VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Aarusha GP (1BM23CS005)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Aug-2025 to Dec-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Aarusha GP (1BM23CS005),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

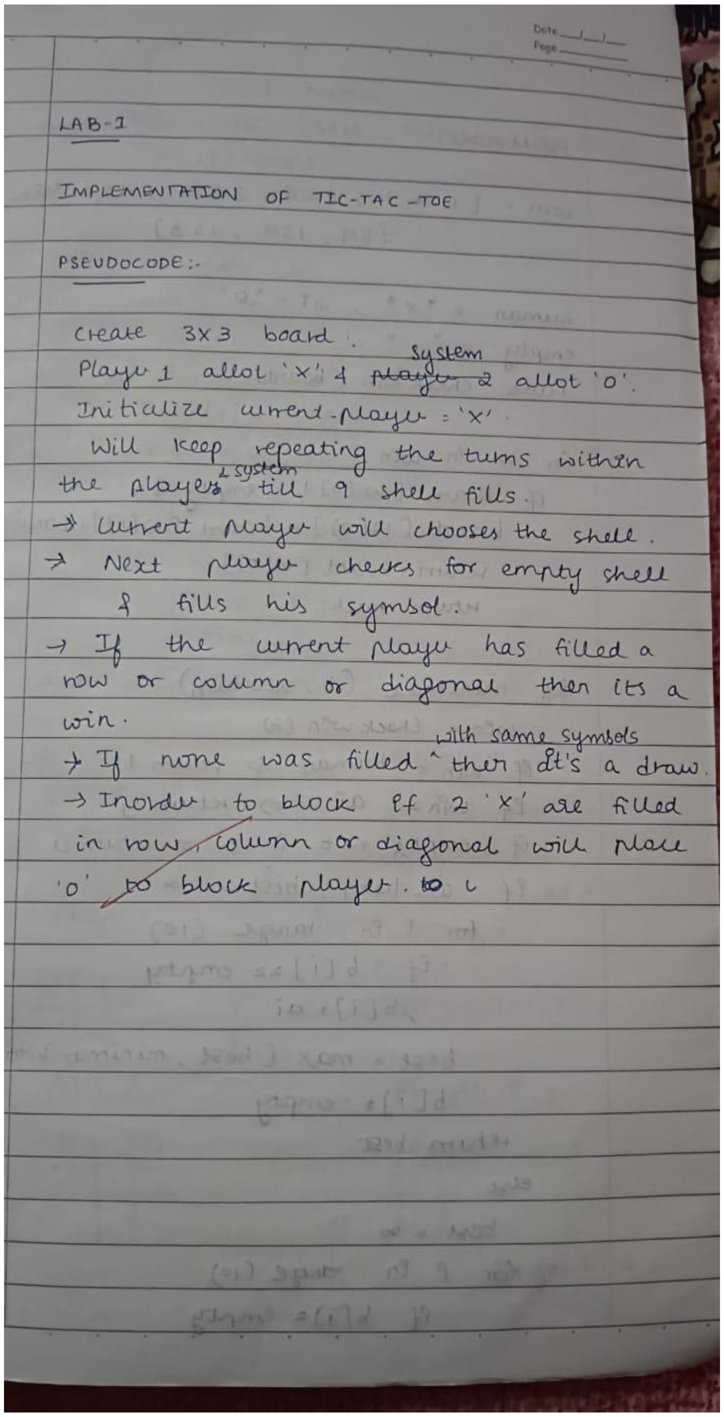
Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/aarushagp/AI-LAB-

Program 1
Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:



LAB-1

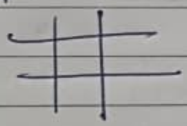
IMPLEMENTATION OF TIC-TAC-TOE

PSEUDOCODE:-

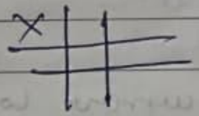
```
Create 3x3 board.
Player 1 allot 'x' & ~~player~~ system 2 allot 'o'.
Initialize current-player = 'x'.
 Will keep repeating the turns within
the players & system till 9 shell fills.
→ current player will chooses the shell.
→ Next player checks for empty shell
    & fills his symsol.
→ If the current player has filled a
row or column or diagonal then its a
win.
→ If none was filled with same symbols then it's a draw.
→ Inorder to block if 2 'x' are filled
in row, column or diagonal will place
'o' ~~to~~ block player. ~~to~~
```

```
        b[i]= human
    best = min [best, minimax (b, false)
      b[i] = empty
        return best

def ai_move (b)
    score = -∞
    move = -1
  for i in range (10)
    if b[i] = empty
      b[i] = a
      best = minimax (b, false)
      b[i] = empty
    if best > score
      score = best
      move = i
      return max.
```

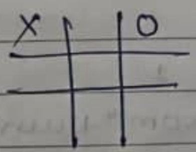
o/p:-



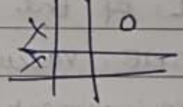
Player 1



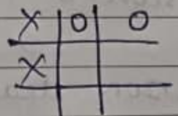
AI = 3



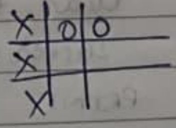
Player = 4



AI = 2



Player 7



Player wins.

Pseudocode :-

```
win = [1 4 7 , 2 5 8 , 3 6 9 , 1 2 3 , 4 5 6 ,
       7 8 9 , 1 5 9 , 7 5 3]

human = "x" , ai = 'o'
empty = " "
func checkWin (board)

for w in win
   if board [w[0]] != empty & &
      board [w[0]] = board [w[1]] = board[w[2]])
        return board [w[0]]
        return null

def minimax (b, ai_turn)
    win = Check win (b)
   if win = human => return 1
    if win = ai => return 1
   if empty not in b => return 0
    if ai_turn , best = -∞
       for i in range (10)
           if b [i] == empty
           b[i] = ai
          best = max (best , minimax(b, false))
          b[i] = empty
    return best
  else
     best = ∞
    for i in range (10)
          if b[i] = empty
```

```
FUNCTION move():
    rooms_list = [A, B, C, D]
    current_index = index of agent's current
location in rooms_list
    next_index = (current_index + 1) MOD length
                                    of rooms_list
  agent's location = rooms_list [next_index]
    PRINT " Moving to room " + agent's location

FUNCTION run(steps):
    FOR step FROM 1 TO steps DO
      PRINT " step " + step
    DISPLAY environment state (agent location &
             rooms status)
      CALL clear()
      CALL move()
      WAIT for 1 second (optional)

PRINT final environment state
PRINT total number of cleared rooms
                                    (score)

MAIN:
    Create environment
    Create VaccumAgent linked to Environment
      CALL run() with desired number of
    steps (e.g., 8)

END.
```

```
O/P:- Room 0,0 was dirty, now cleared
Room 0,1 already clear
Room 1,1 is cleared
Room 1,0 already cleared
All rooms cleared.
```

Code:
Tic tac toa game:-

```
import math

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


def check_winner(board, player):
    for row in board:
        if all(cell == player for cell in row):
            return True

    for col in range(3):
        if all(board[row][col] == player for row in range(3)):
            return True

    if all(board[i][i] == player for i in range(3)):
        return True
    if all(board[i][2 - i] == player for i in range(3)):
        return True

    return False


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


def get_human_move(board):
    while True:
        try:
            move = input("Enter your move (row and column: 1 1): ")
            row, col = map(int, move.split())
            row -= 1
            col -= 1

            if row not in range(3) or col not in range(3):
                print("Please enter values between 1 and 3.")
                continue

            if board[row][col] != ' ':
                print("That cell is already taken. Try again.")
                continue

            return row, col
        except ValueError:
            print("Invalid input. Enter row and column numbers separated by a space.")
```

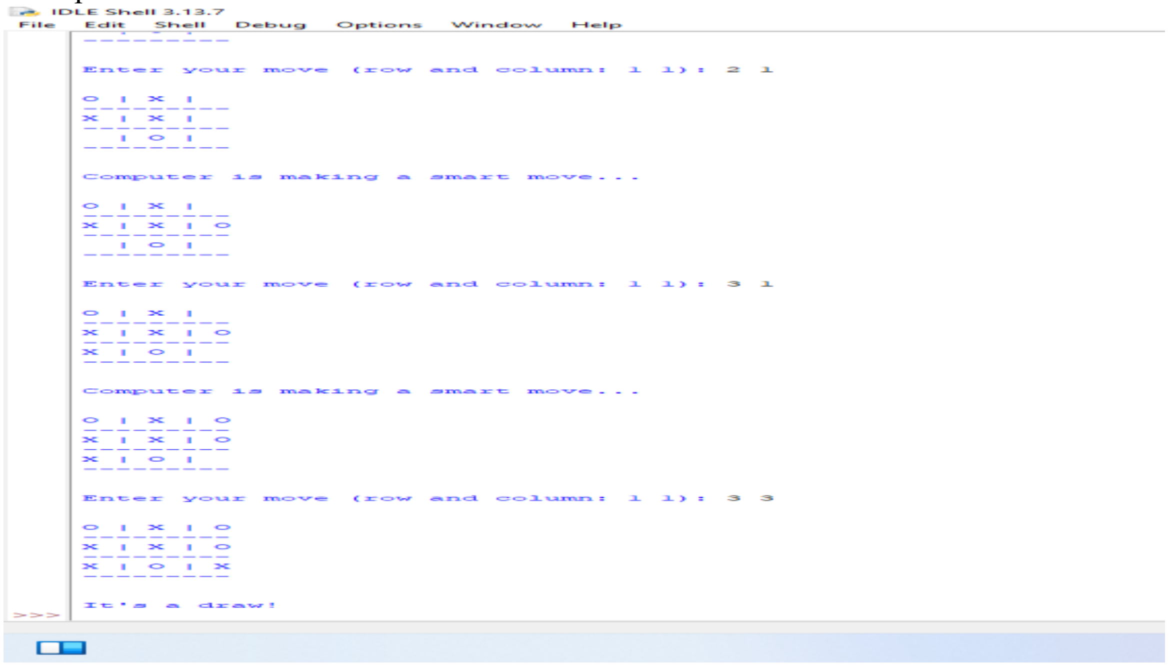
```
def minimax(board, depth, is_maximizing, human, computer): if
check_winner(board, computer):
            return 1
        elif check_winner(board, human):
            return -1
        elif is_full(board):
            return 0

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


    def get_best_move(board, human, computer):
        best_score = -math.inf
        move = None

        for  i  in  range(3):
            for j in range(3):
                if   board[i][j]    ==    '    ':
                    board[i][j] = computer
                    score = minimax(board, 0, False, human, computer)
                    board[i][j] = ' '
                    if score > best_score:
                        best_score  =  score
                        move = (i, j)
        return move


    def play_game():
```

```
    board = [[' ' for _ in range(3)] for _ in range(3)]
    human = 'X'
    computer = 'O'
    current_player = human  # Human starts first

    print("Welcome to Tic-Tac-Toe (You vs Unbeatable Computer)!")
    print_board(board)

    while True:
        if current_player == human:
            row, col = get_human_move(board)
        else:
            print("Computer is making a smart move...")
            row, col = get_best_move(board, human, computer)

        board[row][col] = current_player
        print_board(board)

        if check_winner(board, current_player):
            if current_player == human:
                print("🎉 You win!")
            else:
                print("☎ Computer wins!")
            break

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

        # Switch turns
        current_player = computer if current_player == human else human


if _name___== "_main_":
    play_game()
```

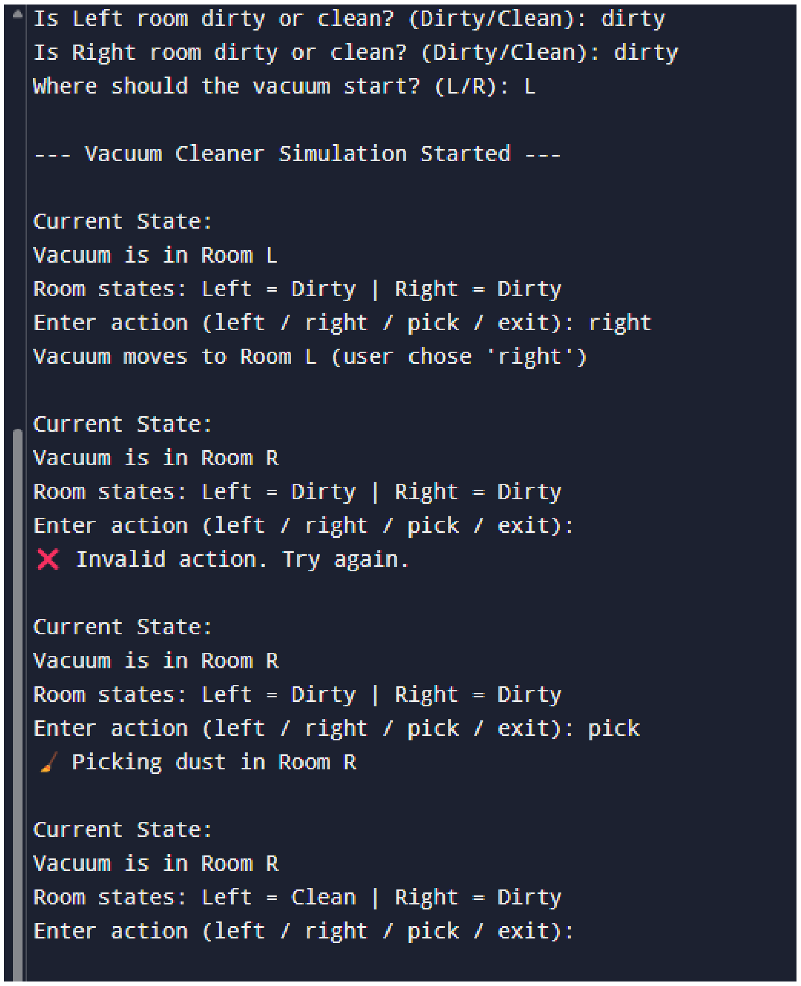
Vacuum cleaner agent:-

```
def display_state(vacuum_location, rooms):
    print("\nCurrent State:")
    print(f"Vacuum is in Room {vacuum_location}")
    print(f"Room states: Left = {rooms['R']} | Right = {rooms['L']}")


def vacuum_simulation():
    # Initial setup
    rooms = {
        'L': input("Is Left room dirty or clean? (Dirty/Clean): ").strip().capitalize(),
        'R': input("Is Right room dirty or clean? (Dirty/Clean): ").strip().capitalize()
    }

    vacuum_location = input("Where should the vacuum start? (L/R): ").strip().upper()
    if vacuum_location not in ['R', 'L']:
        print("Invalid input. Defaulting to 'L'")
        vacuum_location = 'L'

    print("\n--- Vacuum Cleaner Simulation Started ---")

    while True:
        display_state(vacuum_location, rooms)

        # Check if both rooms are clean
        if rooms['L'] == 'Clean' and rooms['R'] == 'Clean':
            print("✅ All rooms are clean! Job done.")
            break

        action = input("Enter action (left / right / pick / exit): ").strip().lower()

        if action == 'exit':
            print("Simulation ended by user.")
            break

        if action == 'pick':
            if rooms[vacuum_location] == 'Dirty':
                print(f"□ Picking dust in Room {vacuum_location}")
                rooms[vacuum_location] = 'Clean'
            else:
                print(f"Room {vacuum_location} is already clean.")
        elif action == 'left':
            vacuum_location = 'L'  # Moving to right if user says left
            print("Vacuum moves to Room R (user chose 'left')")
        elif action == 'right':
            vacuum_location = 'R'  # Moving to left if user says right
            print("Vacuum moves to Room L (user chose 'right')")
        else:
            print("❌ Invalid action. Try again.")

    print("Simulation complete.")


if _name___== "_main_":
    vacuum_simulation()
```

Output:-



```
-------------

Enter your move (row and column: 1 1): 2 1

O | X |
-------------
X | X |
-------------
  | O |
-------------

Computer is making a smart move...

O | X |
-------------
X | X | O
-------------
  | O |
-------------

Enter your move (row and column: 1 1): 3 1

O | X |
-------------
X | X | O
-------------
X | O |
-------------

Computer is making a smart move...

O | X | O
-------------
X | X | O
-------------
X | O |
-------------

Enter your move (row and column: 1 1): 3 3

O | X | O
-------------
X | X | O
-------------
X | O | X
-------------

It's a draw!
>>>
```

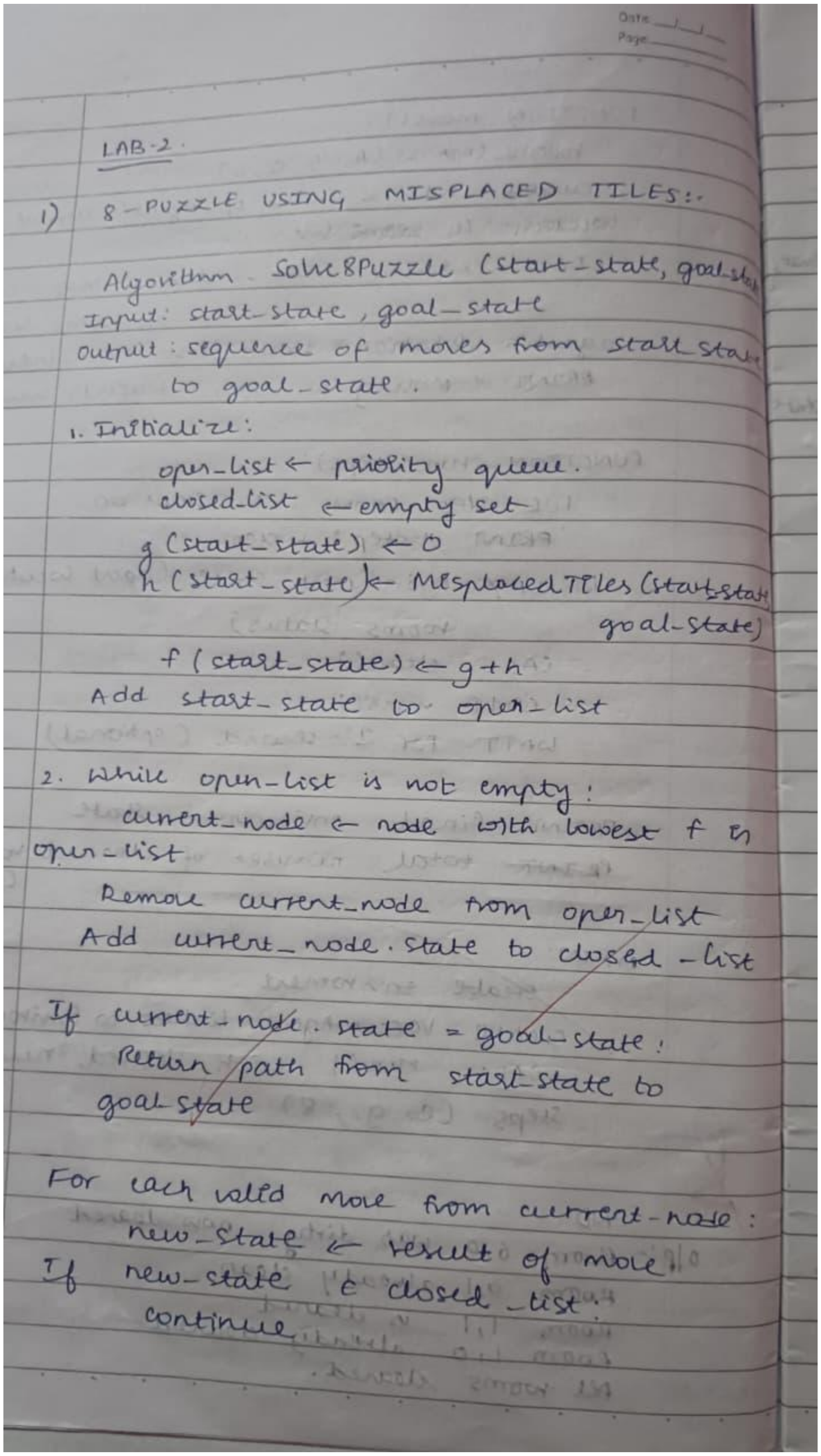


```
Is Left room dirty or clean? (Dirty/Clean): dirty
Is Right room dirty or clean? (Dirty/Clean): dirty
Where should the vacuum start? (L/R): L

--- Vacuum Cleaner Simulation Started ---

Current State:
Vacuum is in Room L
Room states: Left = Dirty | Right = Dirty
Enter action (left / right / pick / exit): right
Vacuum moves to Room L (user chose 'right')

Current State:
Vacuum is in Room R
Room states: Left = Dirty | Right = Dirty
Enter action (left / right / pick / exit):
❌ Invalid action. Try again.

Current State:
Vacuum is in Room R
Room states: Left = Dirty | Right = Dirty
Enter action (left / right / pick / exit): pick
🧹 Picking dust in Room R

Current State:
Vacuum is in Room R
Room states: Left = Clean | Right = Dirty
Enter action (left / right / pick / exit):
```

Program 2
Solve 8 puzzle problems

Implement Iterative deepening search algorithm
Implement Depth first search algorithm

Algorithm:



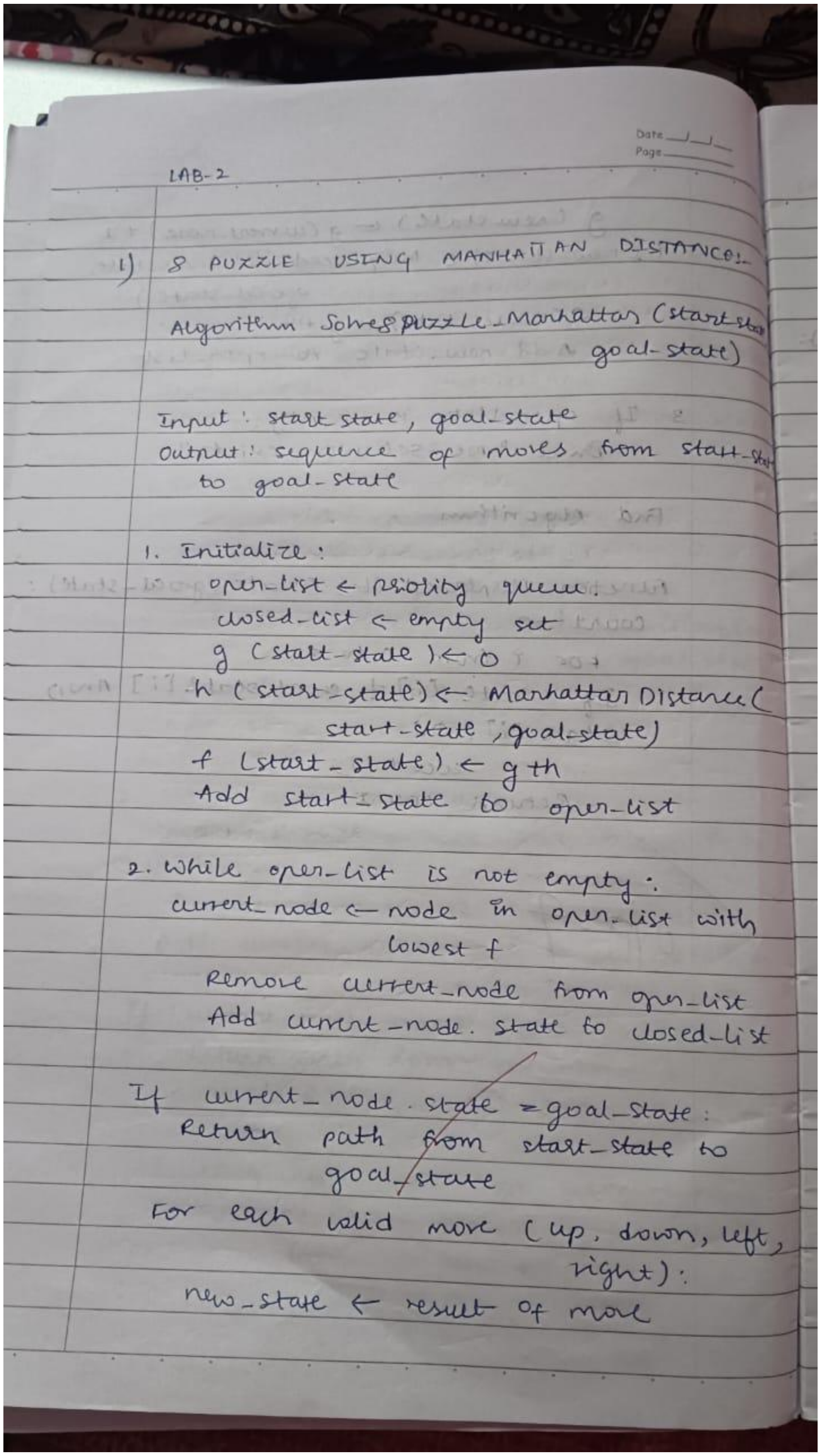
Date __/__/__
Page ____

LAB-2.

1) 8-PUZZLE USING MISPLACED TILES:-

```
Algorithm Solve8Puzzle (start-state, goal-state)
Input: start-state, goal-state
Output: sequence of moves from start state
        to goal-state.
1. Initialize:
        open-list ← priority queue.
        closed-list ← empty set
        g (start-state) ← 0
        h (start-state) ← MisplacedTiles (start-state,
                                          goal-state)
        f (start-state) ← g+h
    Add start-state to open-list

2. While open-list is not empty:
        current-node ← node with lowest f in
    open-list
      Remove current-node from open_list
      Add current_node.state to closed-list

   If current-node.state = goal-state:
      Return path from start-state to
      goal state

   For each valid move from current-node:
        new-state ← result of move.
   If new-state ∈ closed-list:
        continue.
        g (new-state) ← g (current-node) + 1
        h(new-state) ← MisplacedTiles (new-state,
                                        goal-state)
        f (new_state) ← g + h
        Add new_state to open_list

3. If open-list is empty:
     Print "NO solution exists."

End Algorithm

Function MisplacedTiles (state, goal-state):
  count ← 0
    For i from 0 to 8:
    If state [i] ≠ goal-state [i] AND
    state [i] ≠ 0:
            count ++;
      Return count
```

O/P - 9

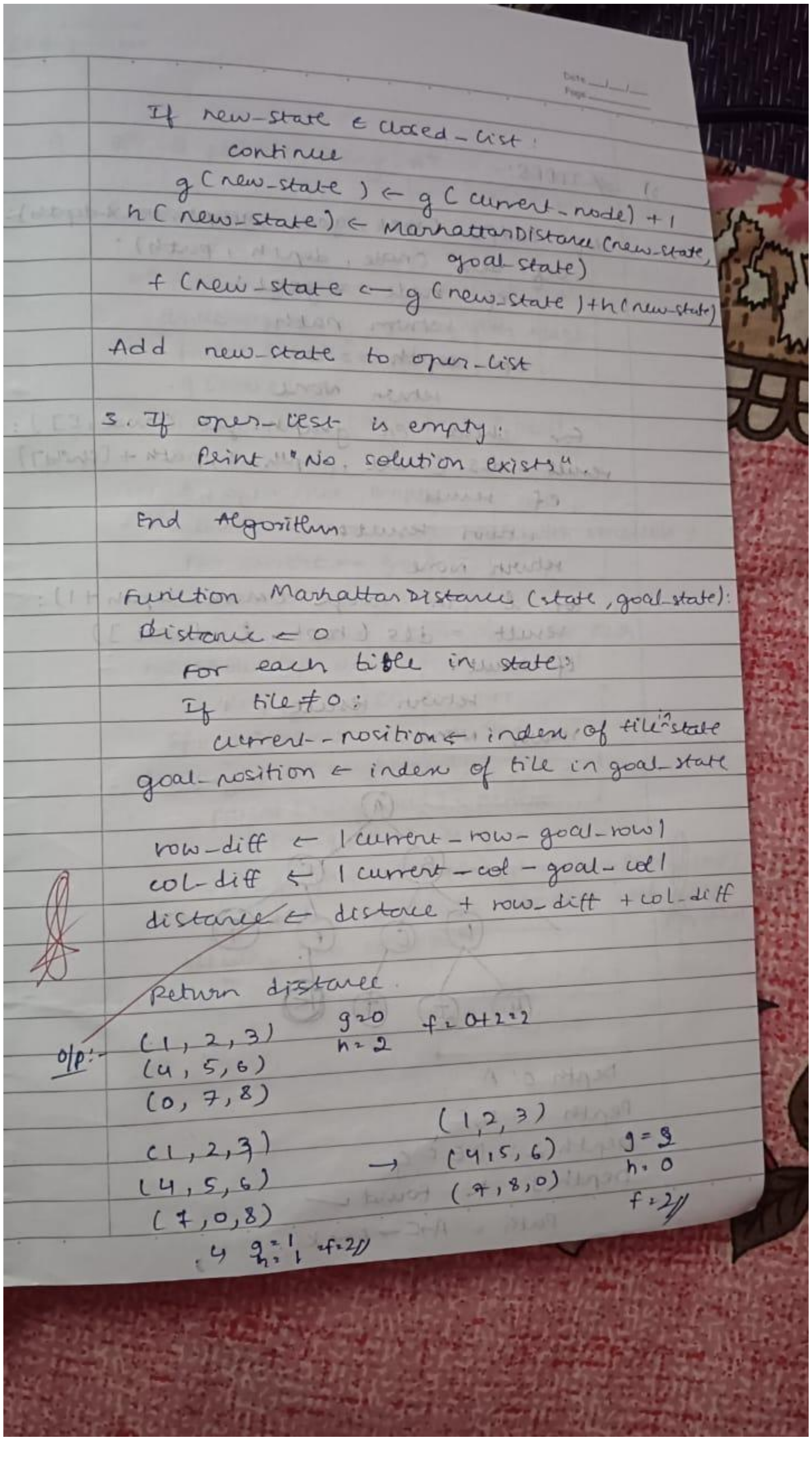


Date __/__/__
Page ____

LAB-2

1) 8 PUZZLE USING MANHATTAN DISTANCE:-

```
Algorithm SolvePuzzle_Manhattan (start_sta
                                 goal_state)

Input : start state, goal_state
Output : sequence of moves from start_sta
   to goal_state

1. Initialize :
     open_list ← priority queue.
     closed_list ← empty set
     g (start_state) ← 0
     h ( start_state) ← Manhattan Distance(
                start_state ;goal_state)
     f (start_state) ← g+h
     Add start_state to open_list

2. While open_list is not empty :
     current_node ← node in open_list with
                lowest f
     Remove current_node from open_list
     Add current_node . state to closed_list

   If current_node . state = goal_state :
     Return path from start_state to
              goal_state
   For each valid move ( up, down, left,
                          right) :
     new_state ← result of move
```



```
If new-state ∈ closed-list:
    continue
  g(new-state) ← g(current-node) + 1
h(new-state) ← ManhattanDistance(new-state,
                goal-state)
 f(new-state ← g(new-state) + h(new-state)

Add new-state to open-list

5. If open-list is empty:
    Print "No. solution exists".

 End Algorithm

Function ManhattanDistance(state, goal-state):
 distance ← 0
   For each tile in state:
   If tile ≠ 0:
     current-position ← index of tile in state
 goal-position ← index of tile in goal-state

  row-diff ← |current-row - goal-row|
  col-diff ← |current-col - goal-col|
  distance ← distance + row-diff + col-diff

  Return distance.
```

O/P:-

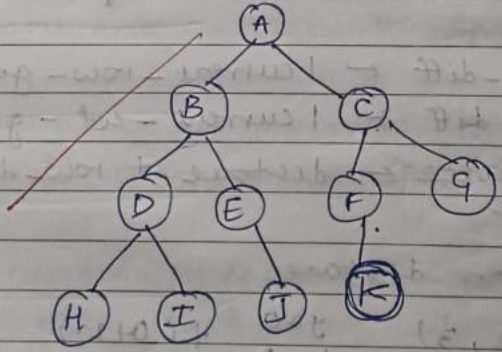
```
(1, 2, 3)     g=0   f = 0+2=2
(4, 5, 6)     h=2
(0, 7, 8)

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

[illegible]

Date __/__/__
Page ____

2) # IDDFS:-

```
def iddfs (root, goal, graph, max-depth):
    def dls (node, depth, path):
        if node == goal:
            return path
        if depth == 0:
            return None
    for child in graph.get (node, []):
    result = dls (child, depth - 1, path + [child])
        if result:
            return result
        return None
    for l in range (max-depth + 1):
        result = dls (root, l, [root])
        if result:
            return result
        return None.
```



Depth 0: A
Depth 1: B, C
Depth 2: D E F G
Depth 3: K found

Path = A→C→F→K !.

Code:

IDS:-

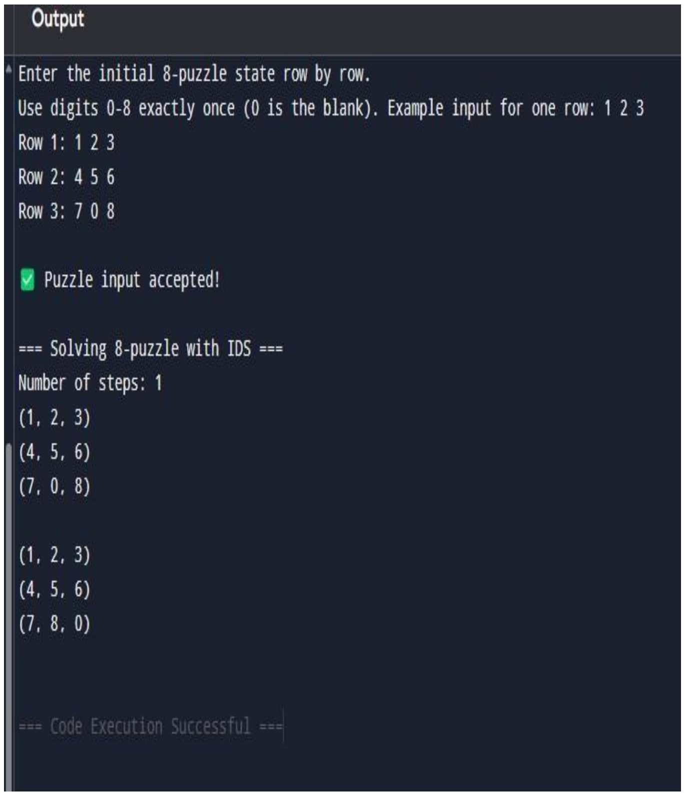
```
GOAL_STATE = (1, 2, 3, 4, 5, 6, 7, 8, 0)

MOVES = {
    'UP': -3,
    'DOWN': 3,
    'LEFT': -1,
    'RIGHT': 1
}

def is_valid_move(pos, move):
    if move == 'UP' and pos < 3:
        return False
    if move == 'DOWN' and pos > 5:
        return False
    if move == 'LEFT' and pos % 3 == 0:
        return False
    if move == 'RIGHT' and (pos + 1) % 3 == 0:
        return False
    return True

def move_tile(state, move):
    new_state = list(state)
    idx = new_state.index(0)
    if not is_valid_move(idx, move):
        return None
    swap_idx = idx + MOVES[move]
    new_state[idx], new_state[swap_idx] = new_state[swap_idx], new_state[idx]
    return tuple(new_state)

def get_successors(state):
    successors = []
    for move in MOVES:
        new_state = move_tile(state, move)
        if new_state:
            successors.append(new_state)
    return successors

def dls(state, depth):
    stack = [(state, [])]
    visited = set()

    while stack:
        curr_state, path = stack.pop()
        if curr_state in visited:
            continue
        visited.add(curr_state)

        if curr_state == GOAL_STATE:
            return path + [curr_state]

        if len(path) >= depth:
            continue

        for succ in get_successors(curr_state):
            stack.append((succ, path + [curr_state]))

    return None

def ids(start_state, max_depth=50):
    for depth in range(max_depth + 1):
        result = dls(start_state, depth)
        if result:
            return result
    return None

def print_path(path):
    print("Number of steps:", len(path) - 1)
    for state in path:
        for i in range(0, 9, 3):
            print(state[i:i+3])
        print()

def get_user_input():
    print("Enter the initial 8-puzzle state row by row.")
    print("Use digits 0-8 exactly once (0 is the blank). Example input for one row: 1 2 3")
    user_values = []

    while len(user_values) < 9:
        try:
            row_input = input(f"Row {len(user_values)//3 + 1}: ").strip()
            row = list(map(int, row_input.split()))
            if len(row) != 3:
                print("Please enter exactly 3 numbers.")
                continue
            user_values.extend(row)
        except ValueError:
            print("Invalid input. Please enter numbers only.")

    if sorted(user_values) != list(range(9)):
        print("The puzzle must contain all digits from 0 to 8 exactly once.\nLet's try again.")
        return get_user_input()

    print("\n✅ Puzzle input accepted!\n")
    return tuple(user_values)

if _name___== "_main_":
    start_state = get_user_input()
    print("=== Solving 8-puzzle with IDS ===")
    solution = ids(start_state)
    if solution:
        print_path(solution)
    else:
        print("No solution found within depth limit.")
```

DFS:-

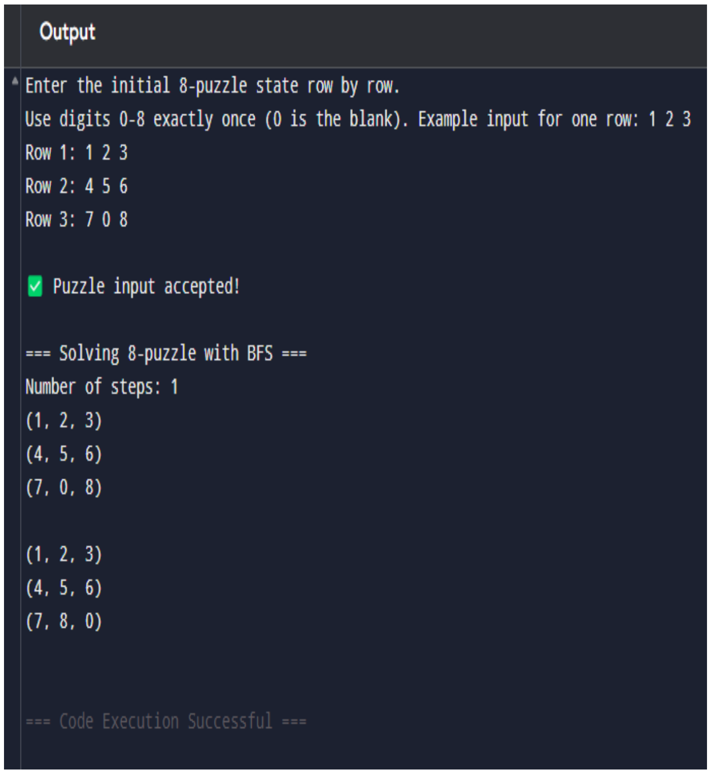
```
from collections import deque

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

MOVES = {
    'UP': -3,
    'DOWN': 3,
    'LEFT': -1,
    'RIGHT': 1
}

def is_valid_move(pos, move):
    if move == 'UP' and pos < 3:
        return False
    if move == 'DOWN' and pos > 5:
        return False
    if move == 'LEFT' and pos % 3 == 0:
        return False
    if move == 'RIGHT' and (pos + 1) % 3 == 0:
        return False
    return True

def move_tile(state, move):
    new_state = list(state)
    idx = new_state.index(0)
    if not is_valid_move(idx, move):
        return None
    swap_idx = idx + MOVES[move]
    new_state[idx], new_state[swap_idx] = new_state[swap_idx], new_state[idx]
    return tuple(new_state)
```

```
def get_successors(state):
    successors = []
    for move in MOVES:
        new_state = move_tile(state, move)
        if new_state:
            successors.append(new_state)
    return successors

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

    while queue:
        state, path = queue.popleft()
        if state in visited:
            continue
        visited.add(state)

        if state == GOAL_STATE:
            return path + [state]

        for succ in get_successors(state):
            queue.append((succ, path + [state]))

    return None

def print_path(path):
    print("Number of steps:", len(path) - 1)
    for state in path:
        for i in range(0, 9, 3):
            print(state[i:i+3])
        print()

def get_user_input():
    print("Enter the initial 8-puzzle state row by row.")
    print("Use digits 0-8 exactly once (0 is the blank). Example input for one row: 1 2 3")
    user_values = []

    while len(user_values) < 9:
        try:
            row_input = input(f"Row {len(user_values)//3 + 1}: ").strip()
            row = list(map(int, row_input.split()))
            if len(row) != 3:
                print(" Please enter exactly 3 numbers.")
                continue
            user_values.extend(row)
        except ValueError:
            print(" Invalid input. Please enter numbers only.")

    if sorted(user_values) != list(range(9)):
        print("The puzzle must contain all digits from 0 to 8 exactly once.")
        return get_user_input()

    print("\n Puzzle input accepted!\n")
    return tuple(user_values)


if _name___== "_main_":
    start_state = get_user_input()
    print("=== Solving 8-puzzle with BFS ===")
    result = bfs(start_state)
    if result:
        print_path(result)
    else:
        print("No solution found.")
```

Output:-



```
Output
Enter the initial 8-puzzle state row by row.
Use digits 0-8 exactly once (0 is the blank). Example input for one row: 1 2 3
Row 1: 1 2 3
Row 2: 4 5 6
Row 3: 7 0 8

✅ Puzzle input accepted!

=== Solving 8-puzzle with IDS ===
Number of steps: 1
(1, 2, 3)
(4, 5, 6)
(7, 0, 8)

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


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

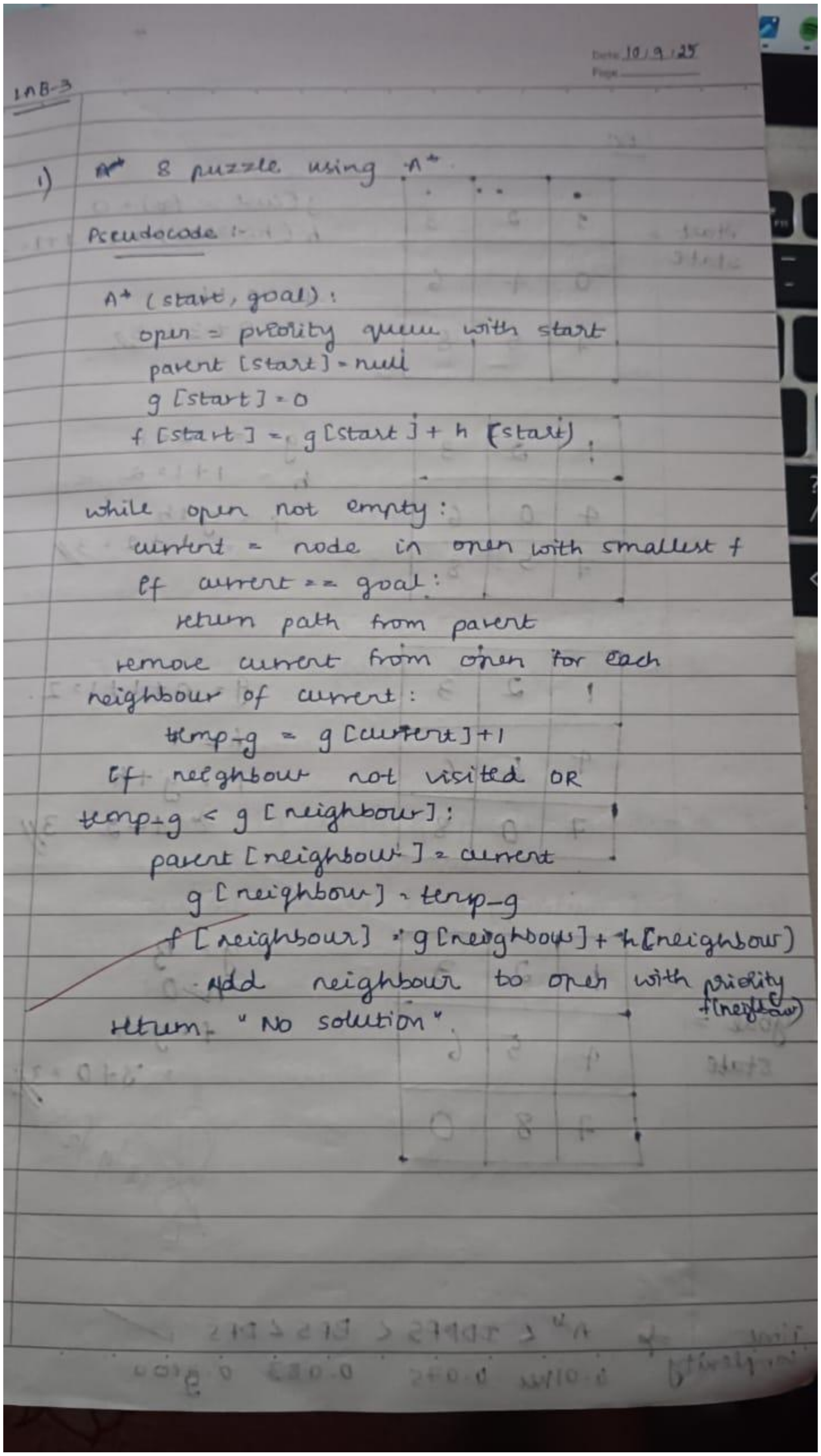


```
Output
Enter the initial 8-puzzle state row by row.
Use digits 0-8 exactly once (0 is the blank). Example input for one row: 1 2 3
Row 1: 1 2 3
Row 2: 4 5 6
Row 3: 7 0 8

✅ Puzzle input accepted!

=== Solving 8-puzzle with BFS ===
Number of steps: 1
(1, 2, 3)
(4, 5, 6)
(7, 0, 8)

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


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

Program 3
Solve 8 puzzle problems

Implement A* search algorithm

Algorithm:

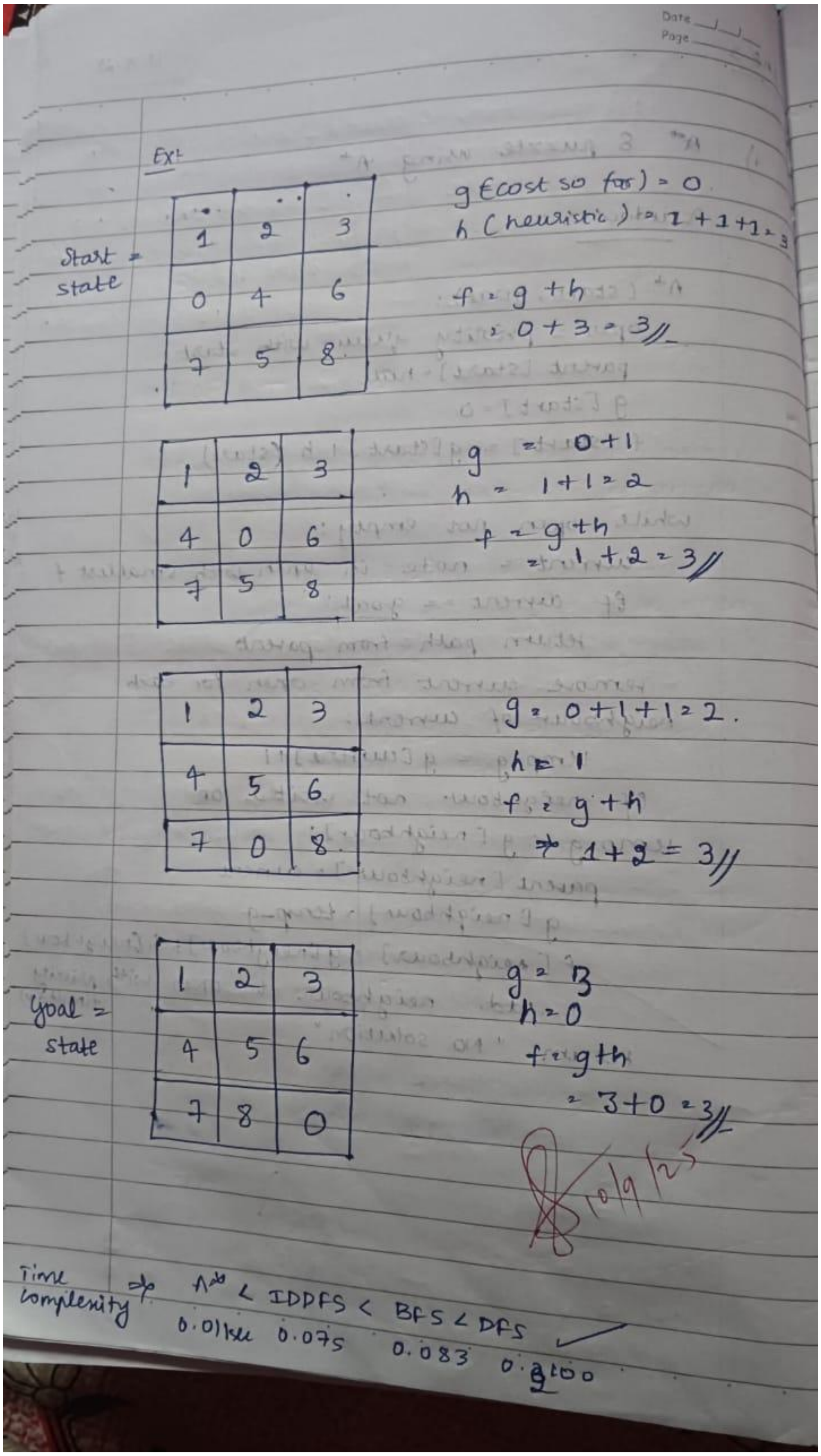

Date 10/9/25

LAB-3

1) ~~A*~~ 8 puzzle using A*.

Pseudocode :-

```
A* (start, goal) :
    open = priority queue with start
     parent [start] = null
     g [start] = 0
    f [start] = g [start] + h (start)

while open not empty :
    current = node in open with smallest f
    if current == goal :
        return path from parent
  remove current from open for each
neighbour of current :
        temp_g = g [current] + 1
   if neighbour not visited OR
temp_g < g [neighbour] :
     parent [neighbour] = current
       g [neighbour] = temp_g
      f [neighbour] = g [neighbour] + h [neighbour]
        Add neighbour to open with priority f(neighbour)
  return "No solution".
```



Ex:

Start state =

1	2	3
0	4	6
7	5	8

g (cost so far) = 0

h (heuristic) = 1 + 1 + 1 = 3

f = g + h

= 0 + 3 = 3

1	2	3
4	0	6
7	5	8

g = 0 + 1

h = 1 + 1 = 2

f = g + h

= 1 + 2 = 3

1	2	3
4	5	6
7	0	8

g = 0 + 1 + 1 = 2.

h = 1

f = g + h

⇒ 1 + 2 = 3

Goal state =

1	2	3
4	5	6
7	8	0

g = 3

h = 0

f = g + h

= 3 + 0 = 3

[illegible] 10/9/25

Time complexity ⇒ A* < IDDFS < BFS < DFS

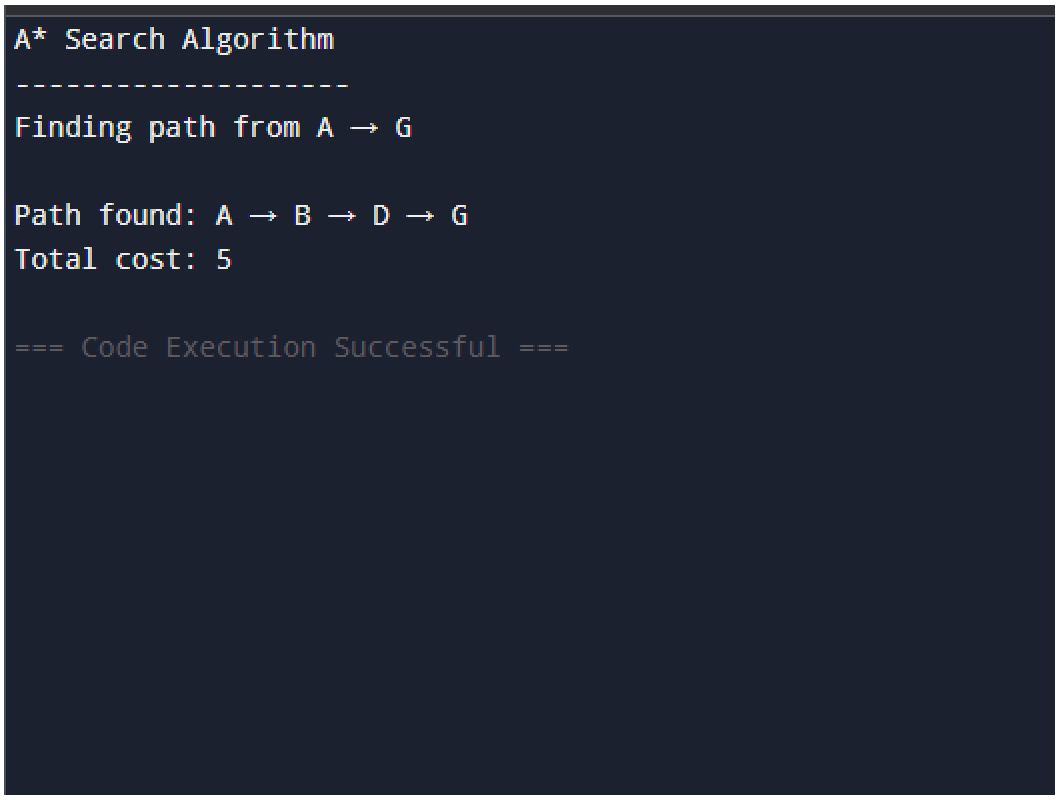
0.01sec 0.075 0.083 0.2100

Code:-

```
# A* Search Algorithm Implementation in Python

from queue import PriorityQueue

def a_star_search(graph, heuristic, start, goal):
    # Priority queue to store (f_score, node, path)
    pq = PriorityQueue()
    pq.put((0, start, [start]))

    # Dictionary to store g_scores (cost from start)
    g_score = {node: float('inf') for node in graph}
    g_score[start] = 0

    while not pq.empty():
        f, current, path = pq.get()

        # Goal check
        if current == goal:
            print("Path found:", ' → '.join(path))
            print("Total cost:", g_score[goal])
            return

        # Explore neighbors
        for neighbor, cost in graph[current]:
            tentative_g = g_score[current] + cost

            if tentative_g < g_score[neighbor]:
                g_score[neighbor] = tentative_g
                f_score = tentative_g + heuristic[neighbor]
                pq.put((f_score, neighbor, path + [neighbor]))

    print("No path found!")

# Example usage
if __name__ == "__main__":
    # Define the graph as adjacency list
    # Each node: [(neighbor, cost), ...]
    graph = {
        'A': [('B', 1), ('C', 3)],
        'B': [('D', 1), ('E', 5)],
        'C': [('F', 2)],
        'D': [('G', 3)],
        'E': [('G', 1)],
'F': [('G', 5)],
'G': []
```

```
}

# Define heuristic values (estimated cost to reach goal)
heuristic = {
    'A': 7,
    'B': 6,
    'C': 5,
    'D': 4,
    'E': 2,
    'F': 1,
    'G': 0
}

start_node = 'A'
goal_node = 'G'

print("A* Search Algorithm")
print("----------------------")
print(f"Finding path from {start_node} → {goal_node}\n")

a_star_search(graph, heuristic, start_node, goal_node)
```

Output:-



```
A* Search Algorithm
--------------------
Finding path from A → G

Path found: A → B → D → G
Total cost: 5

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

<u>Program 4</u>

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:-

LAB-4

Hill climbing for N-queens :-

$$\Delta E = e^{\frac{\Delta E}{KT}}$$

	Q1		
			Q2
Q3			
		Q4	

Algorithm :-

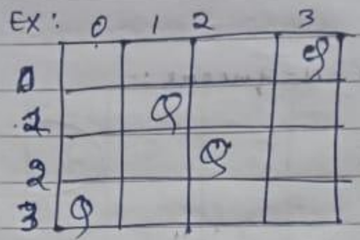
1) Start with random board.
2) Compute h = number of $\overset{attacking}{\sim}$ pairs.

Repeat

a) Generate all neighbouring boards.
b) Choose neighbour with minimum h.
c)

```
function HILL CLIMBING (problem) returns a St
    that is local maximum
        current ← Make_node (problem. Initial.state
    loop do
        neighbour ← a highest valued successor
                        of current
        if neighbour.value ≤ current.value
    then return current.state
                current ← neighbour.
```

Ex:



Col :	0	1	2	3
row :	3	1	2	0

For Q(0,3) vs Q(1,1)
rows ≠ (3−1) = 2 , |0−1| = 1 no attacks.

Q (0, 3) vs Q (2,2)
rows = (0−2) = 2 , |3 −2| = 1 ≠ no attack

Q (0, 3) vs (3,0)
= (3−0) = 3 , (0−3) = 3 = equal
diagonal attack.

(1,1) vs (2,2)
equal yes diagonal attack

(1,1) & (3,0): no attack
(2,2) & (3,0) : no attack

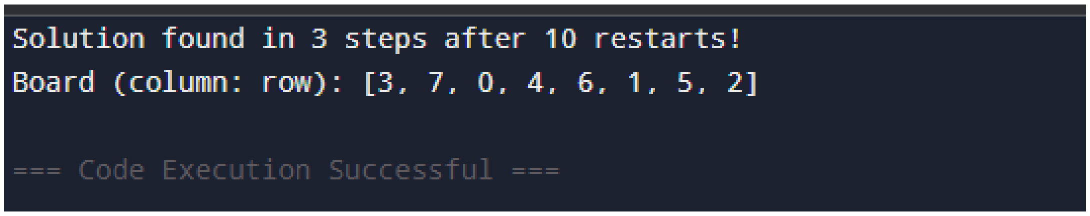
There are 2 attacks = h = 2.
Therefore [3, 1, 2, 0] not a goal state.

3, 0, 2]

Code:-

```
import random

def generate_board(N):
    """Generate a random board: board[i] = row of queen in column i"""
    return [random.randint(0, N-1) for _ in range(N)]

def compute_heuristic(board):
    """Compute number of pairs of queens attacking each other"""
    h = 0
    N = len(board)
    for i in range(N):
        for j in range(i+1, N):
            if board[i] == board[j]:            # same row
                h += 1
            elif abs(board[i] - board[j]) == j - i:  # same diagonal
                h += 1
    return h

def get_neighbors(board):
    """Generate all neighbors by moving one queen in its column"""
    neighbors = []
    N = len(board)
    for col in range(N):
        for row in range(N):
            if board[col] != row:
                new_board = list(board)
                new_board[col] = row
                neighbors.append(new_board)
    return neighbors

def hill_climbing(N, max_restarts=100):
    for restart in range(max_restarts):
        board = generate_board(N)
        steps = 0
        while True:
            h = compute_heuristic(board)
            if h == 0:
                print(f"Solution found in {steps} steps after {restart} restarts!")
                return board
        neighbors = get_neighbors(board)
        h_values = [compute_heuristic(nb) for nb in neighbors]
        min_h = min(h_values)
        if min_h >= h:  # no improvement
            break       # local maxima, do random restart
        # move to the neighbor with minimum heuristic
        board = neighbors[h_values.index(min_h)]
        steps += 1
    print("No solution found")
    return None

# Example usage
N = 8
solution = hill_climbing(N)
if solution:

    print("Board (column: row):", solution)
```

Output:-

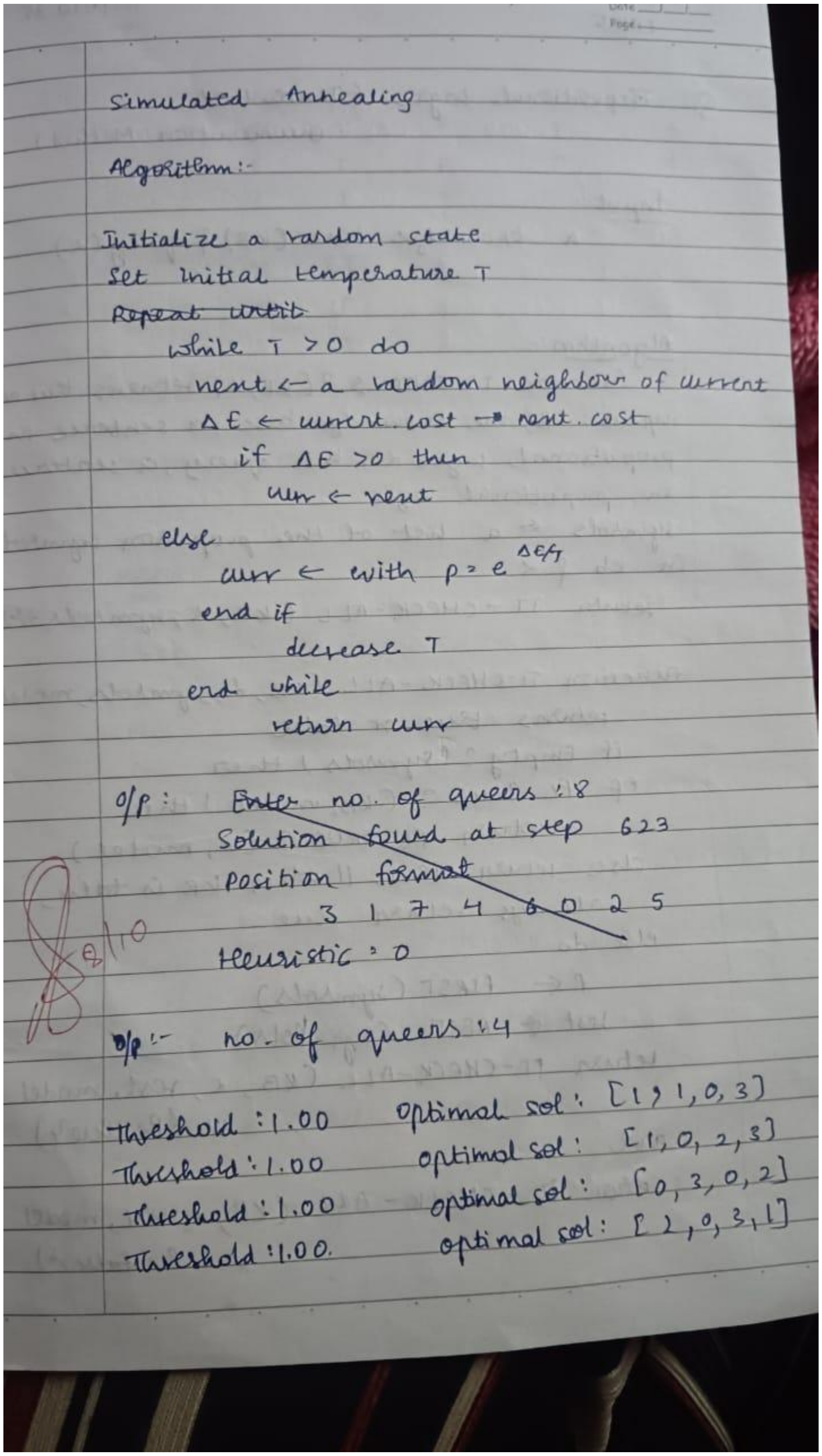


```
Solution found in 3 steps after 10 restarts!
Board (column: row): [3, 7, 0, 4, 6, 1, 5, 2]


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

Program 5

Simulated Annealing

Algorithm



Simulated Annealing

Algorithm:-

```
Initialize a random state
Set initial temperature T
~~Repeat until~~
    while T > 0 do
        next <- a random neighbour of current
        ΔE <- current.cost - next.cost
            if ΔE > 0 then
                curr <- next
    else
        curr <- with $p = e^{\Delta E/T}$
    end if
            decrease T
    end while
            return curr
```

```
O/P :   Enter no. of queens : 8
        Solution found at step 623
        Position format
                3  1  7  4  6  0  2  5
        Heuristic : 0
```

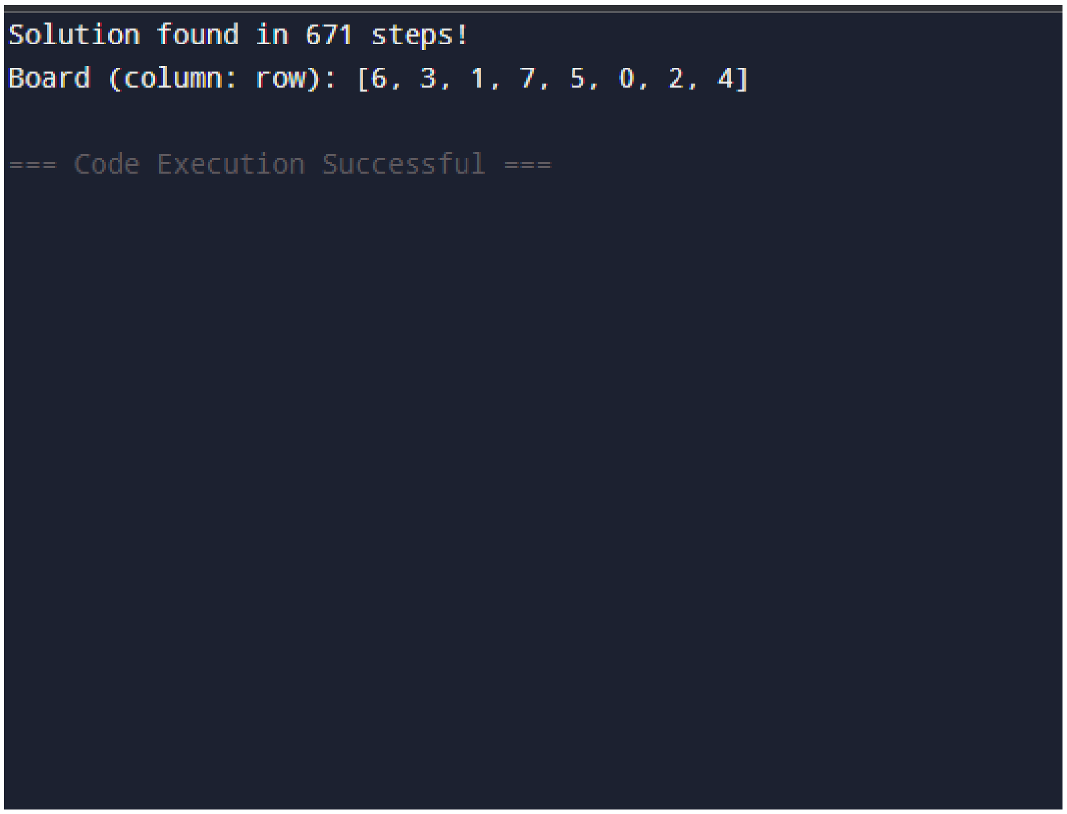
8/10

```
O/P :-   no. of queens : 4

Threshold : 1.00      optimal sol : [1, 1, 0, 3]
Threshold : 1.00      optimal sol : [1, 0, 2, 3]
Threshold : 1.00      optimal sol : [0, 3, 0, 2]
Threshold : 1.00.     optimal sol : [2, 0, 3, 1]
```

Code:-

```
import random
import math

def generate_board(N):
    """Generate a random board: board[i] = row of queen in column i"""
    return [random.randint(0, N-1) for _ in range(N)]

def compute_heuristic(board):
    """Compute number of pairs of queens attacking each other"""
    h = 0
    N = len(board)
    for i in range(N):
        for j in range(i+1, N):
            if board[i] == board[j]:             # same row
                h += 1
            elif abs(board[i] - board[j]) == j - i:  # same diagonal
                h += 1
    return h

def get_random_neighbor(board):
    """Generate a neighbor by moving one queen in its column to a random row"""
    N = len(board)
    col = random.randint(0, N-1)
    row = random.randint(0, N-1)
    while board[col] == row:
        row = random.randint(0, N-1)
    new_board = list(board)
    new_board[col] = row
    return new_board

def simulated_annealing(N, max_steps=100000, initial_temp=1000, cooling_rate=0.99):
    board = generate_board(N)
    current_h = compute_heuristic(board)
    T = initial_temp

    for step in range(max_steps):
        if current_h == 0:
            print(f"Solution found in {step} steps!")
            return board
        neighbor = get_random_neighbor(board)
        neighbor_h = compute_heuristic(neighbor)
        delta_h = neighbor_h - current_h

        if delta_h < 0:
            # Better neighbor, move to it
            board = neighbor
            current_h = neighbor_h
        else:
            # Worse neighbor, move with probability e^(-ΔH/T)
            probability = math.exp(-delta_h / T)
            if random.random() < probability:
                board = neighbor
                current_h = neighbor_h
        # Cool down the temperature
        T *= cooling_rate

    print("No solution found")
    return None

# Example usage
N = 8
solution = simulated_annealing(N)
if solution:
    print("Board (column: row):", solution)
```

Output:-



```
Solution found in 671 steps!
Board (column: row): [6, 3, 1, 7, 5, 0, 2, 4]

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

<u>Program 6</u>

Propositional Logic

Algorithm:-

LAB-5 Date 15/10/25

Page

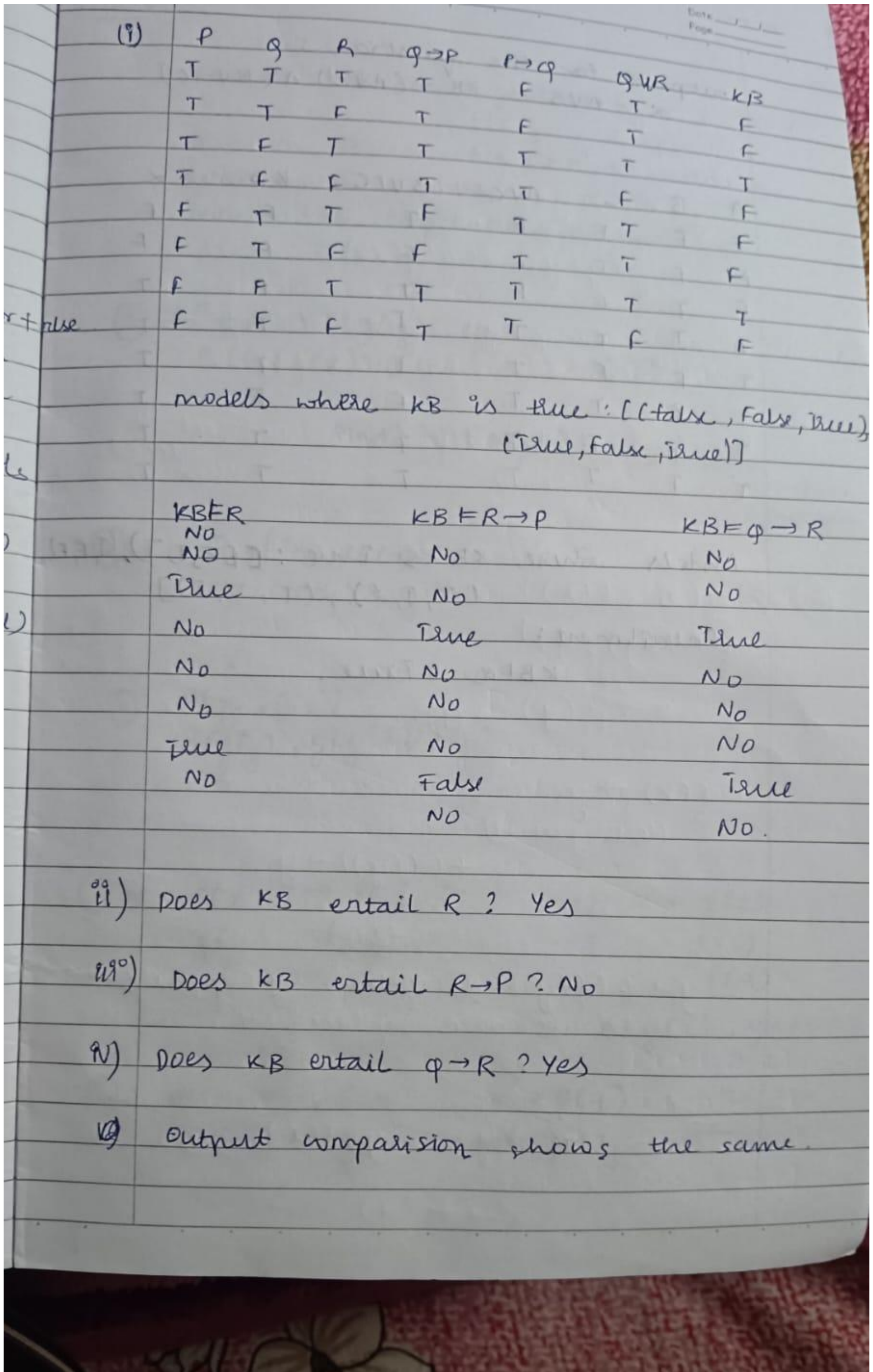
Q. Propositional logic (Truth table enumeration Method)

Input:-

a knowledge base (KB) & query (α).

Algorithm:-

```
function TT-ENTAILS ? (KB, $\alpha$) returns true or false
inputs: KB, the knowledge base, a sentence in
propotional logic $\alpha$, the query, a sentence
in propotional logic
symbols ← a list of the propotion symbols
in kb & $\alpha$.
  return TT-CHECK-ALL (KB, $\alpha$, symbols, {})

function TT-CHECK-ALL (KB, $\alpha$, symbols, model)
    returns true or false
    if Empty ? (symbols) then
   if PL-True ? (KB, model) then
       return PL-TRUE ? ($\alpha$, model)
   else return true // when KB is false,
       always return true
   else do
       P ← FIRST (symbols)
       rest ← REST (symbols)
     return TT-CHECK-ALL (KB, $\alpha$, rest, model
                               ∪ {P = true})
   and
     return TT-CHECK-ALL (KB, $\alpha$, rest, model
                               ∪ {P = false})
```



(i)

P	Q	R	Q→P	P→Q	QvR	KB
T	T	T	T	F	T	F
T	T	F	T	F	T	F
T	F	T	T	T	T	T
T	F	F	T	T	F	F
F	T	T	F	T	T	F
F	T	F	F	T	T	F
F	F	T	T	T	T	T
F	F	F	T	T	F	F

models where KB is true : [(false, False, True), (True, False, True)]

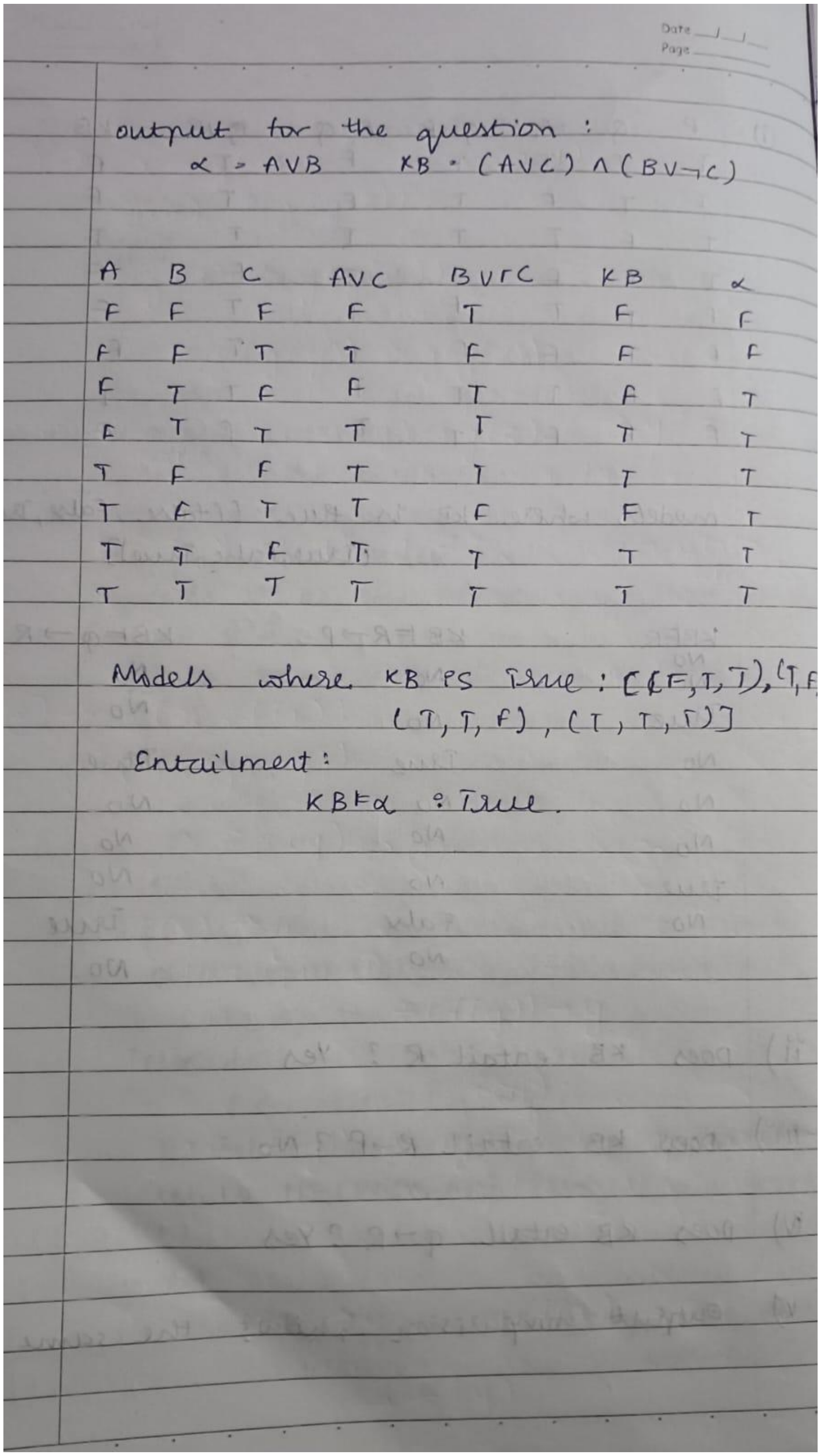
KB⊨R	KB⊨R→P	KB⊨Q→R
NO		
NO	No	No
True	NO	No
No	True	True
No	NO	No
No	No	No
True	NO	NO
NO	False	True
	NO	No.

ii) Does KB entail R ? Yes

iii) Does KB entail R→P ? No

iv) Does KB entail Q→R ? Yes

v) Output comparision shows the same.



output for the question :

$\alpha = A \lor B$ KB $= (A \lor C) \land (B \lor \neg C)$

A	B	C	$A \lor C$	$B \lor \neg C$	KB	α
F	F	F	F	T	F	F
F	F	T	T	F	F	F
F	T	F	F	T	F	T
F	T	T	T	T	T	T
T	F	F	T	T	T	T
T	F	T	T	F	F	T
T	T	F	T	T	T	T
T	T	T	T	T	T	T

Models where KB is True : [(F,T,T), (T,F

(T, T, F) , (T , T, T)]

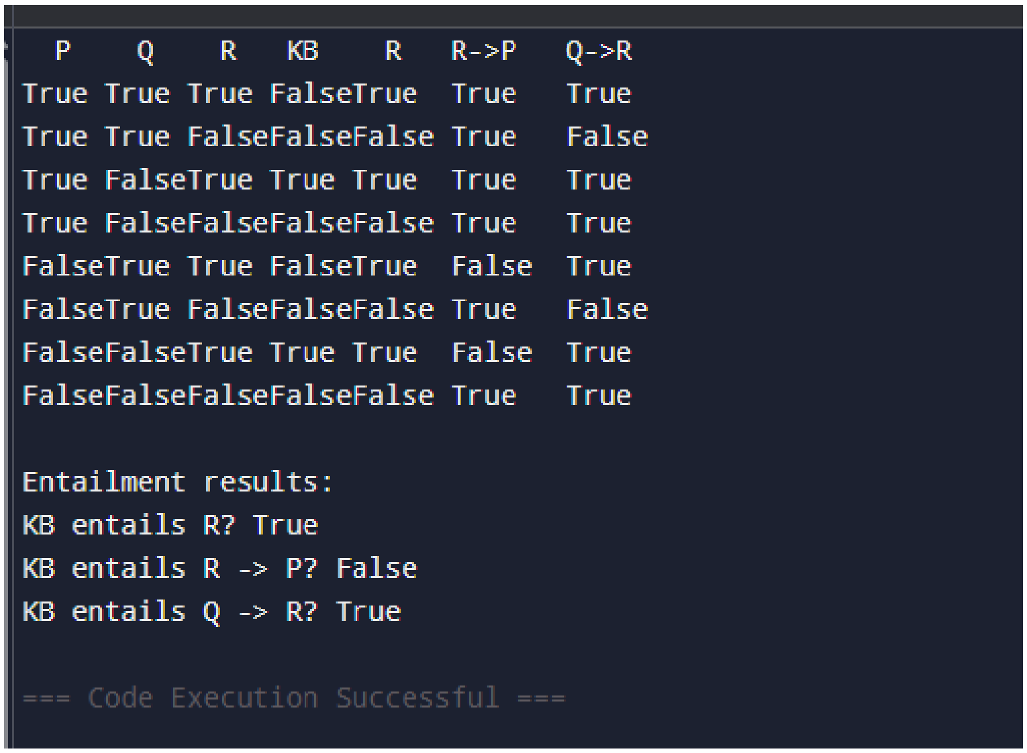
Entailment :

KB ⊨ α : True.

Code:-

```
from itertools import product

# Define the propositional variables
variables = ['P', 'Q', 'R']

# Define all possible truth assignments
assignments = list(product([True, False], repeat=len(variables)))

# Function to evaluate the KB sentences
def evaluate_KB(P, Q, R):
    s1 = (not Q) or P       # Q -> P
    s2 = (not P) or (not Q) # P -> ¬Q
    s3 = Q or R             # Q ∨ R
    KB_true = s1 and s2 and s3
    return KB_true, s1, s2, s3

# Function to evaluate a statement
def evaluate_statement(statement, P, Q, R):
    # statement is a lambda function
    return statement(P, Q, R)

# Statements to check entailment
statements = {
    "R": lambda P,Q,R: R,
    "R -> P": lambda P,Q,R: (not R) or P,
    "Q -> R": lambda P,Q,R: (not Q) or R
}

# Loop through all assignments
print(f"{'P':^5}{'Q':^5}{'R':^5}{'KB':^5}{'R':^5}{'R->P':^7}{'Q->R':^7}")
for values in assignments:
    P, Q, R = values
    KB_true, s1, s2, s3 = evaluate_KB(P, Q, R)
    R_val = evaluate_statement(statements["R"], P, Q, R)
    RtoP_val = evaluate_statement(statements["R -> P"], P, Q, R)
    QtoR_val = evaluate_statement(statements["Q -> R"], P, Q, R)

    print(f"{P!s:^5}{Q!s:^5}{R!s:^5}{KB_true!s:^5}{R_val!s:^5}{RtoP_val!s:^7}{QtoR_val!s:^7}")

# Check entailment
def check_entailment(statement_func):
    for values in assignments:
        P,Q,R = values
        KB_true, _, _, _ = evaluate_KB(P,Q,R)
        if KB_true and not statement_func(P,Q,R):
            return False
    return True

print("\nEntailment results:")
for name, func in statements.items():
    print(f"KB entails {name}? {check_entailment(func)}")
```

Output:-


```
  P    Q    R   KB    R   R->P   Q->R
True True True FalseTrue  True   True
True True FalseFalseFalse True   False
True FalseTrue True True  True   True
True FalseFalseFalseFalse True   True
FalseTrue True FalseTrue  False  True
FalseTrue FalseFalseFalse True   False
FalseFalseTrue True True  False  True
FalseFalseFalseFalseFalse True   True

Entailment results:
KB entails R? True
KB entails R -> P? False
KB entails Q -> R? True

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

Program 7

Implement unification in first order logic

Algorithm:-

unification Algorithm:-

Step 1: If ψ_1 or ψ_2 is a variable or constant, then:

a) If ψ_1 or ψ_2 are identical, then return NIL

b) Else if ψ_1 is a variable,

a. then if ψ_1 occurs in ψ_2, then return Failure

b. Else return {(ψ_2 / ψ_1)}

c) Else if ψ_2 is a variable,

a. If ψ_2 occurs in ψ_1 then return Failure,

b) Else return {(ψ_1 / ψ_2)}.

d) Else return Failure

Step 2: If the initial Predicate symbol in ψ_1 & ψ_2 are not same, return Failure

Step 3: If ψ_1 & ψ_2 have different number of arguments, then return Failure.

Step 4: Set substitution set (SUBST) to NIL.

Step 5: For i = 1 to the number of elements in ψ

a) call unify function with ith element of ψ_1 & ith element of ψ_2, & put the result into S.

b) If S = failure then return failure.

c) If S ≠ NIL then do,

a) Apply S to the remainder of both L1 & L2.

b) SUBST = APPEND (S, SUBST .

step 6 : Return SUBST.

[illegible] 12/11

ur

en
ilue

of

DIL.
ets in y_1
it g

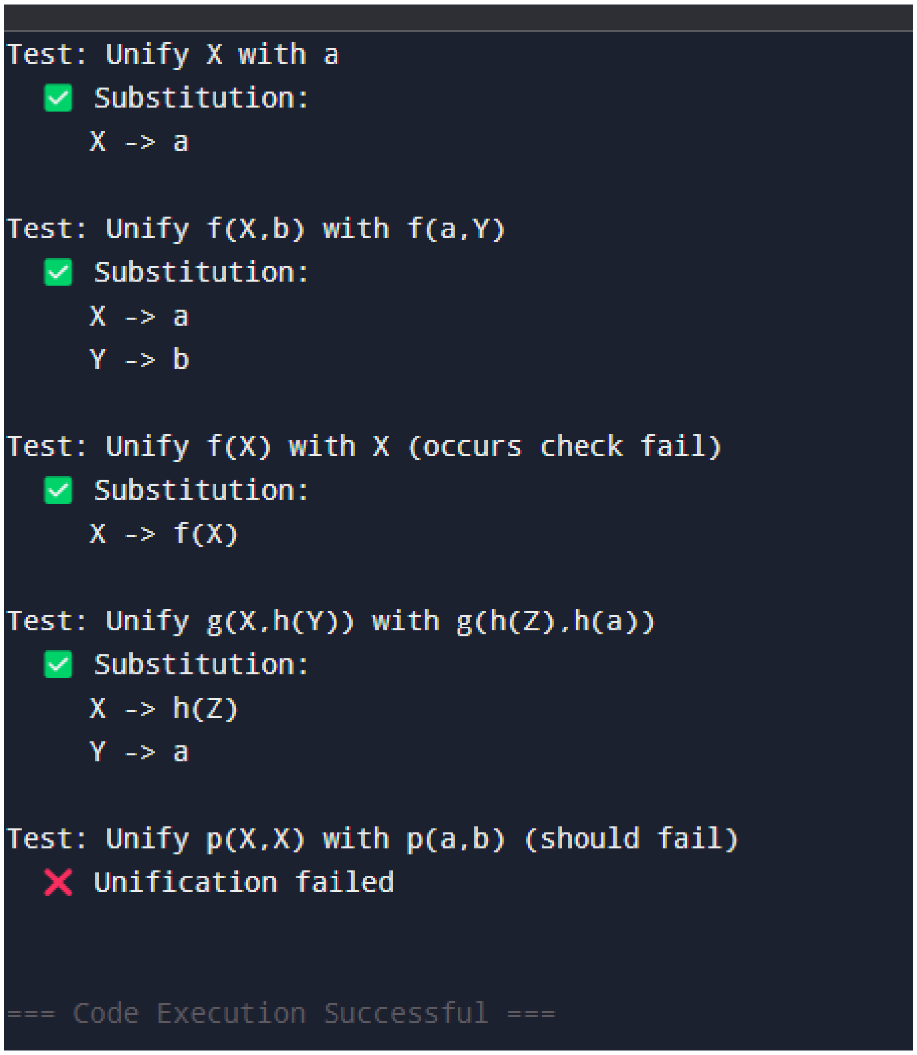
a.

oth

Code:-

```
# ----------------------------------
# Unification Algorithm in First-Order Logic
# ----------------------------------

class Term:
    def _init_(self, name, args=None):
        self.name = name
        self.args = args or []

    def is_variable(self):
        return self.args == [] and self.name[0].isupper()

    def is_constant(self):
        return self.args == [] and self.name[0].islower()

    def _repr_(self):
        if not self.args:
            return self.name
        else:
            return f"{self.name}({', '.join(map(str, self.args))})"


# Occurs check: prevents infinite recursion like X = f(X)
def occurs_check(var, term, subs):
    if var == term:
        return True
    elif term.is_variable() and term.name in subs:
        return occurs_check(var, subs[term.name], subs)
    elif term.args:
        return any(occurs_check(var, t, subs) for t in term.args)
    return False


# Apply substitution to a term
def apply(subs, term):
    if term.is_variable() and term.name in subs:
        return apply(subs, subs[term.name])
    elif term.args:
        return Term(term.name, [apply(subs, t) for t in term.args])
    else:
        return term


# Unification function
def unify(x, y, subs=None):
    if subs is None:
        subs = {}

    x = apply(subs, x)
    y = apply(subs, y)
    if x == y:
        return subs
    elif x.is_variable():
        if occurs_check(x, y, subs):
            return None
        subs[x.name] = y
        return subs
    elif y.is_variable():
        if occurs_check(y, x, subs):
            return None
        subs[y.name] = x
        return subs
    elif x.name == y.name and len(x.args) == len(y.args):
        for a, b in zip(x.args, y.args):
            subs = unify(a, b, subs)
            if subs is None:
                return None
        return subs
    else:
        return None


# -------- Example Test Cases --------

if _name___== "_main_":
    tests = [
        ("Unify X with a", Term("X"), Term("a")),
        ("Unify f(X,b) with f(a,Y)", Term("f", [Term("X"), Term("b")]), Term("f", [Term("a"),
Term("Y")])),
        ("Unify f(X) with X (occurs check fail)", Term("f", [Term("X")]), Term("X")),
        ("Unify g(X,h(Y)) with g(h(Z),h(a))", Term("g", [Term("X"), Term("h", [Term("Y")])]),
Term("g", [Term("h", [Term("Z")]), Term("h", [Term("a")])])),
        ("Unify p(X,X) with p(a,b) (should fail)", Term("p", [Term("X"), Term("X")]), Term("p",
[Term("a"), Term("b")])),
    ]

    for desc, t1, t2 in tests:
        print("Test:", desc)
        result = unify(t1, t2)
        if result is None:
            print(" ❌ Unification failed")
        else:
            print(" ✅ Substitution:")
            for k, v in result.items():
                print(f"     {k} -> {v}")
        print()
```

Output:-



```
Test: Unify X with a
  ✅ Substitution:
     X -> a

Test: Unify f(X,b) with f(a,Y)
  ✅ Substitution:
     X -> a
     Y -> b

Test: Unify f(X) with X (occurs check fail)
  ✅ Substitution:
     X -> f(X)

Test: Unify g(X,h(Y)) with g(h(Z),h(a))
  ✅ Substitution:
     X -> h(Z)
     Y -> a

Test: Unify p(X,X) with p(a,b) (should fail)
  ❌ Unification failed


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

Program 8

Forward Reasoning Algorithm

Algorithm:-

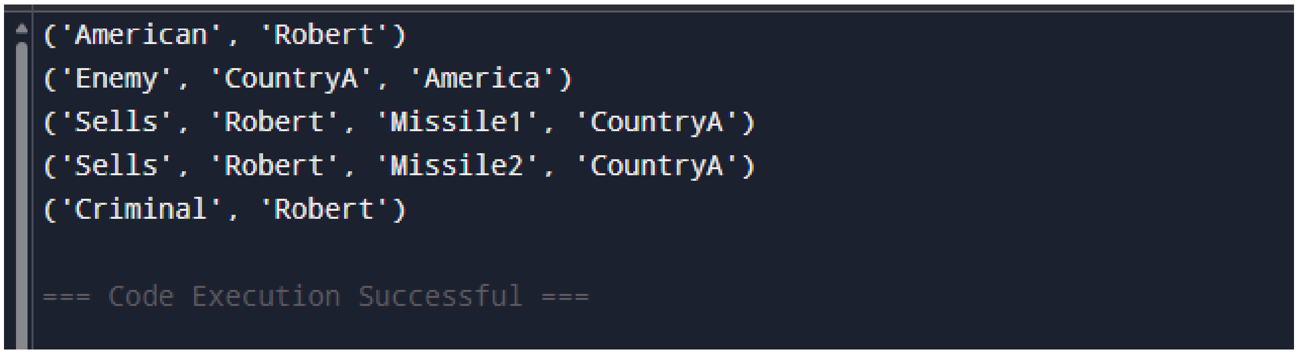
Forward Reasoning :-

```
function FOL-FC-ASK ( KB, $\alpha$ ) returns a
substitution or false
inputs : KB, the knowledge Base, a set of
first-order definite clauses $\alpha$, the query,
    an atomic sentence
local variables : new, the new sentences
    inferred on each iteration.
repeat until new is empty
    new ← { }
  for each rule in KB do
     ( $P_1 \wedge \cdots \wedge P_n \Rightarrow q$ ) ← Standardize-variable
for each $\theta$ such that SUBST ( $\theta$, $P_1 \wedge \cdots \wedge P_n$ )
              = SUBST ( $\theta$, $P_1' \wedge \cdots \wedge P_n'$ )
       for some $P_1'$, $\cdots$ $P_n'$ in KB
       $q'$ ← SUBST ( $\theta$, $q$ )
     if $q'$ does not unify with some sentence
already in KB or new then
        add $q'$ to new
           $\phi$ ← UNIFY ( $q'$, $\alpha$ )
           if $\phi$ is not fail then return $\phi$
     add new to KB
return false.
```

Code:-

```
# Facts as tuples (Predicate, Arguments)
facts = [
    ('American', 'Robert'),
    ('Enemy', 'CountryA', 'America'),
    ('Sells', 'Robert', 'Missile1', 'CountryA'),
    ('Sells', 'Robert', 'Missile2', 'CountryA')
]

# Rule: American sells weapons to hostile nation -> Criminal
def infer_criminal(facts):
    new_facts = []
    for fact in facts:
        if fact[0] == 'American':
            person = fact[1]
            for sell in facts:
                if sell[0] == 'Sells' and sell[1] == person:
                    weapon, country = sell[2], sell[3]
                for enemy in facts:
                    if enemy[0] == 'Enemy' and enemy[1] == country and enemy[2] == 'America':
                        criminal_fact = ('Criminal', person)
                        if criminal_fact not in facts and criminal_fact not in new_facts:
                            new_facts.append(criminal_fact)
    return new_facts

# Forward chaining
while True:
    new_facts = infer_criminal(facts)
    if not new_facts:
        break
    facts.extend(new_facts)

# Result
for f in facts:
    print(f)
```

Output:-

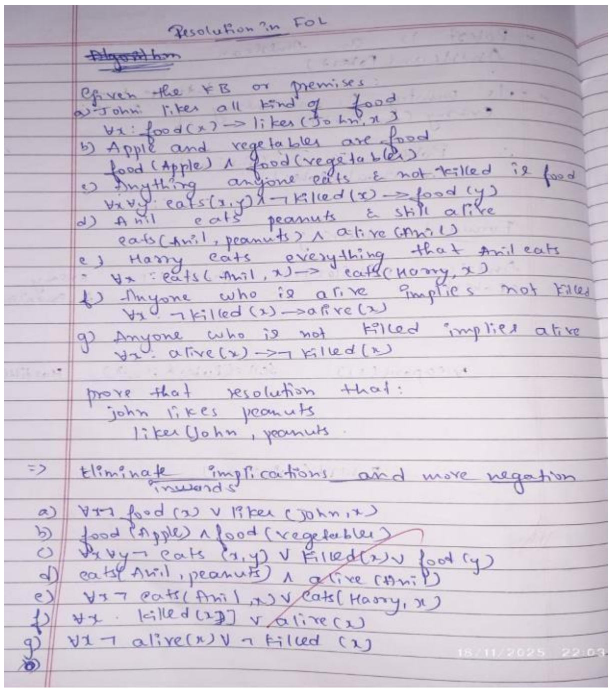


```
('American', 'Robert')
('Enemy', 'CountryA', 'America')
('Sells', 'Robert', 'Missile1', 'CountryA')
('Sells', 'Robert', 'Missile2', 'CountryA')
('Criminal', 'Robert')

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

Program 9

Resolution in FOL

Algorithm:-



Resolution in FOL

~~Algorithm~~

Given the KB or premises

a) John likes all kind of food
   $\forall x: food(x) \rightarrow likes(John, x)$

b) Apple and vegetables are food
   $food(Apple) \wedge food(vegetables)$

c) Anything anyone eats & not killed is food
   $\forall x \forall y: eats(x,y) \wedge \neg killed(x) \rightarrow food(y)$

d) Anil eats peanuts & still alive
   $eats(Anil, peanuts) \wedge alive(Anil)$

e) Harry eats everything that Anil eats
   $\forall x: eats(Anil, x) \rightarrow eats(Harry, x)$

f) Anyone who is alive implies not killed
   $\forall x: \neg killed(x) \rightarrow alive(x)$

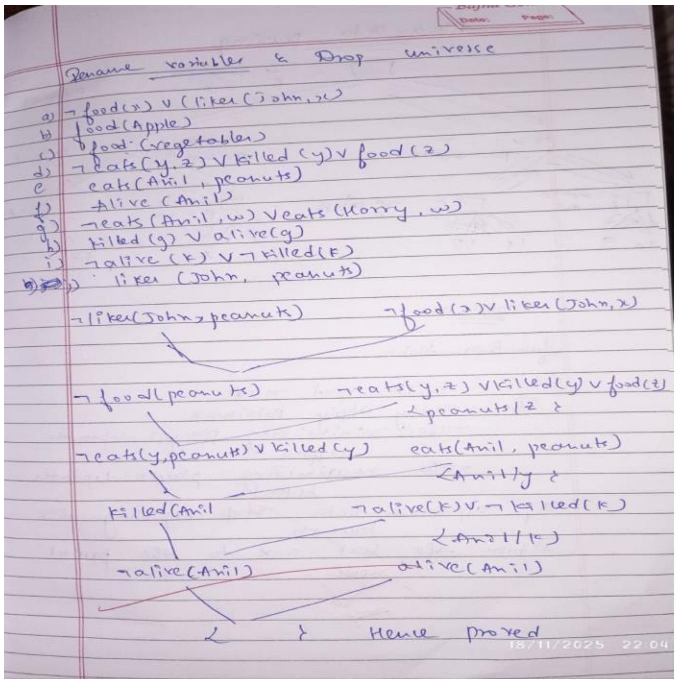
g) Anyone who is not killed implies alive
   $\forall x: alive(x) \rightarrow \neg killed(x)$

prove that resolution that:
john likes peanuts
likes(John, peanuts

=> Eliminate implications and move negation inwards

a) $\forall x \neg food(x) \vee likes(John, x)$
b) $food(Apple) \wedge food(vegetables)$
c) $\forall x \forall y \neg eats(x,y) \vee killed(x) \vee food(y)$
d) $eats(Anil, peanuts) \wedge alive(Anil)$
e) $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$
f) $\forall x\ killed(x) \vee alive(x)$
g) $\forall x \neg alive(x) \vee \neg killed(x)$

18/11/2025 22:03



Rename variables & Drop universe

a) $\neg food(x) \vee likes(John, x)$
b) $food(Apple)$
c) $food(vegetables)$
d) $\neg eats(y,z) \vee killed(y) \vee food(z)$
e) $eats(Anil, peanuts)$
f) $Alive(Anil)$
g) $\neg eats(Anil, w) \vee eats(Harry, w)$
h) $killed(g) \vee alive(g)$
i) $\neg alive(k) \vee \neg killed(k)$
j) $likes(John, peanuts)$

$\neg likes(John, peanuts)$ $\neg food(x) \vee likes(John, x)$

$\neg food(peanuts)$ $\neg eats(y,z) \vee killed(y) \vee food(z)$
{peanuts/z}

$\neg eats(y, peanuts) \vee killed(y)$ $eats(Anil, peanuts)$
{Anil/y}

$killed(Anil$ $\neg alive(k) \vee \neg killed(k)$
{Anil/k}

$\neg alive(Anil)$ $alive(Anil)$

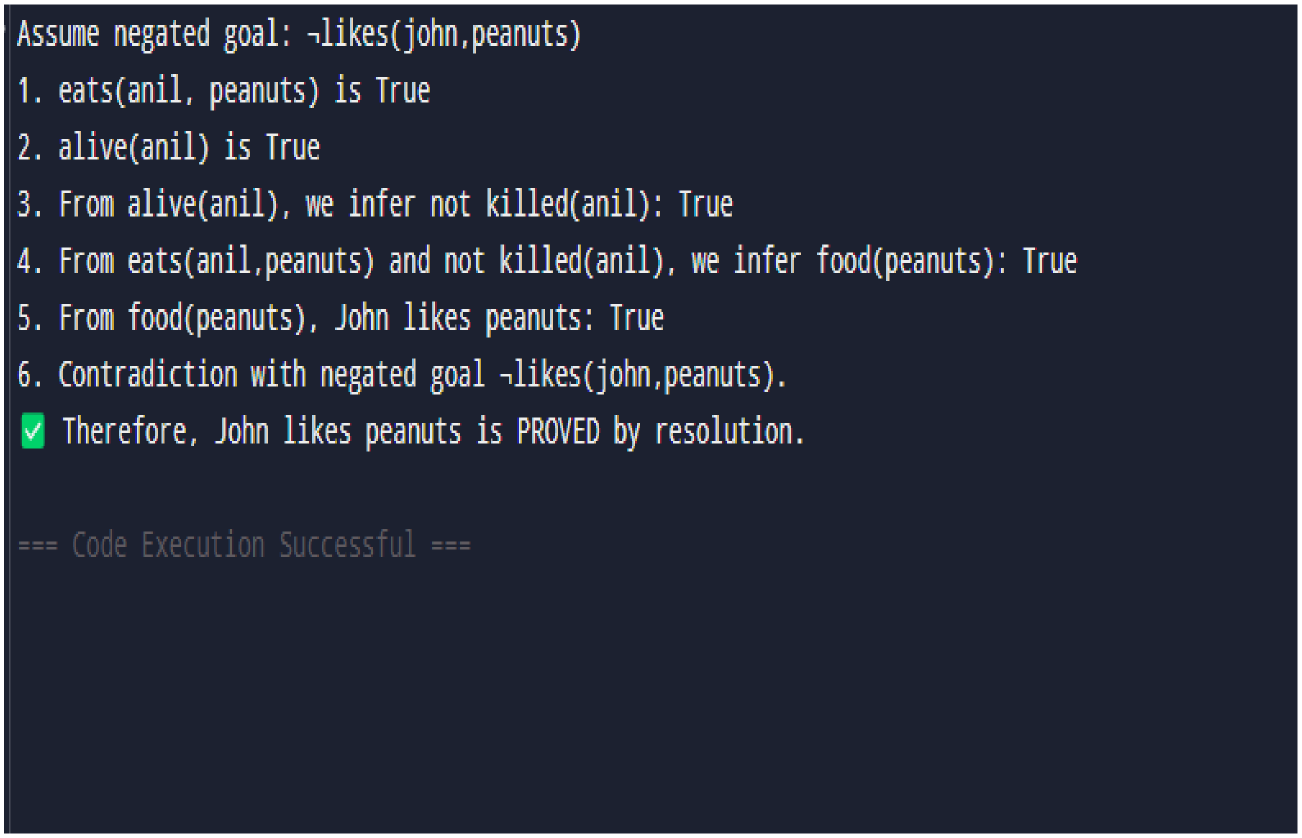
{ } Hence proved

18/11/2025 22:04

Code:-

```
# Resolution Proof for "John likes peanuts"

# Facts and rules (simulated as Python logic)
def proof():
    # Premises
    john_likes_food = lambda y: food(y)  # John likes all kinds of food if it's food
    eats = {
        ('anil', 'peanuts'): True,
    }
    alive = {
        'anil': True
    }

    # Rules
    def is_food(y):
        # Apple and vegetables are food
        if y in ['apple', 'vegetables']:
            return True
        # Rule: Anything anyone eats and not killed is food
        for (x, item) in eats:
            if item == y and not killed(x):
                return True
        return False

    def killed(x):
        # If alive(x) then not killed(x)
        if alive.get(x, False):
            return False
        # If not alive(x) then possibly killed(x)
        return True

    def food(y):
        return is_food(y)

    # Negated goal: assume John does NOT like peanuts
    goal = "¬likes(john,peanuts)"
    print(f"Assume negated goal: {goal}")

    # Step 1: Anil eats peanuts and is alive
    print("1. eats(anil, peanuts) is True")
    print("2. alive(anil) is True")

    # Step 2: From 'alive(anil)' => 'not killed(anil)'
    not_killed_anil = not killed('anil')
    print(f"3. From alive(anil), we infer not killed(anil): {not_killed_anil}")

    # Step 3: From 'Anil eats peanuts' and 'not killed(anil)' => 'food(peanuts)'
    food_peanuts = eats[('anil', 'peanuts')] and not_killed_anil
    print(f"4. From eats(anil,peanuts) and not killed(anil), we infer food(peanuts): {food_peanuts}")

    # Step 4: From 'food(peanuts)' => 'John likes peanuts'
    if food_peanuts:
        likes_john_peanuts = True
    else:
        likes_john_peanuts = False
    print(f"5. From food(peanuts), John likes peanuts: {likes_john_peanuts}")

    # Step 5: Contradiction with negated goal
    if likes_john_peanuts:
        print("6. Contradiction with negated goal ¬likes(john,peanuts).")
        print("✅ Therefore, John likes peanuts is PROVED by resolution.")
    else:
        print("❌ Could not prove John likes peanuts.")

proof()
```

Output:-

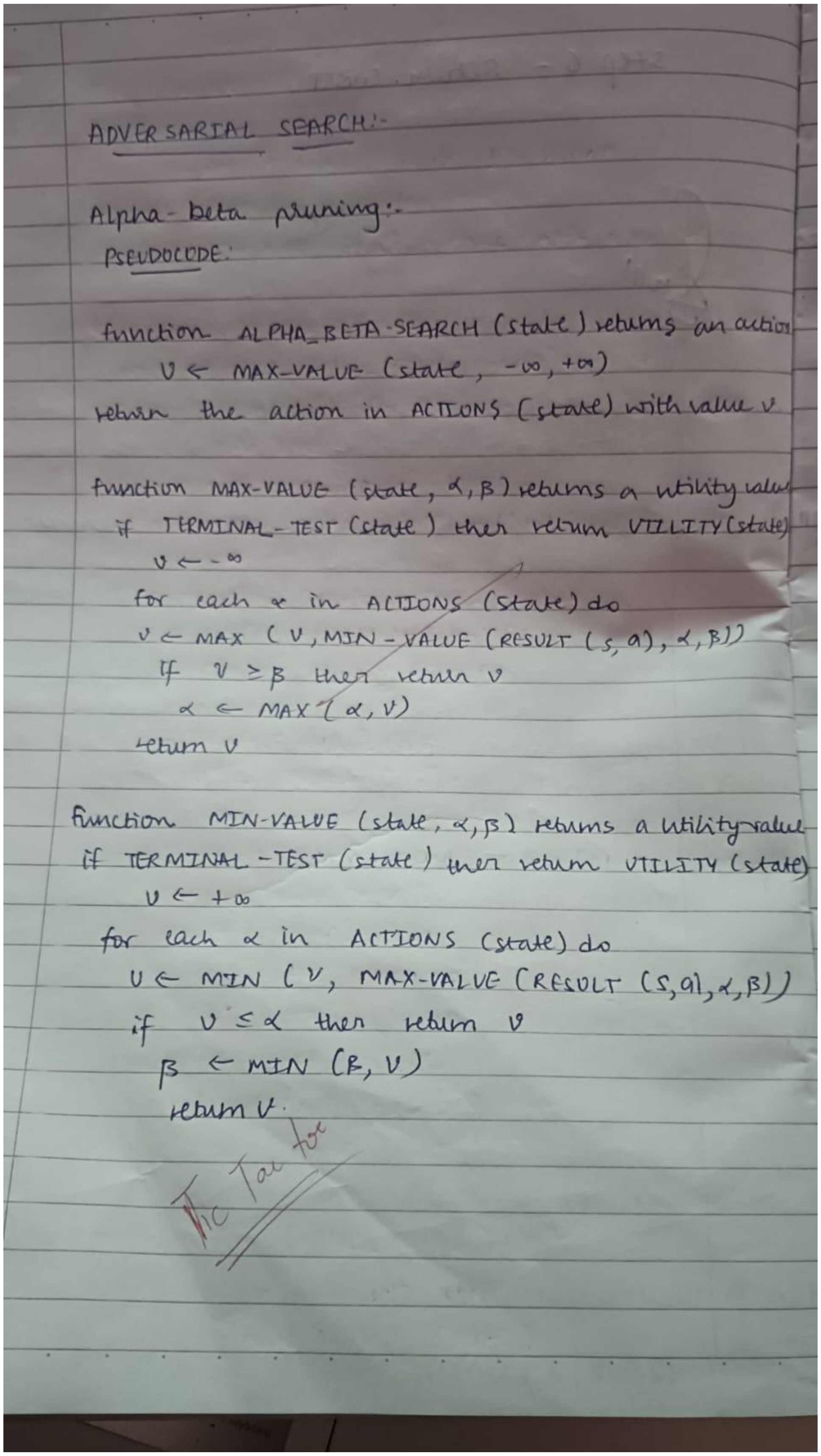


```
Assume negated goal: ¬likes(john,peanuts)
1. eats(anil, peanuts) is True
2. alive(anil) is True
3. From alive(anil), we infer not killed(anil): True
4. From eats(anil,peanuts) and not killed(anil), we infer food(peanuts): True
5. From food(peanuts), John likes peanuts: True
6. Contradiction with negated goal ¬likes(john,peanuts).
✅ Therefore, John likes peanuts is PROVED by resolution.


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

Program 10

Alpha beta pruning

Algorithm:-



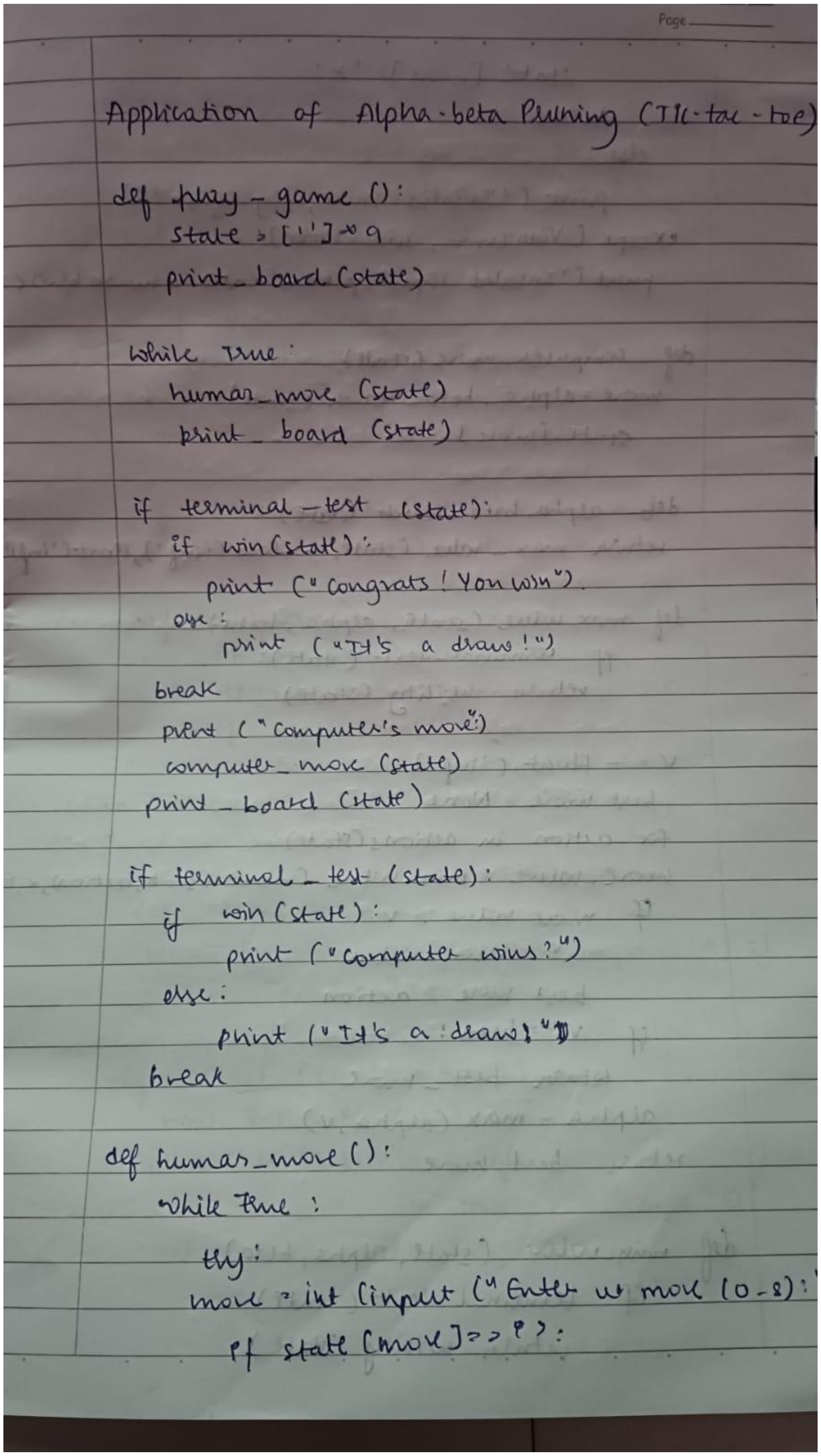
ADVERSARIAL SEARCH:-

Alpha-beta pruning:-

PSEUDOCODE:

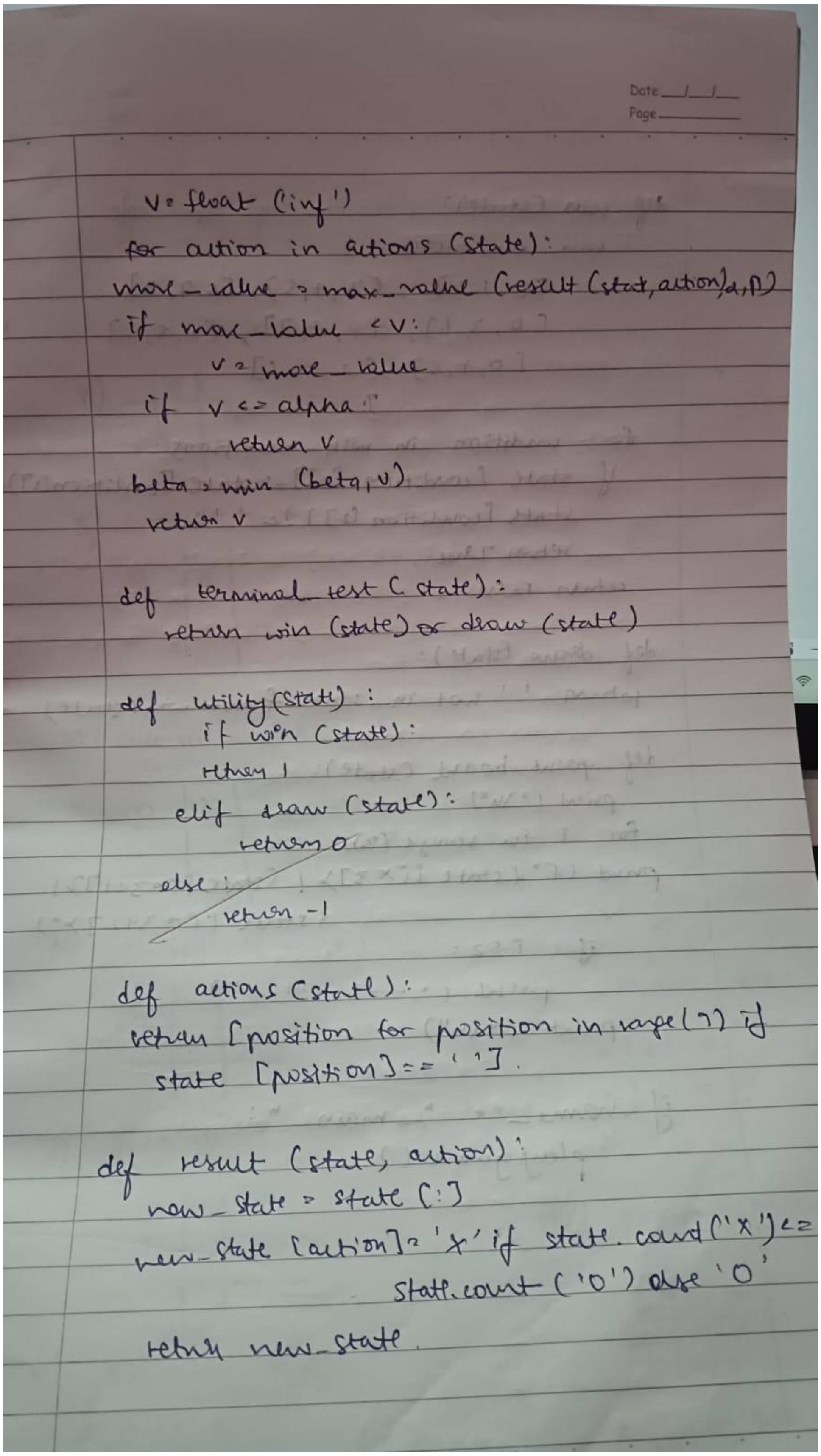
```
function ALPHA_BETA-SEARCH (state) returns an action
    v ← MAX-VALUE (state, -∞, +∞)
return the action in ACTIONS (state) with value v

function MAX-VALUE (state, α, β) returns a utility value
  if TERMINAL-TEST (state) then return UTILITY(state)
    v ← -∞
    for each a in ACTIONS (state) do
    v ← MAX ( v, MIN - VALUE (RESULT (s, a), α, β))
      if v ≥ β then return v
       α ← MAX (α, v)
    return v

function MIN-VALUE (state, α, β) returns a utility value
 if TERMINAL -TEST (state) then return UTILITY (state)
     v ← +∞
  for each a in ACTIONS (state) do
    v ← MIN ( v, MAX-VALUE (RESULT (s, a), α, β))
    if v ≤ α then return v
     β ← MIN (β, v)
     return v.
```

[illegible]



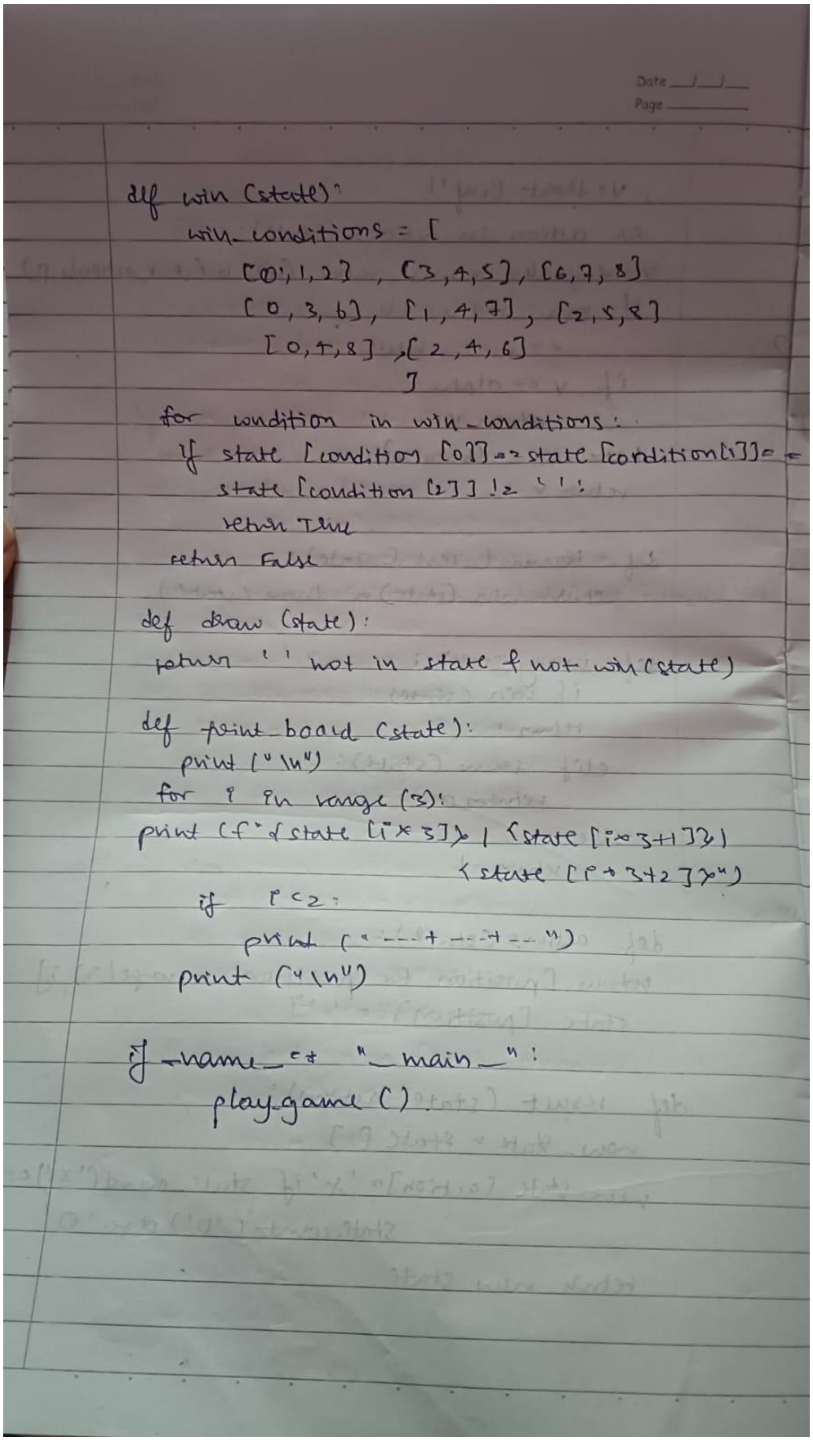
Application of Alpha-beta Pruning (Tic-tac-toe)

```
def play-game ():
    state = [' ']*9
    print_board (state)

 while True:
    human_move (state)
    print_board (state)

 if terminal_test (state):
    if win(state):
       print ("Congrats! You win")
    else:
       print ("It's a draw!")
   break
   print ("computer's move")
   computer_move (state)
  print_board (state)

 if terminal_test (state):
    if win (state):
       print ("computer wins?")
    else:
       print ("It's a draw!")
   break

def human_move ():
   while True:
     try:
     move = int (input ("Enter ur move (0-8):
       if state [move]== ' ':
            state [move] = 'x'
          break
       else:
            print ("Invalid move, try again!")
    except (ValueError, IndexError):
       print ("Invalid input, please enter a number b/w 0 & 8.")

def computer_move (state):
   move = alpha_beta_search (state)
      state [move] = 'O'

def alpha_beta_search (state):
  return max_value (state, - float ('inf'), float ('inf'))

def max_value (state, alpha, beta):
       if terminal_test (state):
          return utility (state)

  v = - float ('inf')
   best_move = None
  for action in actions (state):
  move_value = min_value (result (state, action), α, β)
    if move_value > v:
            v = move_value
          best_move = action
    if v >= beta:
       return best_move
  alpha = max (alpha, v)
return best_move

def min_value (state, alpha, beta):
   if terminal_test (state):
       return utility (state)
```



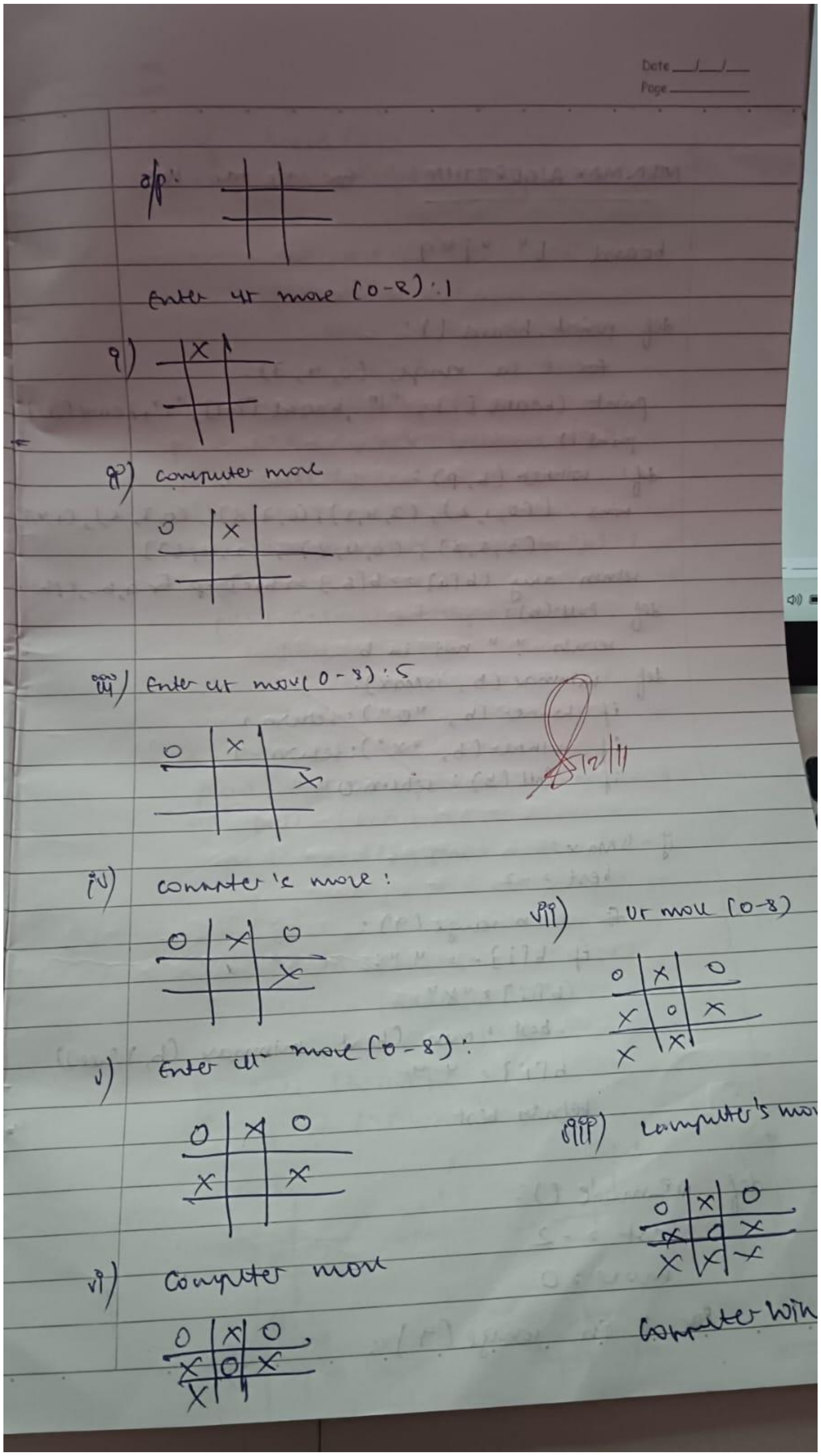
Date__/__/__
Page_____

```
    v = float ('inf')
    for action in actions (state):
    move_value = max_value (result (stat, action), α, β)
    if move_value < v:
          v = move_value
     if v <= alpha:
          return v
    beta = min (beta, v)
    return v

def terminal_test ( state):
    return win (state) or draw (state)

def utility (state) :
      if win (state):
      return 1
    elif draw (state):
          return 0
   ~~else :~~
       ~~return -1~~

def actions (state):
  return [position for position in range(9) if
    state [position] == ' '].

def result (state, action):
  new_state = state [:]
  new_state [action] = 'x' if state. count ('x') <=
                       state.count ('O') else 'O'

  return new_state
```



Date __/__/__
Page ______

```
def win (state):
    win_conditions = [
        [0,1,2] , [3,4,5], [6,7,8]
        [0, 3, 6], [1 , 4, 7] , [2,5,8]
         [0,4,8] ,[2 ,4, 6]
                 ]
    for condition in win_conditions:
      if state [condition [0]]==state [condition[1]]==
         state [condition [2]] != ' ':
         return True
    return False

def draw (state):
   return ' ' not in state & not win(state)

def print_board (state):
   print ("\n")
  for i in range (3):
print (f"{state [i*3]} | {state [i*3+1]} |
                      {state [i*3+2]}")
     if  i<2:
          print ("---+---+--")
   print ("\n")

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



o/p:

```
 | | 
-+-+-
 | | 
-+-+-
 | | 
```

Enter ur move (0-8): 1

i)

```
 |X| 
-+-+-
 | | 
-+-+-
 | | 
```

ii) computer move

```
O|X| 
-+-+-
 | | 
-+-+-
 | | 
```

iii) Enter ur move (0-8): 5

```
O|X| 
-+-+-
 | |X
-+-+-
 | | 
```

iv) computer's move:

```
O|X|O
-+-+-
 | |X
-+-+-
 | | 
```

v) Enter ur move (0-8):

```
O|X|O
-+-+-
X| |X
-+-+-
 | | 
```

vi) Computer move

```
O|X|O
-+-+-
X|O|X
-+-+-
X| | 
```

vii) Ur move (0-8)

```
O|X|O
-+-+-
X|O|X
-+-+-
X|X| 
```

viii) Computer's mo

```
O|X|O
-+-+-
X|O|X
-+-+-
X|X|X
```

Computer win

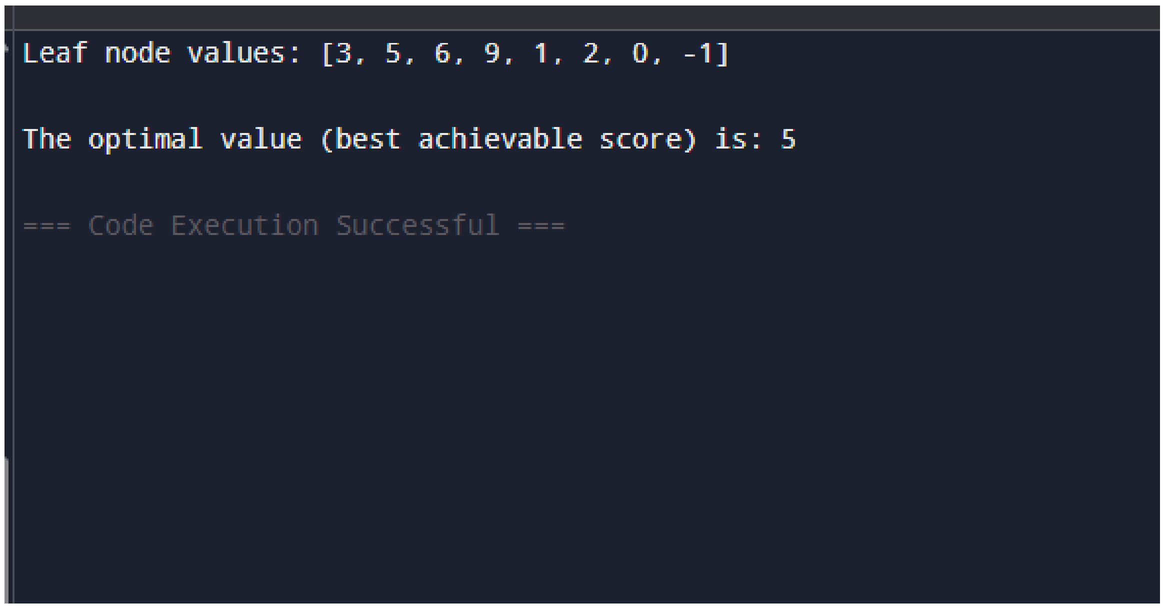
[illegible] 12/11

Code:-

```
# Alpha-Beta Pruning in Python

# Minimax function with Alpha-Beta Pruning
def alpha_beta(depth, node_index, maximizing_player, values, alpha, beta, max_depth):
    # Base case: when we reach the maximum depth (leaf node)
    if depth == max_depth:
        return values[node_index]

    if maximizing_player:
        best = float('-inf')

        # Traverse left and right children
        for i in range(2):
            val = alpha_beta(depth + 1, node_index * 2 + i, False, values, alpha, beta, max_depth)
            best = max(best, val)
            alpha = max(alpha, best)

            # Alpha-Beta Pruning condition
            if beta <= alpha:
                break
        return best

    else:
        best = float('inf')

        # Traverse left and right children
        for i in range(2):
            val = alpha_beta(depth + 1, node_index * 2 + i, True, values, alpha, beta, max_depth)
            best = min(best, val)
            beta = min(beta, best)

            # Alpha-Beta Pruning condition
            if beta <= alpha:
                break
        return best
```

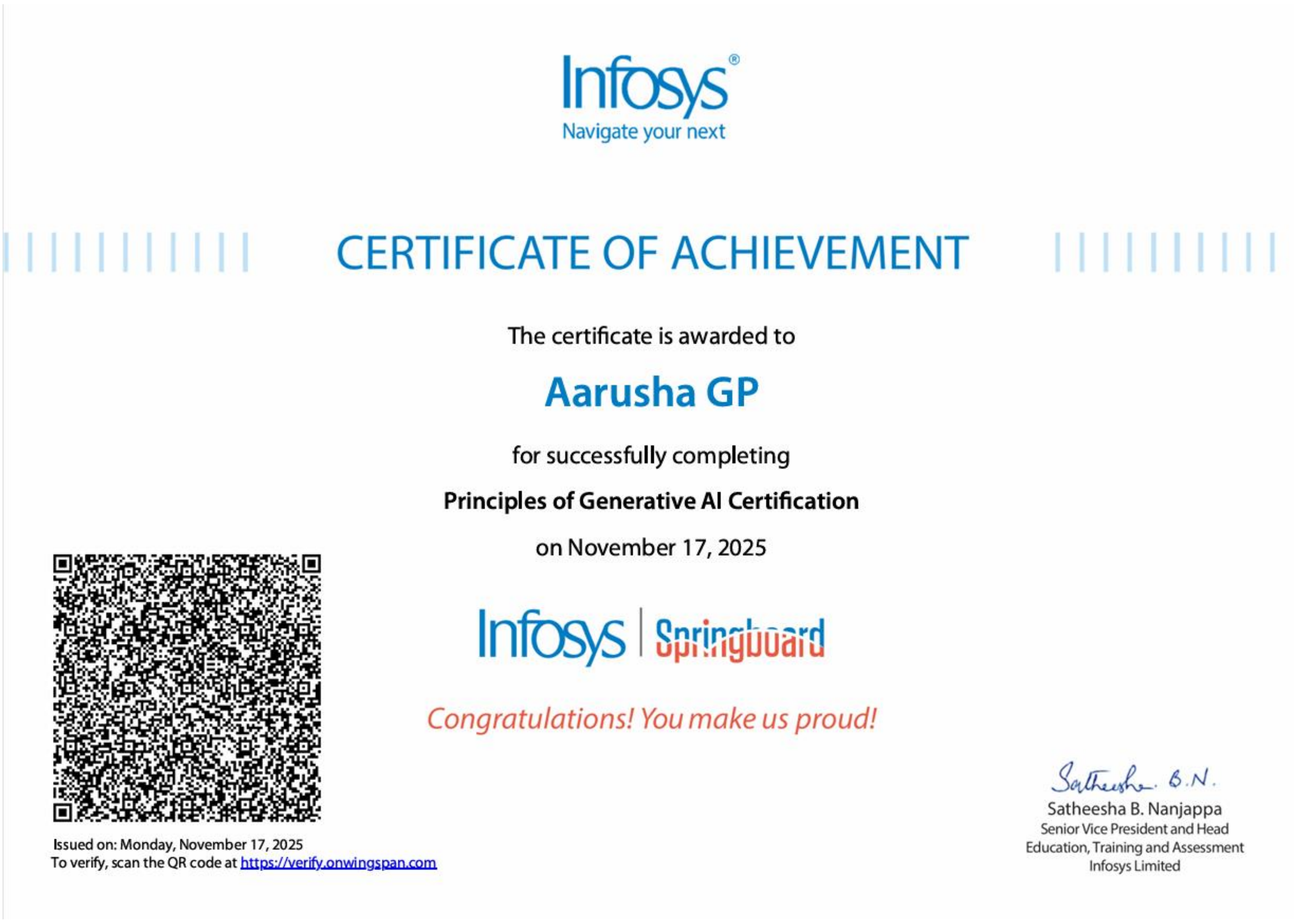
```
# Example usage:
if __name__ == "__main__":
    # Example game tree (values at leaf nodes)
    values = [3, 5, 6, 9, 1, 2, 0, -1]
    max_depth = 3  # since 2^3 = 8 leaf nodes

    print("Leaf node values:", values)
    optimal_value = alpha_beta(0, 0, True, values, float('-inf'), float('inf'), max_depth)
    print("\nThe optimal value (best achievable score) is:", optimal_value)
```

Output:-



```
Leaf node values: [3, 5, 6, 9, 1, 2, 0, -1]

The optimal value (best achievable score) is: 5

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

AI CERTIFICATION:



Infosys®
Navigate your next

CERTIFICATE OF ACHIEVEMENT

The certificate is awarded to

Aarusha GP

for successfully completing

Principles of Generative AI Certification

on November 17, 2025

Infosys | Springboard

Congratulations! You make us proud!

Issued on: Monday, November 17, 2025
To verify, scan the QR code at https://verify.onwingspan.com

Satheesha B.N.
Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited