4 5
7 8 6
Step state:
1 2 3
4 0 5
7 8 6
Step state:
1 2 3
4 5 0
7 8 6
Step state:
1 2 3
4 5 6
7 8 0

Output (Hill Climbing)



Program 6 - Simulated Annealing

Algorithm



- Generate a New soln
  - perturb $S \rightarrow S'$
  - evaluate $\Delta E = f(s') - f(S)$
  - Acceptance:

    if $\Delta E < 0$, accept s' as bettn

    if $\Delta E > 0$

    - accept s' with prob. $P = e^{-\Delta E / T}$
    - this allows accepting worse sol' to escape local maxima
  - Update
    - set $S = s'$ if s' is accpted.
    - Update T according to cooling schedule

4) Keep track of best soln & retn it.

Proceed

Code

```
import random

import math

print("Yashraj Sinha (1BM22CS335)")

# Objective function: count the number of attacking pairs of queen
```

```
def calculate_attacks(board):
attacks = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            # Check if two queens are in the same row, column, or
            diagonal if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                attacks += 1
    return attacks


# Function to generate a random initial state (random queen positions in each column)
def generate_initial_state(n):
    return [random.randint(0, n - 1) for _ in range(n)]


# Function to generate a neighboring solution by moving one queen in a column
def generate_neighbor(board):
    neighbor = board[:]
    column = random.randint(0, len(board) - 1)
```

Randomly select a new row for the queen in the chosen column

```
    neighbor[column] = random.randint(0, len(board) - 1)
    return neighbor

# Simulated Annealing algorithm to solve the N-Queens problem
def simulated_annealing(n, max_iterations, initial_temperature, cooling_rate):
    current_state = generate_initial_state(n)
    current_attacks = calculate_attacks(current_state)
    temperature = initial_temperature

    best_state = current_state
    best_attacks = current_attacks

    for iteration in range(max_iterations):
        # Generate a neighbor solution
        neighbor = generate_neighbor(current_state)
        neighbor_attacks = calculate_attacks(neighbor)

        # Calculate the energy difference (how much worse the new state is)
```

```
delta_attacks = neighbor_attacks - current_attacks
```

```
# Accept the neighbor if it has fewer attacks or with a probability if it's worse
if delta_attacks < 0 or random.random() < math.exp(-delta_attacks / temperature):
    current_state = neighbor
    current_attacks = neighbor_attacks

# Update the best solution if
necessary if current_attacks <
best_attacks:
    best_state = current_state
    best_attacks = current_attacks

# Cool down the temperature
temperature *= cooling_rate

# If no attacks, we found the
solution if best_attacks == 0:
    break
```

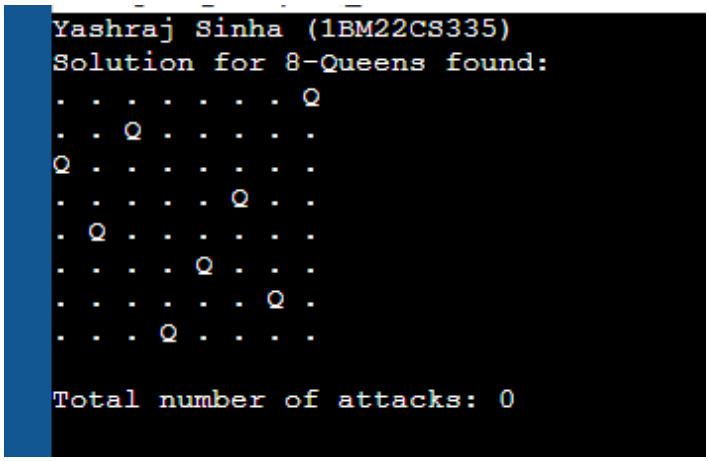
```
return best_state, best_attacks
```

```
# Function to display the board (where 'Q' is a queen and '.' is an empty space)

def display_board(board):

    n = len(board)

    for i in range(n):

        row = ['Q' if i == board[col] else '.' for col in

        range(n)] print(' '.join(row))

    print()



# Parameters for Simulated Annealing

N = 8 # Set the size of the board (N x N)

max_iterations = 10000 # Higher number of iterations for better convergence

initial_temperature = 1000 # High initial temperature

cooling_rate = 0.995 # Cooling rate (temperature decreases by 0.5% every iteration)



# Solve the N-Queens problem using Simulated Annealing

solution, attacks = simulated_annealing(N, max_iterations, initial_temperature, cooling_rate)
```

Output the result

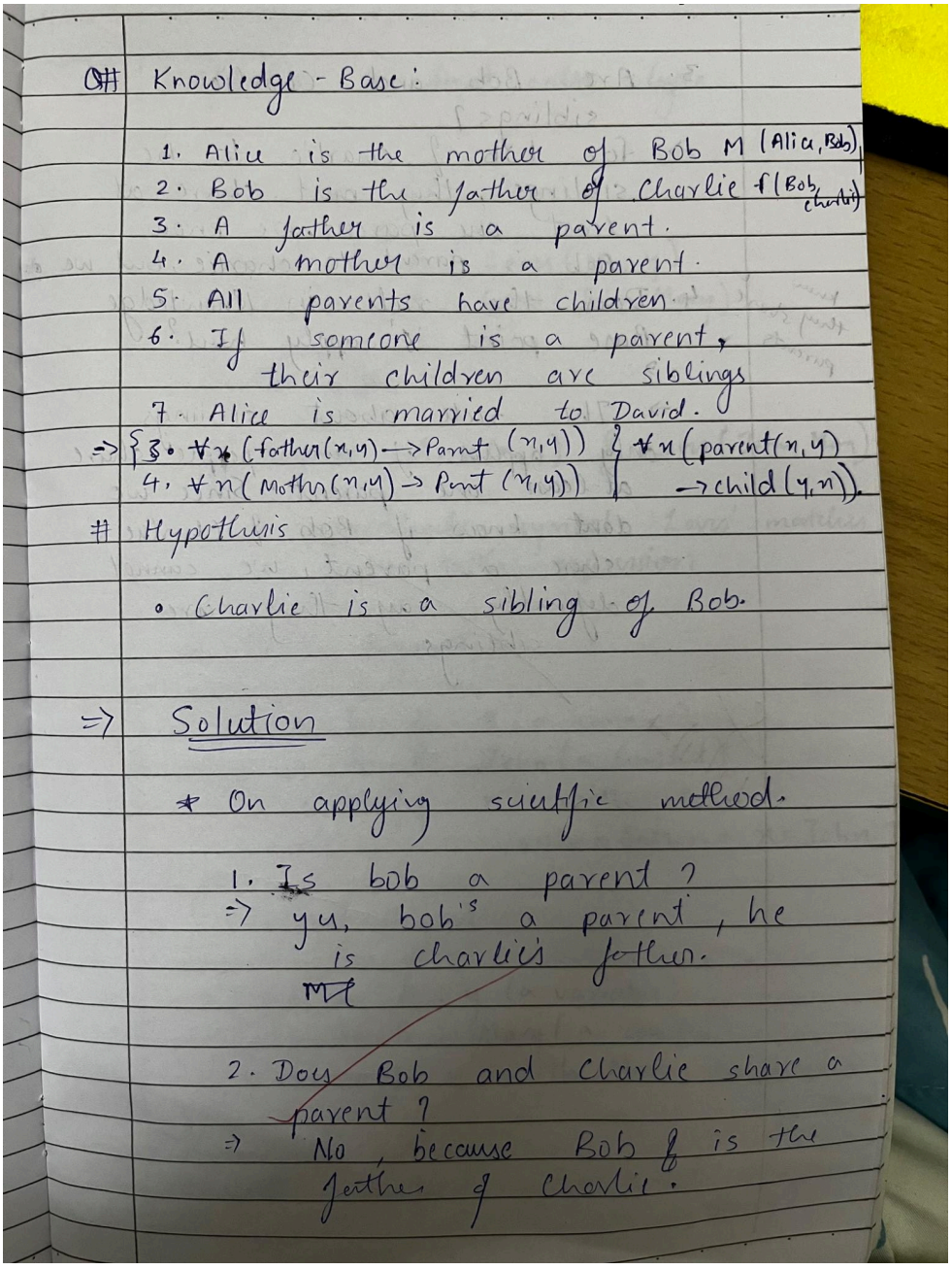
```
print(f"Solution for {N}-Queens found:")

display_board(solution)

print(f"Total number of attacks: {attacks}")
```

Output


Yashraj Sinha (1BM22CS335)
Solution for 8-Queens found:
. . . . Q
. . Q . . . . .
Q . . . .
. . . . . Q . .
. Q . . . . . .
. . . . Q . . .
. . . . . . Q .
. . . Q . . . .
Total number of attacks: 0

Program 7 - Knowledge base using propositional logic and show that the given query entails the knowledge base or not.

Algorithm



Code

```
from sympy.logic.boolalg import Or, And,
Not from sympy.abc import A, B, C, D, E, F
57from sympy import simplify_logic
def is_entailment(kb, query):
# Negate the query
negated_query = Not(query)
# Add negated query to the knowledge base
kb_with_negated_query = And(*kb, negated_query)
# Simplify the combined KB to CNF
simplified_kb = simplify_logic(kb_with_negated_query,
form="cnf") # If the simplified KB evaluates to False, the query is
entailed return simplified_kb == False
# Define a larger Knowledge Base
kb = [
Or(A, B),
# A ∨ B
Or(Not(A), C), # ¬A ∨ C
Or(Not(B), D), # ¬B ∨ D
Or(Not(D), E), # ¬D ∨ E
Or(Not(E), F), # ¬E ∨ F
F
# F
]
# Query to check
query = Or(C, F) # C ∨ F
# Check entailment
result = is_entailment(kb, query)
print(f"Is the query '{query}' entailed by the knowledge base? {'Yes' if result else 'No'}")
```

OUTPUT:

Is the query 'C | F' entailed by the knowledge base? Yes

Code

```
Keep track of all unique
    clauses in the knowledge
    base
    print(f"Initial Knowledge
    Base + negation of query:
    {kb}")

    while True:
        added_new_clause =
    False

        # Try to resolve every
    pair of clauses
        clauses =
    list(new_clauses)
        for i in
    range(len(clauses)):
            for j in range(i + 1,
    len(clauses)):
                clause1 = clauses[i]
                clause2 = clauses[j]

                # Try to resolve
    these two clauses
                resolvent =
    resolve(clause1, clause2)

                if resolvent is not
    None:
                    print(f"Resolving
    clauses: {clause1} and
    {clause2}")
                    print(f"Resolved
    to: {resolvent}")

                    # If resolvent is
    empty, we found a
    contradiction
                    if not resolvent:
```

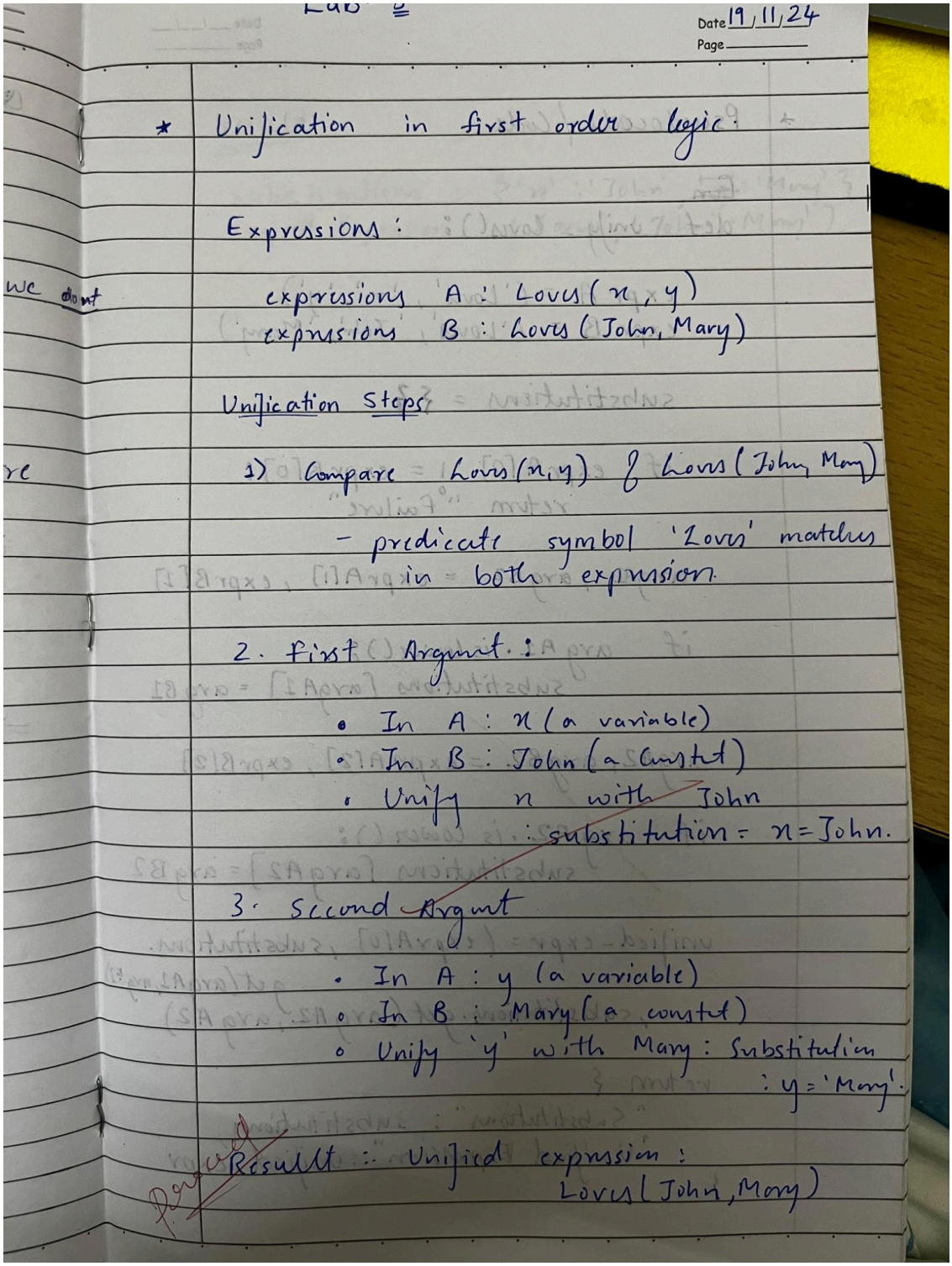
```
                    return True  #
    Found a contradiction, so
    the query is provable

                # Add the new
    clause if it's not already in
    the set
                if resolvent not in
    new_clauses:

    new_clauses.add(resolvent)

    added_new_clause = True

        # If no new clause was
    added, resolution has
    terminated without a
    contradiction
        if not
    added_new_clause:
            break

    return False  # No
    contradiction found, so the
    query is not provable


def resolve(clause1, clause2):
    """Resolve two clauses if
    possible and return the
    resolvent."""
    # Split clauses into literals
    literals1 =
    set(clause1.split(" v "))
    literals2 =
    set(clause2.split(" v "))

    # Try to find
    complementary literals
    for literal in literals1:
        neg_literal =
    negation(literal)
```

```
        if neg_literal in literals2:
            # Resolve the two
    clauses by removing
    complementary literals
            new_clause =
    literals1.union(literals2) -
    {literal, neg_literal}
            return " v
    ".join(sorted(new_clause))
    # Return the resolved
    clause as a string

    return None  # No resolvent
    found


# Example knowledge base
    and query (where T is
    provable)
kb = [
    "P v Q",     # P or Q
    "~P v R",    # Not P or R
    "Q v ~R",    # Q or Not R
    "R v T"      # R or T
]



query = "T"  # Query to prove
    (e.g., prove T)

# Perform resolution to prove
    the query
result = resolution(kb, query)

if result:
    print(f"\nQuery '{query}' is
    provable from the
    knowledge base.")
else:
    print(f"\nQuery '{query}' is
```

```
    not provable from the
    knowledge base.")


def negation(p):
    """Negate a literal."""
    if p.startswith("~"):
        return p[1:]  # Remove
    the '~' from negated literals
    return f"~{p}"


def resolution(kb, query):
    """Perform resolution on
    the knowledge base to
    prove the query."""

    # Add the negation of the
    query to the knowledge
    base (for proof by
    contradiction)
    kb.append(negation(query))

    # Apply the resolution rule
    until we reach an empty
    clause (which means
    contradiction)
    new_clauses = set(kb)  #
```

Program 8- Unification

Algorithm:



Lab 8

Date 19/11/24
Page

* Unification in first order logic:

Expressions:

expressions A : Loves(x, y)
expressions B : Loves(John, Mary)

Unification Steps:

1) Compare Loves(x,y) & Loves(John, Mary)
- predicate symbol 'Loves' matches in both expression.

2. first Argmt.:
- In A : x (a variable)
- In B : John (a Constt)
- Unify x with John
: substitution = x=John.

3. Second Argmt
- In A : y (a variable)
- In B : Mary (a constt)
- Unify 'y' with Mary : Substitution
: y='Mary'.

Result : Unified expression :
Loves(John, Mary)

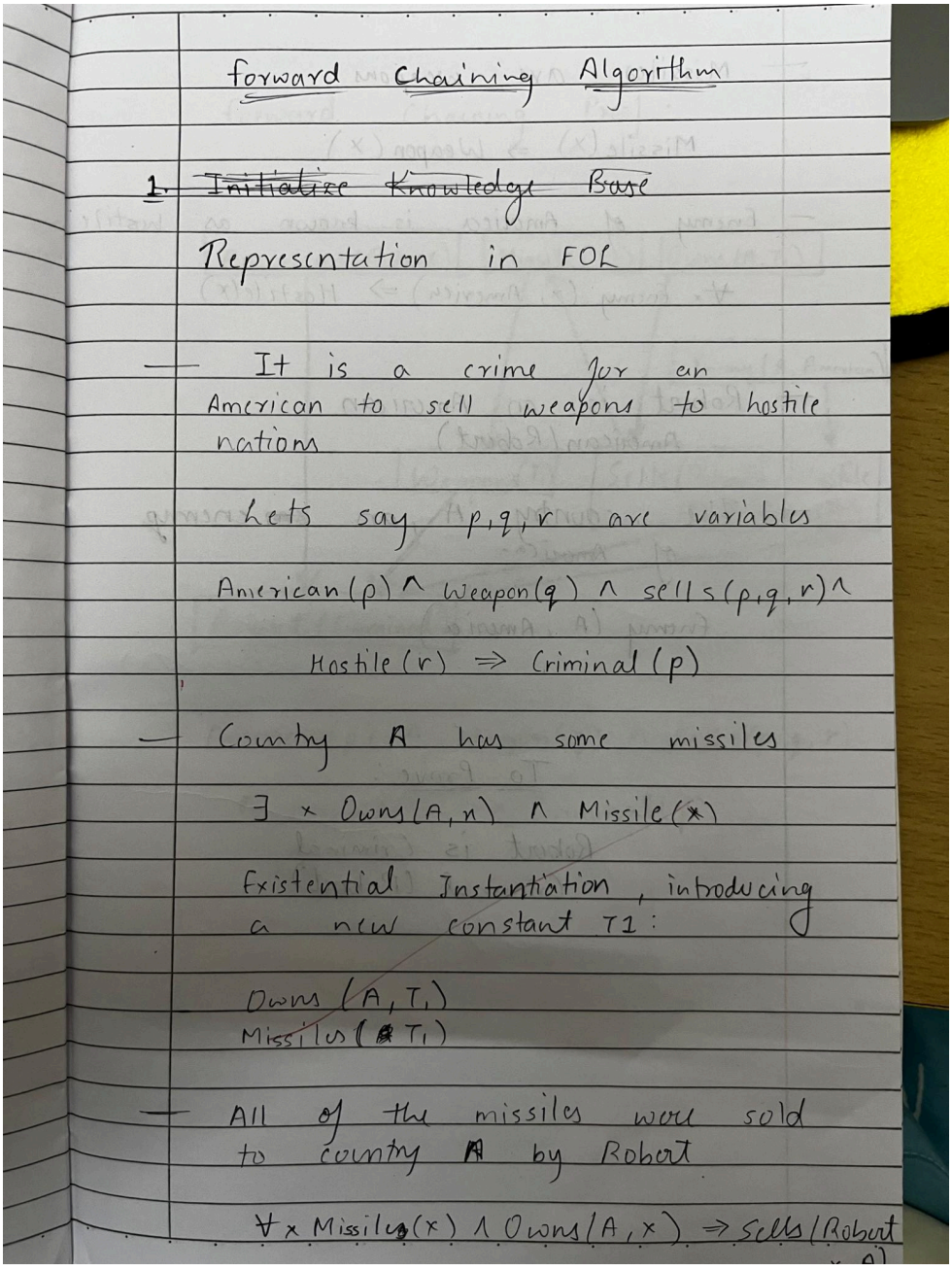
CODE:

```
def is_variable(term):
    """Check if a term is a variable (starts with an uppercase letter)."""
    return isinstance(term, str) and term[0].isupper()


def unify(x, y, subst):
    """
    Perform unification of two terms x and y under a given substitution subst.
    """
    if subst is None:  # If unification fails, return None
        return None
    elif x == y:  # If terms are identical, return the current substitution
        return subst
    elif is_variable(x):  # If x is a variable, unify it with y
        return unify_variable(x, y, subst)
    elif is_variable(y):  # If y is a variable, unify it with x
        return unify_variable(y, x, subst)
    elif isinstance(x, tuple) and isinstance(y, tuple):  # If both are compound terms
        if len(x) != len(y):
            return None
        for xi, yi in zip(x, y):
            subst = unify(xi, yi, subst)
        return subst
    else:
        return None


def unify_variable(var, term, subst):
    """
    Unify a variable with a term under a substitution subst.
    """
    if var in subst:
        return unify(subst[var], term, subst)
    elif term in subst:
        return unify(var, subst[term], subst)
    elif occurs_check(var, term, subst):
        return None
    else:
        subst[var] = term
        return subst


def occurs_check(var, term, subst):
    """
    Check for circular references in unification (occurs check).
    """
```

```
    if var == term:
        return True
    elif isinstance(term, tuple):
        return any(occurs_check(var, t, subst) for t in term)
    elif term in subst:
        return occurs_check(var, subst[term], subst)
    return False


def unify_terms(term1, term2):
    """
    Top-level function for unification.
    Returns the substitution if unification succeeds, else None.
    """
    return unify(term1, term2, {})


# Example usage
if __name__ == "__main__":
    # Terms are represented as tuples
    term1 = ("f", "X", "a")  # f(X, a)
    term2 = ("f", "b", "a")  # f(b, a)

    result = unify_terms(term1, term2)

    if result:
        print("Unification succeeded!")
        print("Substitution:", result)
    else:
        print("Unification failed.")
```

Program 9 - Knowledge base consisting of first order logic statements and prove the given query using forward reasoning..

Algorithm



Forward Chaining Algorithm

1. ~~Initialize Knowledge Base~~

Representation in FOL

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

lets say p, q, r are variables

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

— County A has some missiles

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

Existential Instantiation, introducing a new constant T1:

$Owns(A,T_1)$
$Missiles(T_1)$

— All of the missiles were sold to county A by Robert

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

- Missiles are weapons

  Missile(x) ⇒ Weapon(x)

- Enemy of America is known as hostile

  ∀x Enemy(x, America) ⇒ Hostile(x)

- Robert is an American

  American(Robert)

- The country A, an ~~em~~ enemy of America.

  Enemy(A, America)

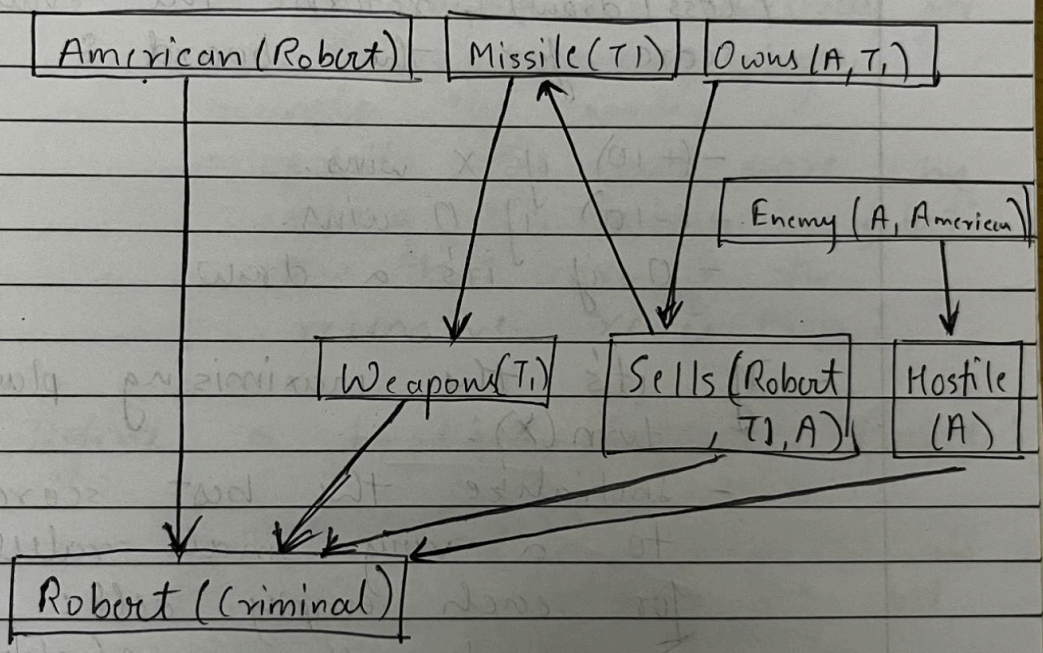
To Prove:

Robert is Criminal

Criminal(Robert)

Date 3/12/24
Page

Forward Chaining Proof:



American(p) ∧ Weapon(q) ∧ Sells(p, q, r)

∧ Hostile(r) => Criminal(p)

Code

```
knowledge_base = {
"facts": {
"American(Robert)": True,
"Enemy(A, America)": True,
"Owns(A, T1)": True,
"Missile(T1)": True,
},
"rules": [
{"if": ["Missile(x)"], "then": ["Weapon(x)"]},
{"if": ["Enemy(x, America)"], "then": ["Hostile(x)"]},
{"if": ["Missile(x)", "Owns(A, x)"], "then": ["Sells(Robert, x, A)"]},
{
"if": ["American(p)", "Weapon(q)", "Sells(p, q, r)", "Hostile(r)"],
"then": ["Criminal(p)"],
},
],
}
def forward_chaining(kb):
facts =
kb["facts"].copy() rules =
kb["rules"] inferred =
set()
while True:
43new_inferences = set()
for rule in rules:
if_conditions = rule["if"]
then_conditions = rule["then"]
substitutions = {}
all_conditions_met = True
for condition in if_conditions:
predicate, args =
condition.split("(") args =
args[:-1].split(",")
matched = False
for fact in facts:
fact_predicate, fact_args = fact.split("(")
fact_args = fact_args[:-1].split(",")
if predicate == fact_predicate and len(args) == len(fact_args):
```

```
temp_subs = {}
for var, val in zip(args, fact_args):
if var.islower():
if var in temp_subs and temp_subs[var] != val:
break
```

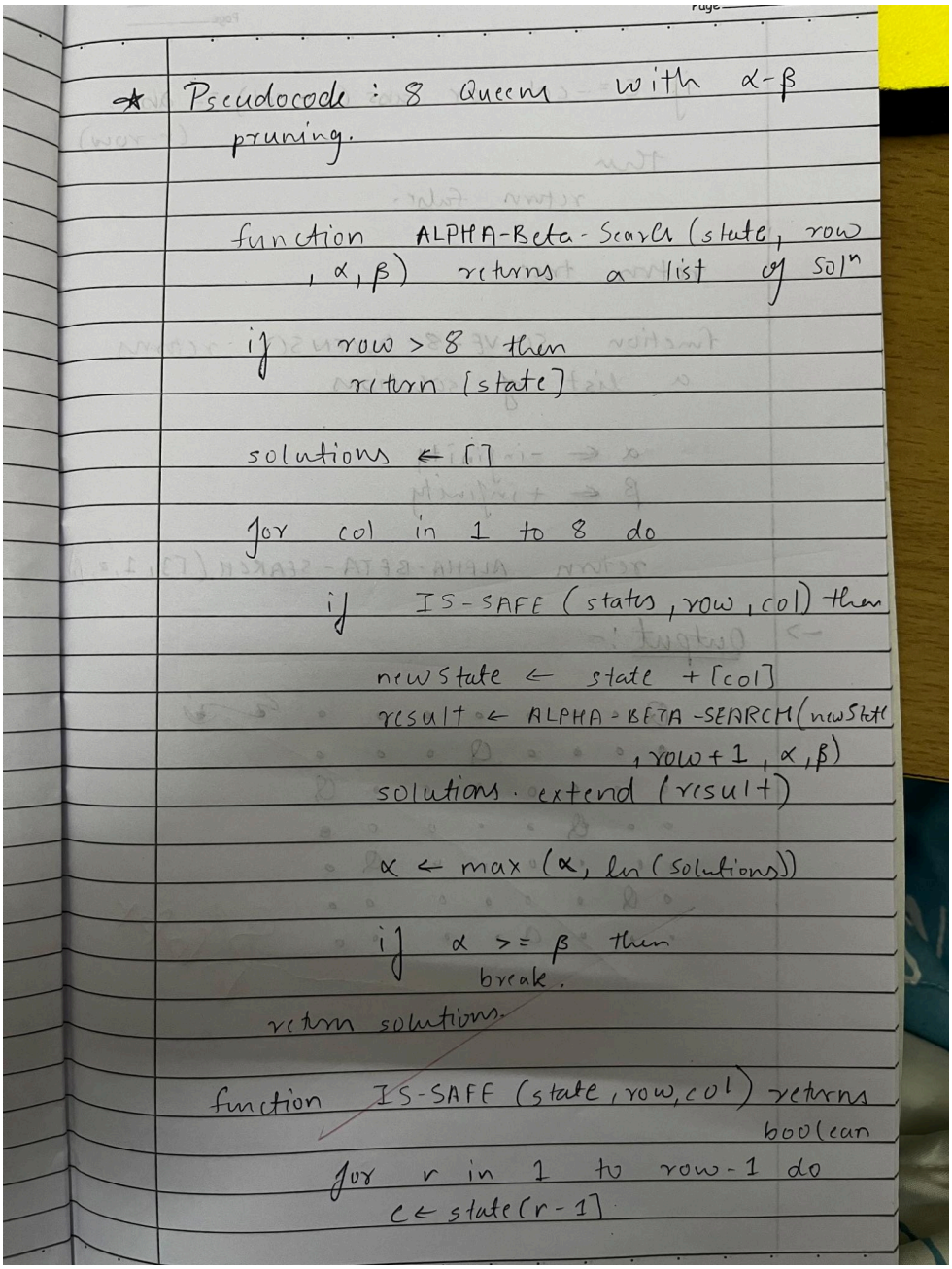
```
temp_subs[var] = val
elif var != val:
break
else:
matched = True
substitutions.update(temp_subs)
break
if not matched:
all_conditions_met = False
break
44if all_conditions_met:
for condition in then_conditions:
predicate, args = condition.split("(")
args = args[:-1].split(",")
new_fact = predicate + "(" + ",".join(substitutions.get(arg, arg) for arg in args)
+ ")"
new_inferences.add(new_fact)
if new_inferences - inferred:
inferred.update(new_inferences) facts.update({fact:
True for fact in new_inferences}) else:
break
return inferred
result = forward_chaining(knowledge_base)
print('Yashraj Sinha (1BM22CS335):')
if "Criminal(Robert)" in result:
print("Proved: Robert is a criminal.")
else:
print("Could not prove that Robert is a criminal.")
```

OUTPUT:

```
Yashraj Sinha (1BM22CS335):
Proved: Robert is a criminal.
```

Program 10 - Implement Alpha-Beta Pruning.

Algorithm



★ Pseudocode : 8 Queens with α-β pruning.

```
function ALPHA-Beta-Search (state, row, α, β) returns a list of soln
    if row > 8 then
        return [state]
    solutions ← []
    for col in 1 to 8 do
        if IS-SAFE (states, row, col) then
            newState ← state + [col]
            result ← ALPHA-BETA-SEARCH(newState, row+1, α, β)
            solutions.extend (result)
            α ← max (α, len (solutions))
            if α >= β then
                break.
    return solutions.

function IS-SAFE (state, row, col) returns boolean
    for r in 1 to row-1 do
        c ← state[r-1]
```

Date __/__/__
Page

if c == col or abs(c - col) == abs(r - row)

then

return false.

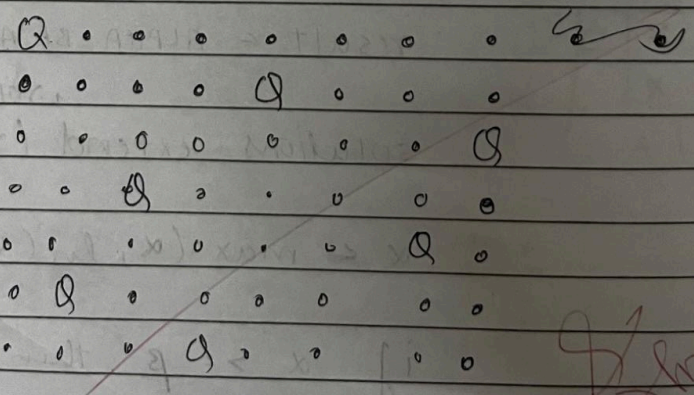
return true.

function SOLVE-8-QUEENS() returns a list of solutions.

α ← -infinity
β ← +infinity

return ALPHA-BETA-SEARCH([], 1, α, β)

-> Output :-



Code

```
import math
def alpha_beta_pruning(depth, node_index, is_maximizing_player, values, alpha, beta,
max_depth):
# Base case: when the maximum depth is
reached if depth == max_depth:
return values[node_index]
if is_maximizing_player:
best = -math.inf
# Recur for left and right children
for i in range(2):
val = alpha_beta_pruning(depth + 1, node_index * 2 + i, False, values, alpha, beta,
max_depth)
best = max(best, val)
alpha = max(alpha, best)
# Prune the remaining nodes
if beta <= alpha:
break
return best
else:
best = math.inf
# Recur for left and right children
for i in range(2):
val = alpha_beta_pruning(depth + 1, node_index * 2 + i, True, values, alpha, beta,
max_depth)
best = min(best, val)
beta = min(beta, best)
49# Prune the remaining nodes
if beta <= alpha:
break
return best
print("Yashraj Sinha (1BM22CS335):")
# Example usage
if _name____== "_main_":
# Example tree represented as a list of leaf node values
values = [3, 5, 6, 9, 1, 2, 0, -1]
max_depth = 3 # Height of the tree
result = alpha_beta_pruning(0, 0, True, values, -math.inf, math.inf, max_depth)
print("The optimal value is:", result)
```

OUTPUT:

Yashraj Sinha (1BM22CS335):
The optimal value is: 5