VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Sinchana Hemanth (1BM23CS330)

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 **Sinchana Hemanth (1BM23CS330),** 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.

Surabhi S Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

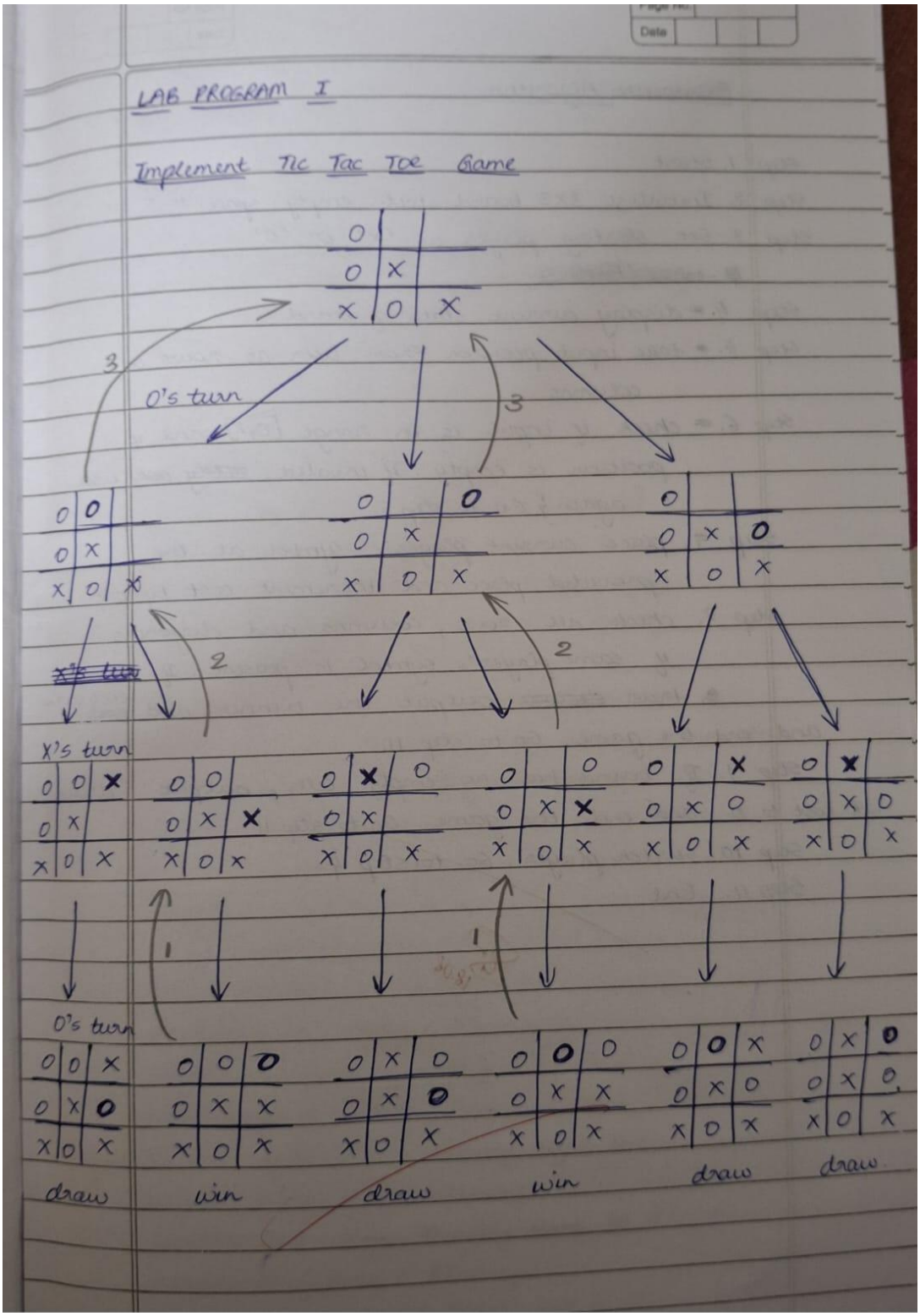
Index

Github Link: https://github.com/SinchanaHemanth/AIpythonLab_SinchanaHemanth.git

Program 1

Implement Tic –Tac –Toe Game

ALGORITHM:



Date

~~Pseudocode~~ Algorithm

Step 1. Start

Step 2. Initialize 3×3 board with empty space " "

Step 3. Set starting player as 'X' or 'O'

Step 4. display current state of board

Step 5. take input position from user as rows and columns

Step 6. check if input is in range [0,2] and if position is empty. If invalid, ask user again & Go to step 5

Step 7. place current player's symbol at the provided place and increment cost by one

Step 8. check all rows, columns and diagonals if same player's symbol is present. If its true output the winner and cost value and end the game. Go to step 11.

Step 9. If board has no empty cells, output "draw" set cost to 0 and end the game. Go to step 11

Step 10. switch player. Go to step 4

Step 11. End

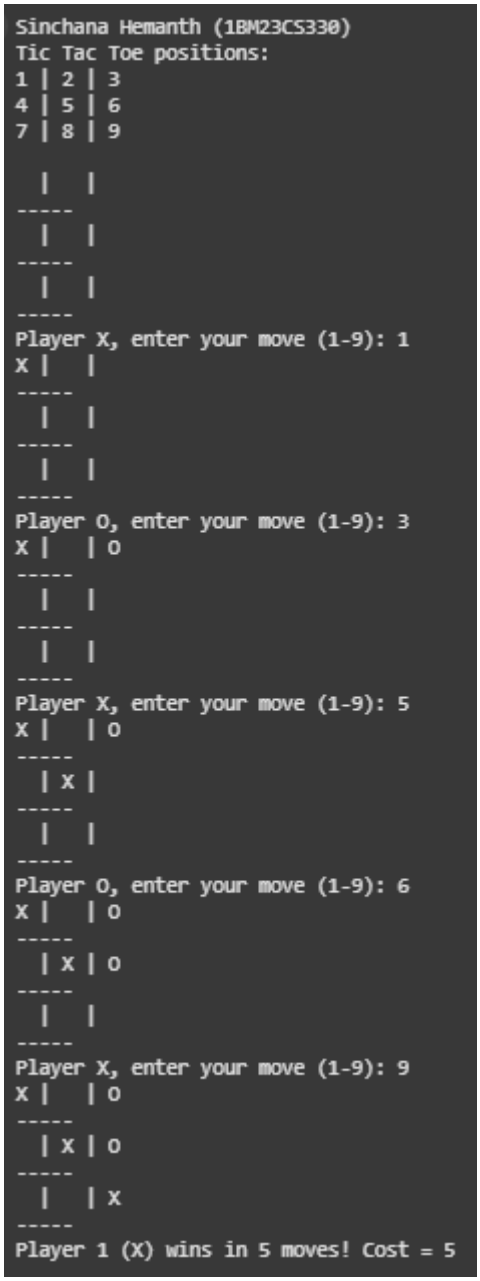
1808

CODE:

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

def check_winner(board):
    for row in board:
        if row.count(row[0]) == 3 and row[0] != " ":
            return row[0]
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] != " ":
            return board[0][col]
    if board[0][0] == board[1][1] == board[2][2] != " ":
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != " ":
        return board[0][2]
    return None

def play_tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = "X"
    moves = 0
    print("Tic Tac Toe positions:")
    print("1 | 2 | 3")
    print("4 | 5 | 6")
    print("7 | 8 | 9\n")

    while moves < 9:
        print_board(board)
        try:
            pos = int(input(f"Player {current_player}, enter your move (1-9): "))
            if pos < 1 or pos > 9:
                print("Invalid position! Choose between 1 and 9.")
                continue
            row, col = divmod(pos - 1, 3)
            if board[row][col] != " ":
                print("Cell already taken! Try again.")
                continue
            board[row][col] = current_player
            moves += 1
            winner = check_winner(board)
            if winner:
                print_board(board)
                print(f"Player {'1 (X)' if winner == 'X' else '2 (O)'} wins in {moves} moves! Cost = {moves}")
                return
            current_player = "O" if current_player == "X" else "X"

        except ValueError:
            print("Please enter a valid number between 1 and 9.")
```

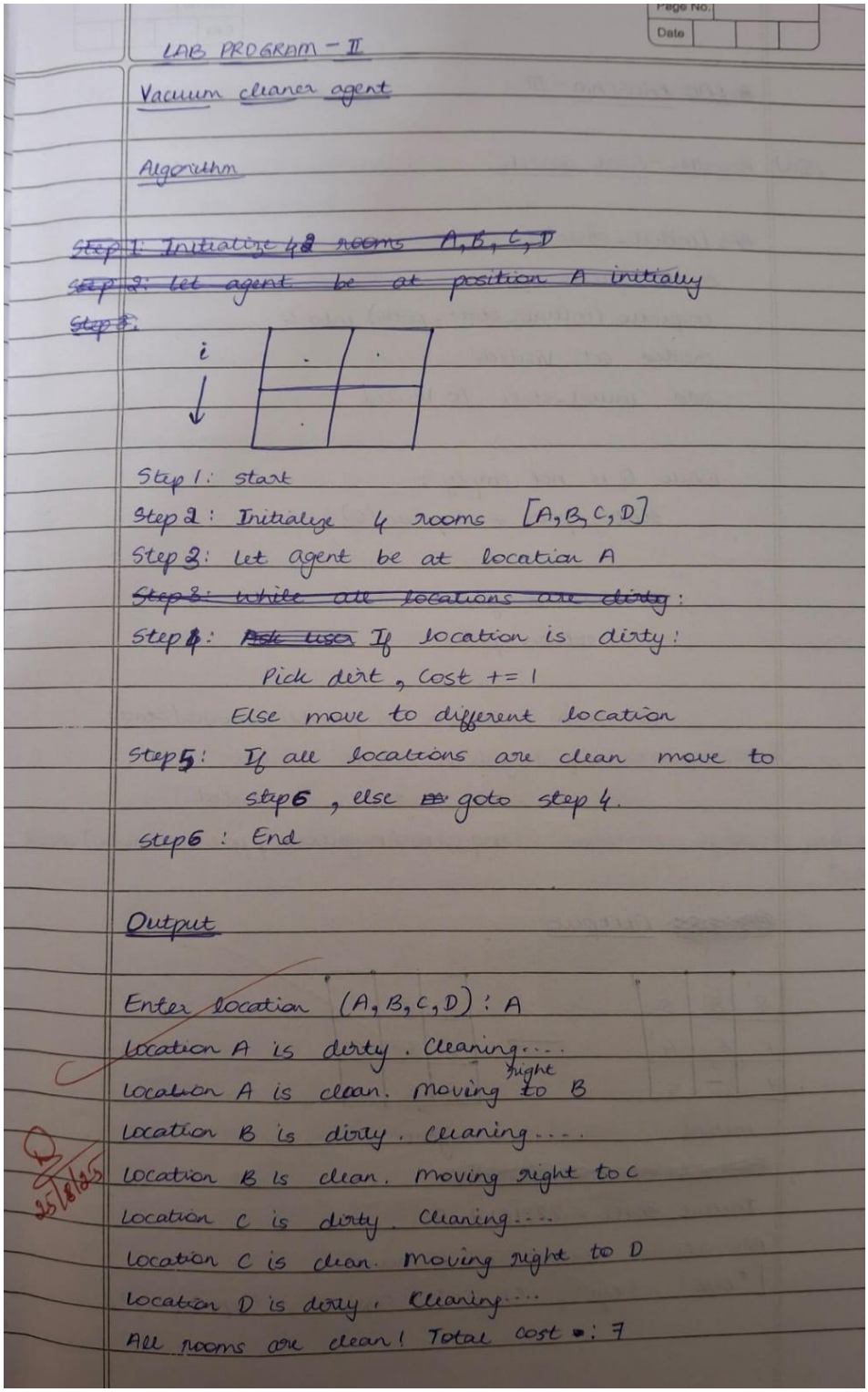
```
    print_board(board)
    print("It's a Draw! Cost: 0")
play_tic_tac_toe()
```

OUTPUT:



Implement vacuum cleaner agent.

ALGORITHM:



LAB PROGRAM - II

Vacuum cleaner agent

Algorithm

~~Step 1: Initialize 4 rooms A, B, C, D~~

~~Step 2: let agent be at position A initially~~

~~Step 3:~~

Step 1: start

Step 2: Initialize 4 rooms [A, B, C, D]

Step 3: let agent be at location A

~~Step 3: while all locations are dirty~~:

Step 4: ~~Pick this~~ If location is dirty:

Pick dirt, Cost += 1

Else move to different location

Step 5: If all locations are clean move to step 6, else ~~go~~ goto step 4.

Step 6: End

Output

Enter location (A, B, C, D): A

location A is dirty. Cleaning....

location A is clean. moving right to B

Location B is dirty. Cleaning....

Location B is clean. moving right to C

Location C is dirty. Cleaning....

location C is clean. moving right to D

location D is dirty. Cleaning....

All rooms are clean! Total cost: 7

25/8/25

CODE:

```
class VacuumEnvironment4Rooms:
    def __init__(self):
        self.rooms = {'A': True, 'B': True, 'C': True, 'D': True}
        self.agent_location = None
        self.room_order = ['A', 'B', 'C', 'D']

    def is_dirty(self, location):
        return self.rooms[location]

    def clean(self, location):
        self.rooms[location] = False

    def move_agent(self, location):
        self.agent_location = location

    def get_percept(self):
        return (self.agent_location, self.is_dirty(self.agent_location))

    def all_clean(self):
        return all(not dirty for dirty in self.rooms.values())

class VacuumAgent4Rooms:
    def __init__(self, environment):
        self.env = environment
```

```
        self.room_order = environment.room_order
        self.direction = 1

    def act(self):
        location, dirty = self.env.get_percept()
        if dirty:
            print(f"Location {location} is dirty. Cleaning...")
            self.env.clean(location)
            return 'Suck'
        current_index = self.room_order.index(location)
        next_index = current_index + self.direction
        if next_index >= len(self.room_order):
            self.direction = -1
            next_index = current_index + self.direction
        elif next_index < 0:
            self.direction = 1
            next_index = current_index + self.direction
        next_location = self.room_order[next_index]
        print(f"Location {location} is clean. Moving {'right' if self.direction == 1 else 'left'} to {next_location}...")
        self.env.move_agent(next_location)
        return 'Move'

def main():
    env = VacuumEnvironment4Rooms()
    start = input("Enter starting location (A, B, C, D): ").strip().upper()
```

```
    while start not in env.room_order:
        start = input("Invalid input. Enter starting location (A, B, C, D): ").strip().upper()
    env.move_agent(start)
    agent = VacuumAgent4Rooms(env)
    steps = 0
    while not env.all_clean():
        agent.act()
        steps += 1
    print(f"All rooms are clean! Total steps taken: {steps}")


if __name__ == "__main__":
    main()
print("Sinchana Hemanth (1BM23CS330)")
```

OUTPUT:

```
Enter starting location (A, B, C, D): A
Location A is dirty. Cleaning...
Location A is clean. Moving right to B...
Location B is dirty. Cleaning...
Location B is clean. Moving right to C...
Location C is dirty. Cleaning...
Location C is clean. Moving right to D...
Location D is dirty. Cleaning...
All rooms are clean! Total steps taken: 7
Sinchana Hemanth (1BM23CS330)
```

Program 2

Using BFS solve 8 puzzle without heuristic approach.

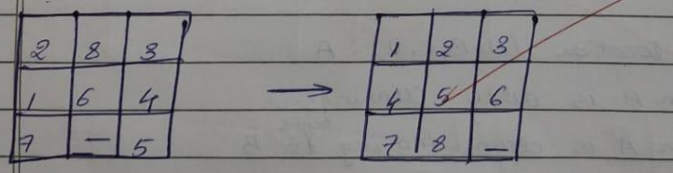
ALGORITHM:

LAB PROGRAM – III

(3a) Breadth – First search

```
BFS (initial_state):
    create queue Q
    enqueue (initial_state, path) into Q
    created set Visited
    add initial_state to Visited

    while Q is not empty:
        state, path = dequeue(Q)

        if state is goal:
            return path

        for each neighbour in possible_moves(state)
            if neighbour not in Visited:
                add neighbour to Visited
                enqueue (neighbour, path + move) into Q
```

Output



initial goal

Initial State = 2831647_5

BFS solution found in 6 moves:

['Left', 'Up', 'Right', 'Down', 'Left', 'Up']

Page No.
Date

Question

Using BFS solve 8 puzzle without heuristic

2 8 3
1 6 4
7 _ 5

→

1 2 3
8 _ 4
7 6 5

initial

goal

① BFS

Right

Left

Up

U (repeat)

U repeats

Solution

CODE:

```
from collections import deque
GOAL_STATE = (1, 2, 3, 8, 0, 4, 7, 6, 5)
MOVES = {
   'left': -1,
   'right': 1,
   'up': -3,
   'down': 3,
}

def is_valid_move(blank_idx, move):
   if move == 'left' and blank_idx % 3 == 0:
      return False
   if move == 'right' and blank_idx % 3 == 2:
      return False
   if move == 'up' and blank_idx < 3:
      return False
   if move == 'down' and blank_idx > 5:
      return False
   return True

def get_neighbors(state):
   neighbors = []
   blank_idx = state.index(0)
   for move, delta in MOVES.items():
      if is_valid_move(blank_idx, move):
         new_idx = blank_idx + delta
         new_state = list(state)
         new_state[blank_idx], new_state[new_idx] = new_state[new_idx], new_state[blank_idx]
         neighbors.append(tuple(new_state))
   return neighbors

def bfs(start_state):
   queue = deque([start_state])
   visited = set([start_state])
   parent = {start_state: None}
   explored_count = 0

   while queue:
      current = queue.popleft()
      explored_count += 1

      if current == GOAL_STATE:
         path = []
         while current:
            path.append(current)
            current = parent[current]
         path.reverse()
         print(f"Total states explored (breadth-wise): {explored_count}")
         return path
```

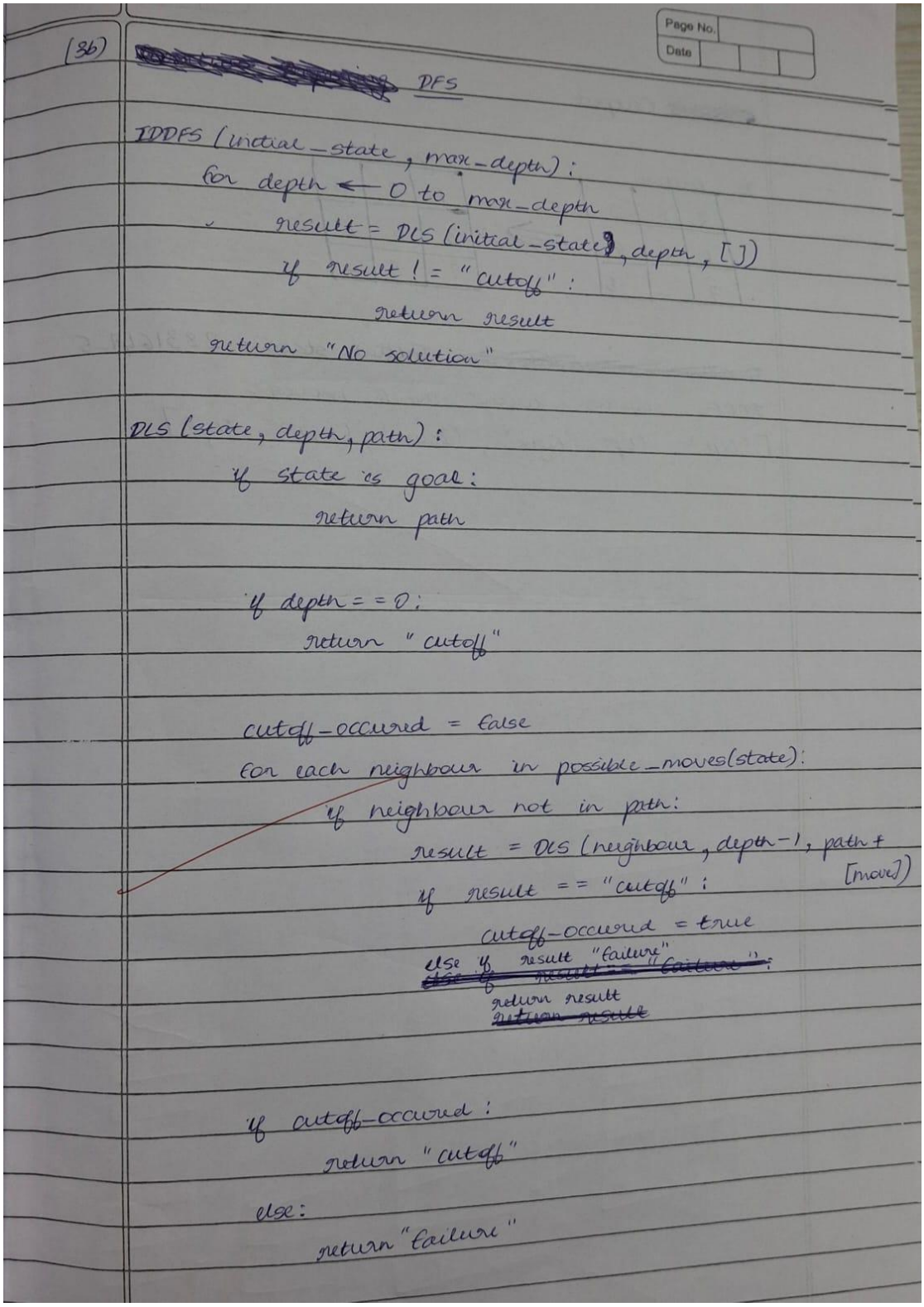
```
        for neighbor in get_neighbors(current):
            if neighbor not in visited:
                visited.add(neighbor)
                parent[neighbor] = current
                queue.append(neighbor)
    print(f"Total states explored (breadth-wise): {explored_count}")
    return None

def print_state(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()

if __name__ == "__main__":
    start = (2, 8, 3,
             1, 6, 4,
             7, 0, 5)

    print("Starting BFS 8-puzzle solver...\nInitial state:")
    print_state(start)
    print("Sinchana Hemanth (1BM23CS330)")
    solution = bfs(start)

    if solution:
        print(f"Solution found in {len(solution)-1} moves:\n")
        for step_num, state in enumerate(solution):
            print(f"Step {step_num}:")
            print_state(state)
    else:
        print("No solution found.")
```

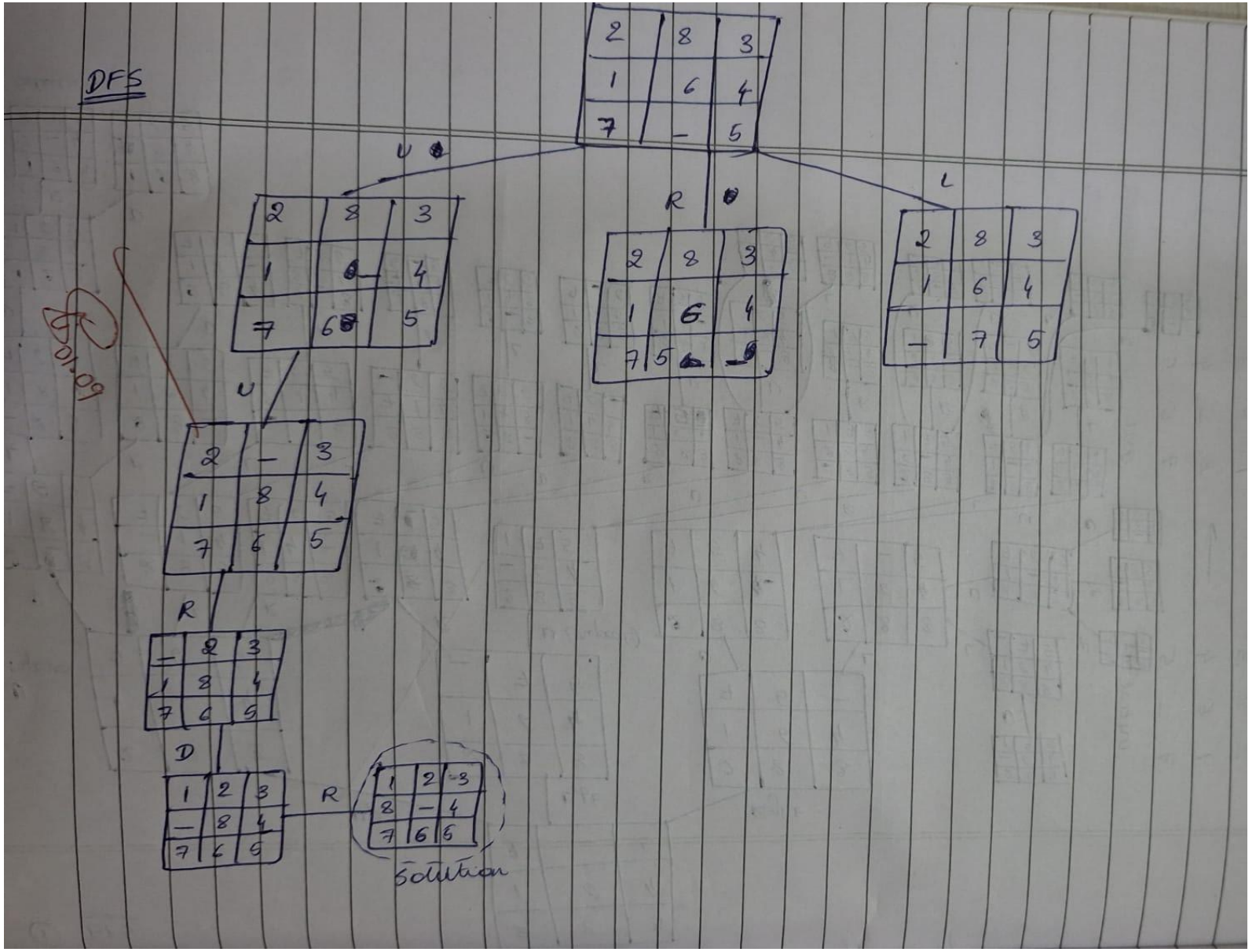
OUTPUT:

```
Starting BFS 8-puzzle solver...
Initial state:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

Sinchana Hemanth (1BM23CS330)
Total states explored (breadth-wise): 58
Solution found in 5 moves:

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

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

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

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

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

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

Using DFS solve 8 puzzle without heuristic approach.

ALGORITHM:




DFS
U
R
L
U
R
D
R
Solution

CODE:

```
GOAL_STATE = (1, 2, 3, 4, 5, 6, 7, 8, 0)
MOVES = {
    'left': -1,
    'right': 1,
    'up': -3,
    'down': 3,
}
def is_valid_move(blank_idx, move):
    if move == 'left' and blank_idx % 3 == 0:
        return False
    if move == 'right' and blank_idx % 3 == 2:
        return False
    if move == 'up' and blank_idx < 3:
        return False
    if move == 'down' and blank_idx > 5:
        return False
    return True
def get_neighbors(state):
    neighbors = []
    blank_idx = state.index(0)
    for move, delta in MOVES.items():
        if is_valid_move(blank_idx, move):
            new_idx = blank_idx + delta
            new_state = list(state)
            new_state[blank_idx], new_state[new_idx] = new_state[new_idx], new_state[blank_idx]
            neighbors.append(tuple(new_state))
    return neighbors
def dfs(start_state, max_depth=50):
    stack = [(start_state, 0)]
    visited = set([start_state])
    parent = {start_state: None}
    while stack:
        current, depth = stack.pop()
        if current == GOAL_STATE:
            path = []
            while current:
                path.append(current)
                current = parent[current]
            path.reverse()
            return path

        if depth < max_depth:
            for neighbor in get_neighbors(current):
                if neighbor not in visited:
                    visited.add(neighbor)
                    parent[neighbor] = current
                    stack.append((neighbor, depth + 1))
    return None
def print_state(state):
    for i in range(0, 9, 3):
```

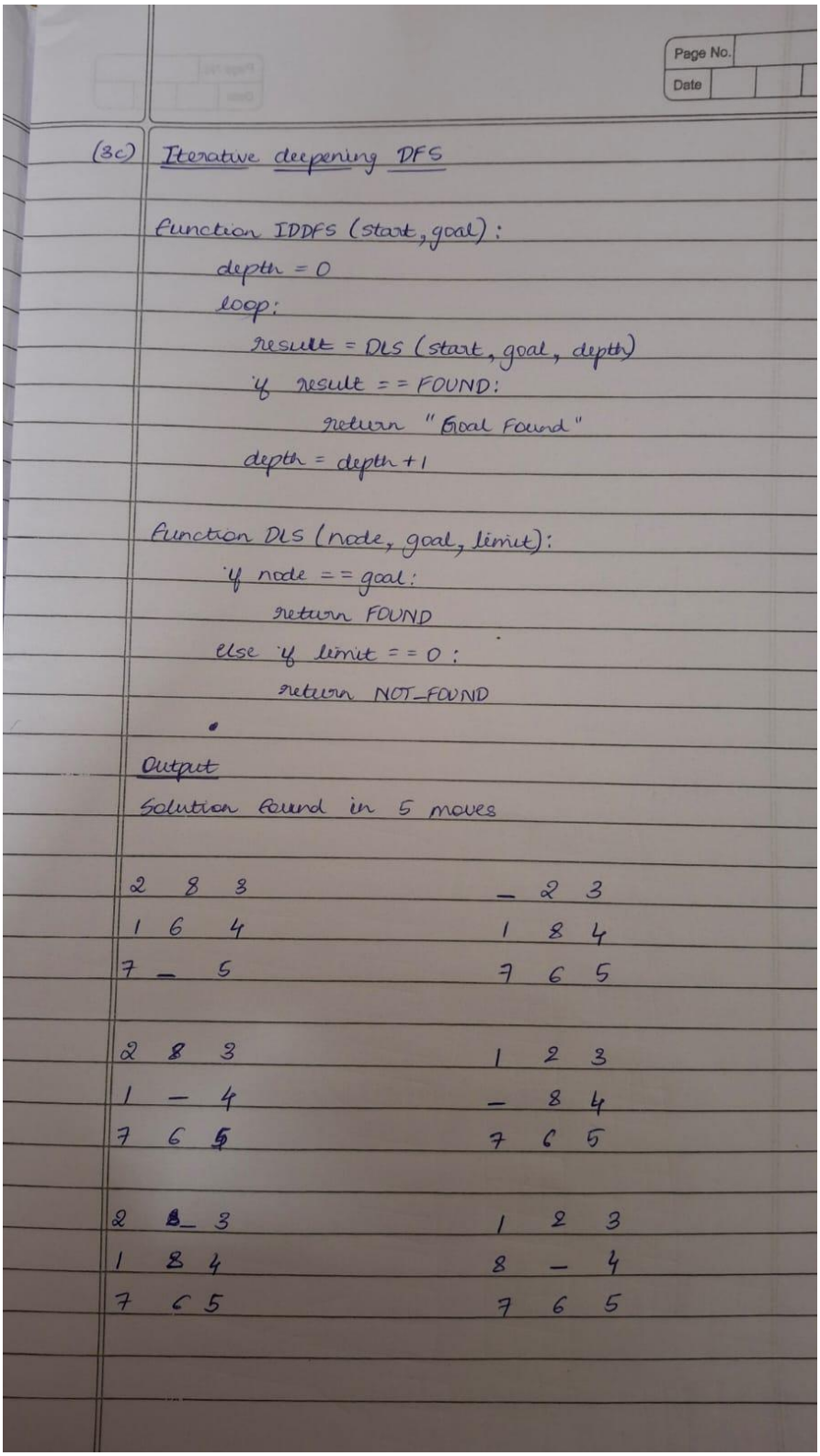
```
        print(state[i:i+3])
    print()
if __name__ == "__main__":
    start = (1, 2, 3,
             4, 0, 6,
             7, 5, 8)
    print("Starting DFS 8-puzzle solver...\nInitial state:")
    print_state(start)
    print("Sinchana Hemanth (1BM23CS330)")
    solution = dfs(start, max_depth=20)

    if solution:
        print(f"Solution found in {len(solution)-1} moves:\n")
        for step in solution:
            print_state(step)
    else:
        print("No solution found or max depth exceeded.")
```

OUTPUT:

```
Starting DFS 8-puzzle solver...
Initial state:
(1, 2, 3)
(4, 0, 6)
(7, 5, 8)

Sinchana Hemanth (1BM23CS330)
Solution found in 2 moves:

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

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

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

Using Iterative Deepening DFS solve 8 puzzle without heuristic approach.

ALGORITHM:



(3c) Iterative deepening DFS

```
function IDDFS (start, goal):
    depth = 0
    loop:
        result = DLS (start, goal, depth)
        if result == FOUND:
            return "Goal Found"
        depth = depth + 1

function DLS (node, goal, limit):
    if node == goal:
        return FOUND
    else if limit == 0 :
        return NOT_FOUND
```

Output

Solution found in 5 moves

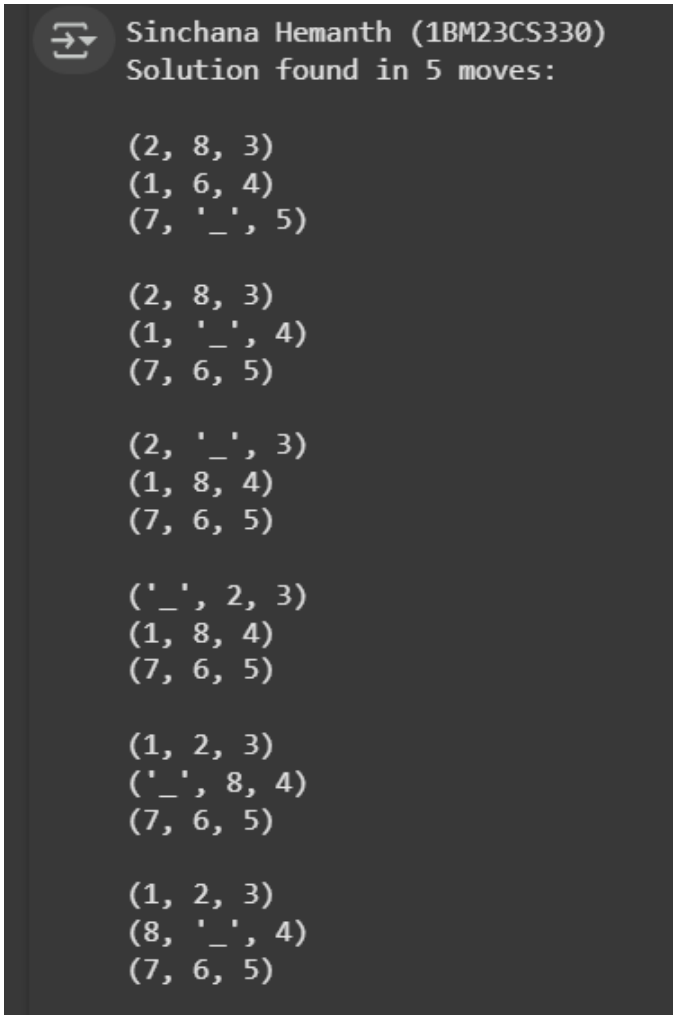
2 8 3 1 6 4 7 _ 5	_ 2 3 1 8 4 7 6 5
2 8 3 1 _ 4 7 6 5	1 2 3 _ 8 4 7 6 5
2 _ 3 1 8 4 7 6 5	1 2 3 8 _ 4 7 6 5

CODE:

```
from collections import deque
N = 3
moves = [(-1,0),(1,0),(0,-1),(0,1)]

def find_blank(state):
    idx = state.index("_")
    return divmod(idx, N)

def swap(state, i1, j1, i2, j2):
    s = list(state)
    idx1, idx2 = i1*N+j1, i2*N+j2
    s[idx1], s[idx2] = s[idx2], s[idx1]
    return tuple(s)

def expand(state):
    x, y = find_blank(state)
    children = []
    for dx, dy in moves:
        nx, ny = x+dx, y+dy
        if 0 <= nx < N and 0 <= ny < N:
            children.append(swap(state, x, y, nx, ny))
    return children

def dls(state, goal, limit, path, visited):
    if state == goal:
        return path
    if limit == 0:
        return None
    visited.add(state)
    for child in expand(state):
        if child not in visited:
            result = dls(child, goal, limit-1, path+[child], visited)
            if result is not None:
                return result
    visited.remove(state)
    return None

def iddfs(start, goal, max_depth=20):
    for depth in range(max_depth):
        visited = set()
        result = dls(start, goal, depth, [start], visited)
        if result is not None:
            return result
    return None
print("Sinchana Hemanth (1BM23CS330)")
initial = (2,8,3,1,6,4,7,"_",5)
goal    = (1,2,3,8,"_",4,7,6,5)
solution = iddfs(initial, goal, max_depth=30)
if solution:
```

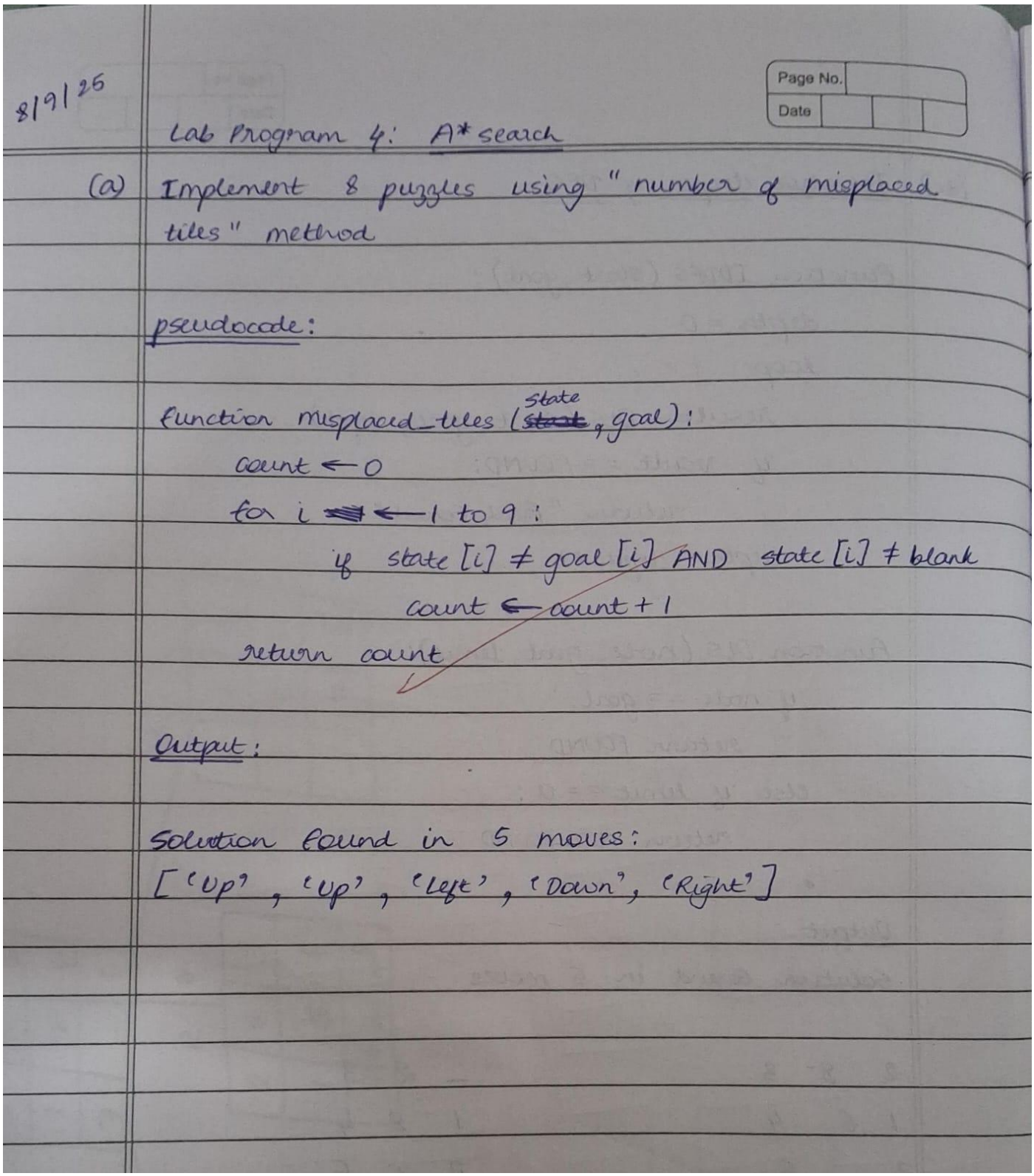
```
    print("Solution found in", len(solution)-1, "moves:\n")
    for step in solution:
        for i in range(0, 9, 3):
            print(step[i:i+3])
        print()
else:
    print("No solution found within depth limit")
```

OUTPUT:



Program 3
Apply A* algorithm for misplaced tiles.

ALGORITHM:



8/9/25

Page No.
Date

Lab Program 4: A* search

(a) Implement 8 puzzles using "number of misplaced tiles" method

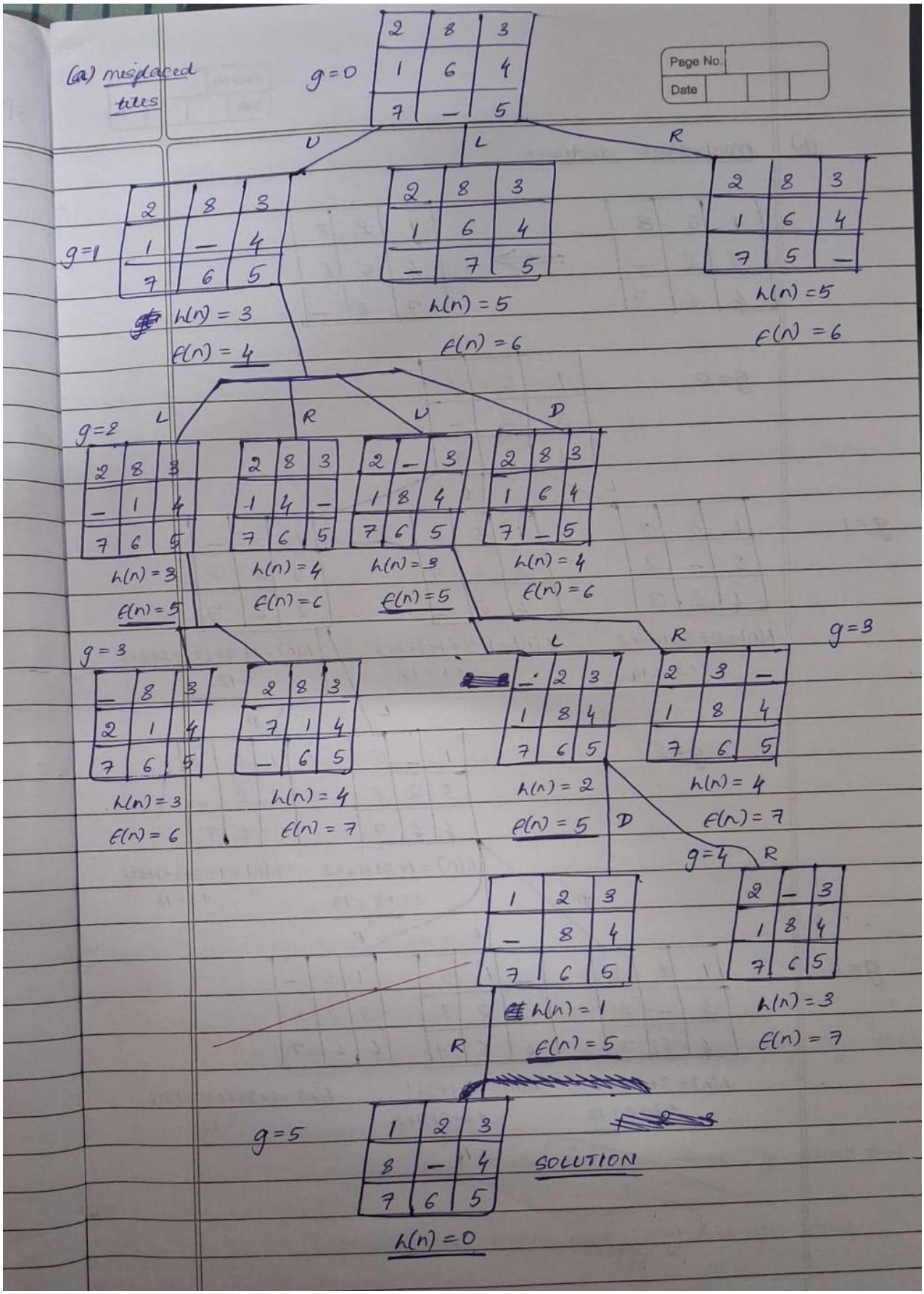
pseudocode:

```
Function misplaced_tiles (state, goal):
    count ← 0
    for i ← 1 to 9:
        if state[i] ≠ goal[i] AND state[i] ≠ blank
            count ← count + 1
    return count
```

Output:

Solution found in 5 moves:
['Up', 'Up', 'Left', 'Down', 'Right']

(a) misplaced tiles



CODE:

```
print("Sinchana Hemanth 1BM23CS330")
import heapq

class PuzzleState:
    def __init__(self, board, goal, parent=None, g=0):
        self.board = board
        self.goal = goal
        self.parent = parent
        self.g = g
        self.h = self.misplaced_tiles()
        self.f = self.g + self.h

    def misplaced_tiles(self):
        """Count misplaced tiles (excluding 0)."""
        return sum(1 for i in range(9) if self.board[i] != 0 and self.board[i] != self.goal[i])

    def get_neighbors(self):
        """Generate possible moves by sliding the blank (0)."""
        neighbors = []
        idx = self.board.index(0)
        x, y = divmod(idx, 3)   # row, col
        moves = [(-1,0),(1,0),(0,-1),(0,1)]  # up, down, left, right

        for dx, dy in moves:
            nx, ny = x+dx, y+dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                new_idx = nx*3 + ny
                new_board = self.board[:]
                new_board[idx], new_board[new_idx] = new_board[new_idx], new_board[idx]
                neighbors.append(PuzzleState(new_board, self.goal, self, self.g+1))
        return neighbors

    def __lt__(self, other):
        return self.f < other.f  # priority queue uses f value

def reconstruct_path(state):
    path = []
    while state:
        path.append(state.board)
        state = state.parent
    return path[::-1]

def astar(start, goal):
    start_state = PuzzleState(start, goal)
    open_list = []
    heapq.heappush(open_list, start_state)
    closed_set = set()

    while open_list:
```

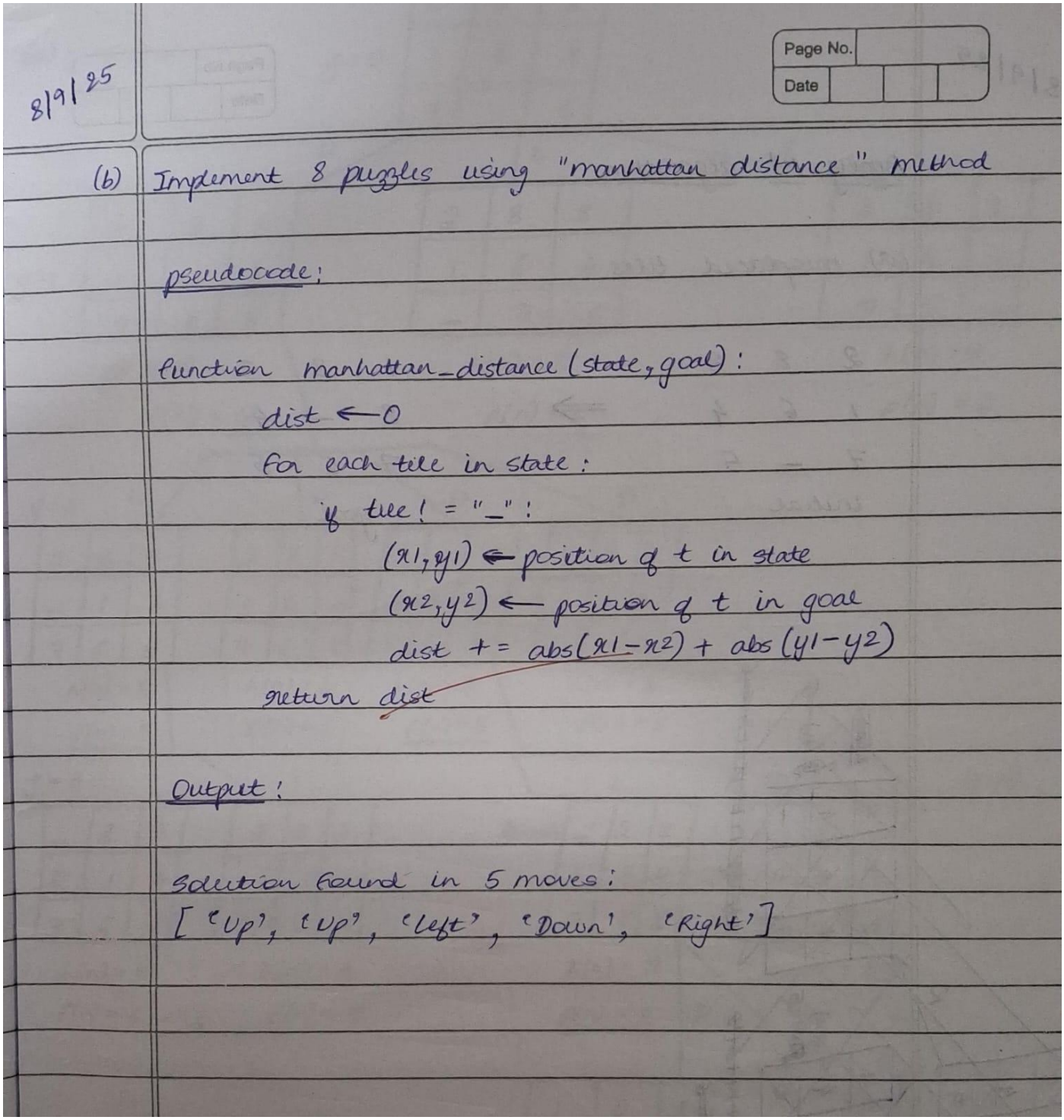
```
        current = heapq.heappop(open_list)

        if current.board == goal:
            return reconstruct_path(current)

        closed_set.add(tuple(current.board))

        for neighbor in current.get_neighbors():
            if tuple(neighbor.board) in closed_set:
                continue
            heapq.heappush(open_list, neighbor)
    return None

print("Enter the 8-puzzle START state (use 0 for blank).")
start_input = list(map(int, input("Enter 9 numbers separated by spaces: ").split()))

print("\nEnter the GOAL state (use 0 for blank).")
goal_input = list(map(int, input("Enter 9 numbers separated by spaces: ").split()))

if len(start_input) != 9 or len(goal_input) != 9:
    print("Invalid input! Please enter exactly 9 numbers for each state.")
else:
    solution = astar(start_input, goal_input)

    if solution:
        print("\n Steps to solve:")
        for step in solution:
            for i in range(0,9,3):
                print(step[i:i+3])
            print("------")
    else:
        print(" No solution found!")
```

OUTPUT:

```
Sinchana Hemanth 1BM23CS330
Enter the 8-puzzle START state (use 0 for blank).
Enter 9 numbers separated by spaces: 2 8 3 1 6 4 7 0 5

Enter the GOAL state (use 0 for blank).
Enter 9 numbers separated by spaces: 1 2 3 8 0 4 7 6 5

 Steps to solve:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
------
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
------
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
------
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
------
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
------
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
------
```

Apply A* algorithm for Manhattan Distance.

ALGORITHM:



8/9/25

Page No.
Date

(b) Implement 8 puzzles using "manhattan distance" method

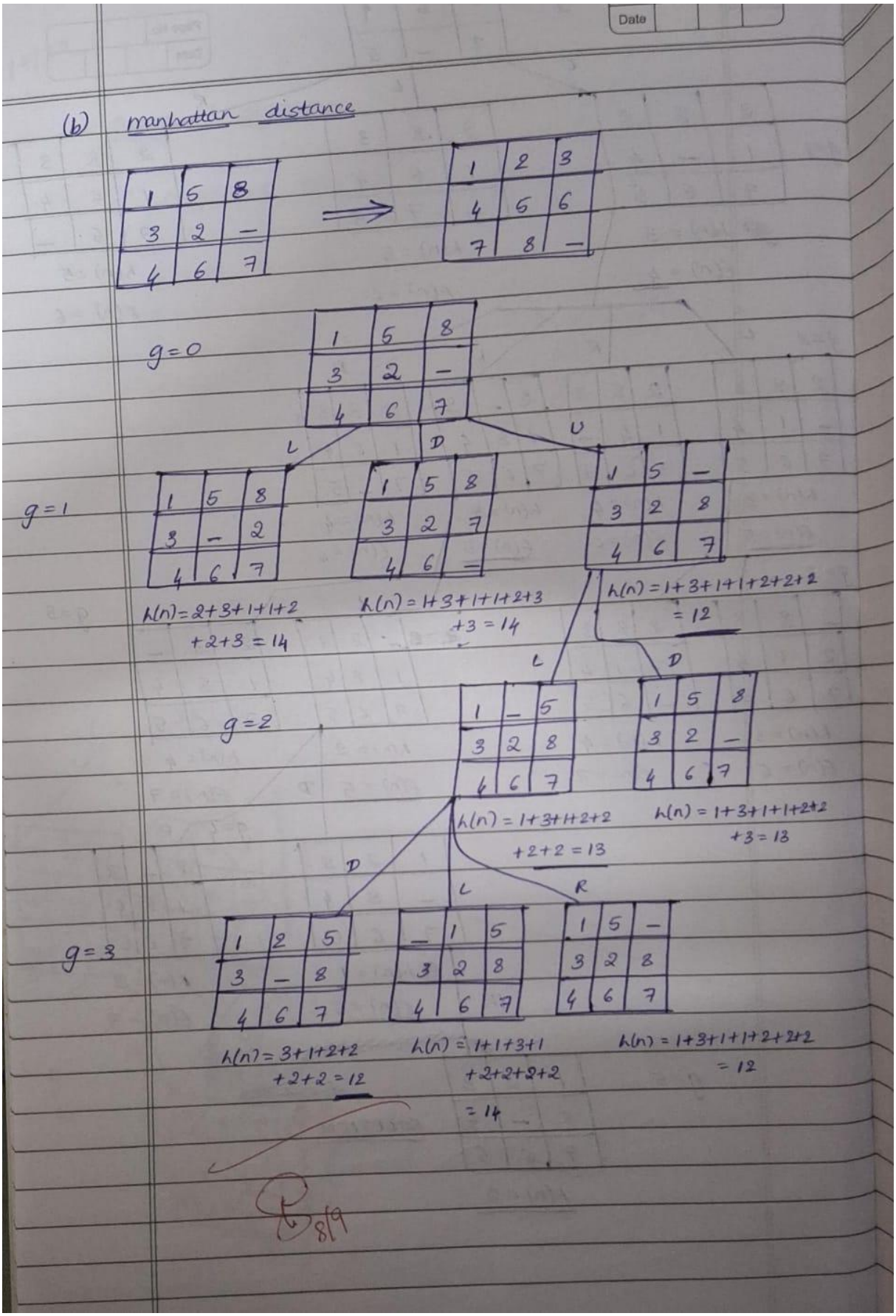
pseudocode:

```
function manhattan_distance(state, goal):
    dist ← 0
    for each tile in state:
        if tile != "_":
            (x1, y1) ← position of t in state
            (x2, y2) ← position of t in goal
            dist += abs(x1 - x2) + abs(y1 - y2)
    return dist
```

Output:

Solution found in 5 moves:
['Up', 'Up', 'Left', 'Down', 'Right']

(b) Manhattan distance



CODE:

```
import heapq
class PuzzleState:
    def __init__(self, board, goal, parent=None, g=0):
        self.board = board
        self.goal = goal
        self.parent = parent
        self.g = g
        self.h = self.manhattan_distance()
        self.f = self.g + self.h

    def manhattan_distance(self):
        """Heuristic: Manhattan Distance."""
        distance = 0
        for i, tile in enumerate(self.board):
            if tile != 0:
                goal_index = self.goal.index(tile)
                x1, y1 = divmod(i, 3)
                x2, y2 = divmod(goal_index, 3)
                distance += abs(x1 - x2) + abs(y1 - y2)
        return distance

    def get_neighbors(self):
        """Generate possible moves by sliding the blank (0)."""
        neighbors = []
        idx = self.board.index(0)
        x, y = divmod(idx, 3)
        moves = [(-1,0),(1,0),(0,-1),(0,1)]
        for dx, dy in moves:
            nx, ny = x + dx, y + dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                new_idx = nx * 3 + ny
                new_board = self.board[:]
                new_board[idx], new_board[new_idx] = new_board[new_idx], new_board[idx]
                neighbors.append(PuzzleState(new_board, self.goal, self, self.g + 1))
        return neighbors

    def __lt__(self, other):
        return self.f < other.f

def reconstruct_path(state):
    path = []
    while state:
        path.append(state.board)
        state = state.parent
    return path[::-1]

def astar(start, goal):
    start_state = PuzzleState(start, goal)
    open_list = []
    heapq.heappush(open_list, start_state)
```

```
    closed_set = set()
    while open_list:
        current = heapq.heappop(open_list)
        if current.board == goal:
            return reconstruct_path(current)
        closed_set.add(tuple(current.board))
        for neighbor in current.get_neighbors():
            if tuple(neighbor.board) in closed_set:
                continue
            heapq.heappush(open_list, neighbor)
    return None
print("Sinchana Hemanth 1BM23CS330")
print("Enter the 8-puzzle START state (use 0 for blank).")
start_input = list(map(int, input("Enter 9 numbers separated by spaces: ").split()))

print("\nEnter the GOAL state (use 0 for blank).")
goal_input = list(map(int, input("Enter 9 numbers separated by spaces: ").split()))
if len(start_input) != 9 or len(goal_input) != 9:
    print("Invalid input! Please enter exactly 9 numbers for each state.")
else:
    solution = astar(start_input, goal_input)

    if solution:
        print("\nSteps to solve:")
        for step in solution:
            for i in range(0, 9, 3):
                print(step[i:i+3])
            print("------")
        print(f"Total moves: {len(solution)-1}")
    else:
        print("No solution found!")
```

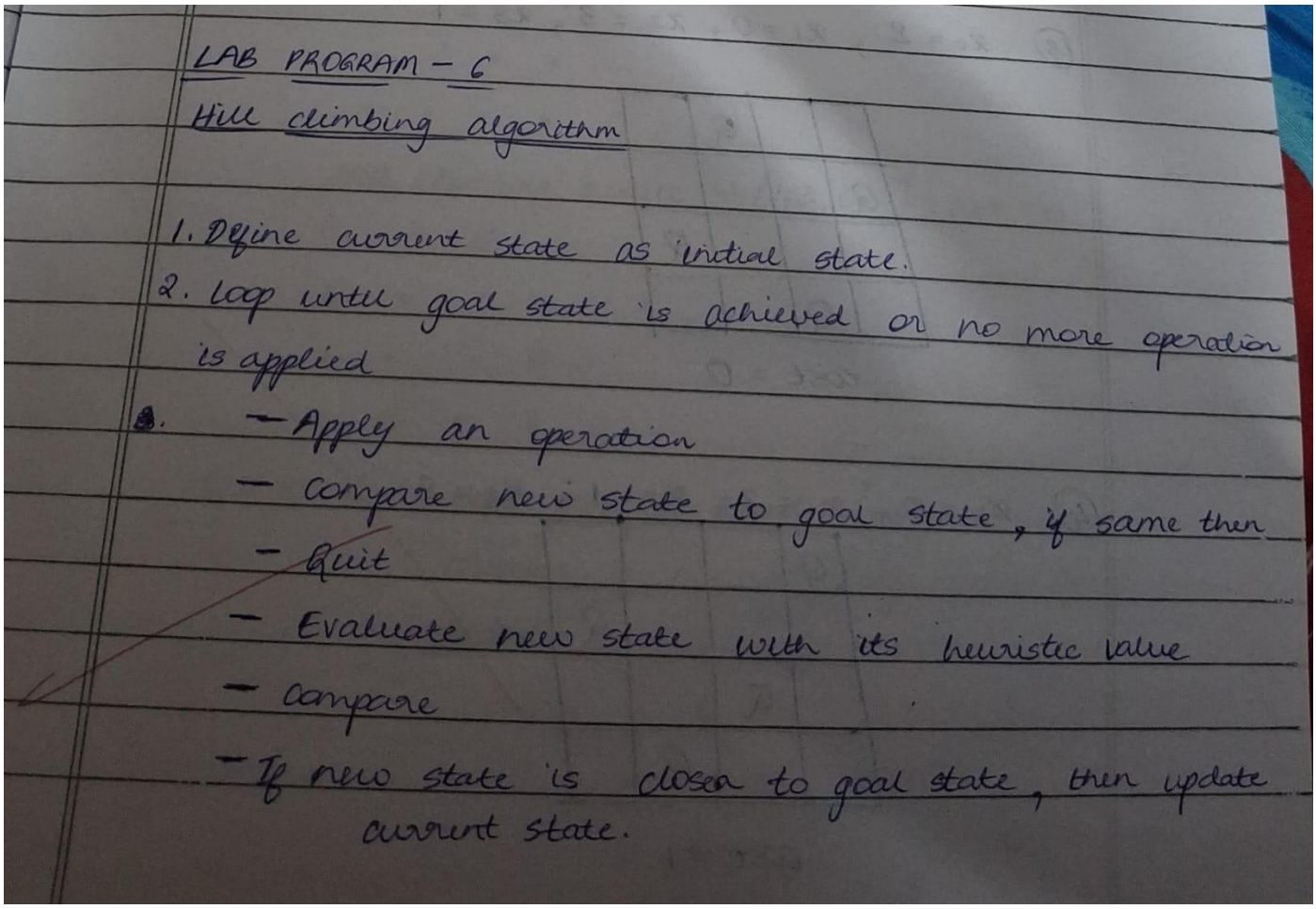
OUTPUT:

```
Sinchana Hemanth 1BM23CS330
Enter the 8-puzzle START state (use 0 for blank).
Enter 9 numbers separated by spaces: 2 8 3 1 6 4 7 0 5

Enter the GOAL state (use 0 for blank).
Enter 9 numbers separated by spaces: 1 2 3 8 0 4 7 6 5

Steps to solve:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
------
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
------
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
------
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
------
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
------
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
------
Total moves: 5
```

Program 4

Implement hill climbing search algorithm to solve N-Queens problem.

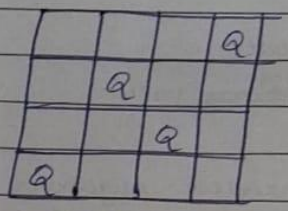
ALGORITHM:



Hill climbing algorithm for 4-queens

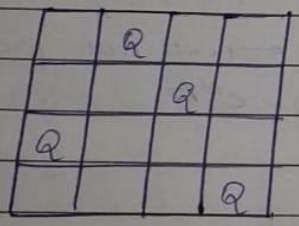
initial state;

① $x_0 = 3, x_1 = 1, x_2 = 2, x_3 = 0$



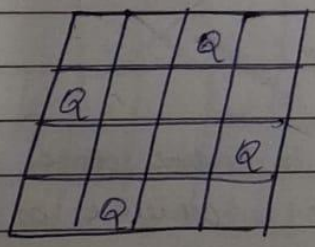
cost = 1+1 = 2

② $x_0 = 1, x_1 = 2, x_2 = 0, x_3 = 3$



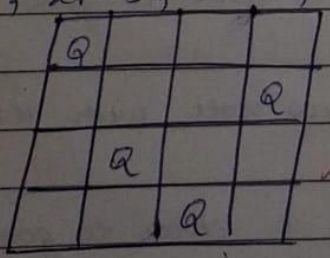
cost = 1

③ $x_0 = 2, x_1 = 0, x_2 = 3, x_3 = 1$



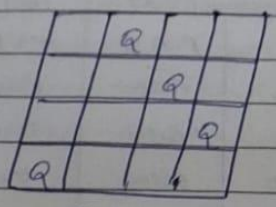
cost = 0

④ $x_0 = 0, x_1 = 3, x_2 = 1, x_3 = 2$



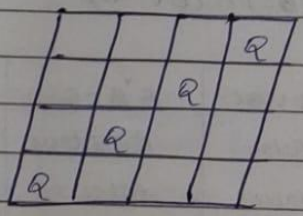
cost = 1

⑤ $x_0 = 1, x_1 = 2, x_2 = 3, x_3 = 0$



cost = 1+1+1+1 = 4

⑥ $x_0 = 3, x_1 = 2, x_2 = 1, x_3 = 0$



cost = 1+1+1+1+1+1 = 6

Output 1 :

Initial State: (0,0,0,0), cost = 6

Q Q Q Q
. . . .
. . . .
. . . .

Hill climbing result : [1,3,0,2], cost = 0

. . Q .
Q . . .
. . . Q
. Q . .

Output 2 : Best position found: [5,1,6,0,2,4,7,3]

Number of non-attacking pairs: 28

Board:

. . . Q . . . .
. . . . Q . . .
. Q . . . . . .
. . . . . Q . .
. . . . Q
. . . . . Q . .
Q . . . .

CODE:

```
import copy
import random

def print_board(state):
    n = len(state)
    for row in range(n):
        line = ""
        for col in range(n):
            line += "Q " if state[col] == row else ". "
        print(line)
    print()

def heuristic(state):
    attacks = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j]:
                attacks += 1
            if abs(state[i] - state[j]) == abs(i - j):
                attacks += 1
    return attacks

def get_neighbors(state):
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if state[col] != row:
                new_state = list(state)
                new_state[col] = row
                neighbors.append(new_state)
    return neighbors

def hill_climbing(start_state):
    current = copy.deepcopy(start_state)
    while True:
        current_h = heuristic(current)
        if current_h == 0:
            return current, 0

        neighbors = get_neighbors(current)
        neighbor_h = [heuristic(neigh) for neigh in neighbors]

        min_h = min(neighbor_h)
        if min_h >= current_h:
            return current, current_h

        current = neighbors[neighbor_h.index(min_h)]
```

```
def hill_climbing_with_restarts(n, max_restarts=100):
    for attempt in range(1, max_restarts + 1):
        start_state = [random.randint(0, n - 1) for _ in range(n)]
        final_state, cost = hill_climbing(start_state)
        print(f"Attempt {attempt}:")
        print(f"Start state: {start_state}")
        print(f"Final state (heuristic={cost}):")
        print_board(final_state)
        if cost == 0:
            print(f"Solution found after {attempt} attempts!\n")
            return final_state
    print("No solution found after all restarts.")
    return None

if __name__ == "__main__":
    n = 4
    print("Sinchana Hemanth 1BM23CS330")
    print(f"Hill Climbing for {n}-Queens Problem\n")
    solution = hill_climbing_with_restarts(n, max_restarts=100)
    if solution:
        print("Final Solution Board:")
        print_board(solution)
```

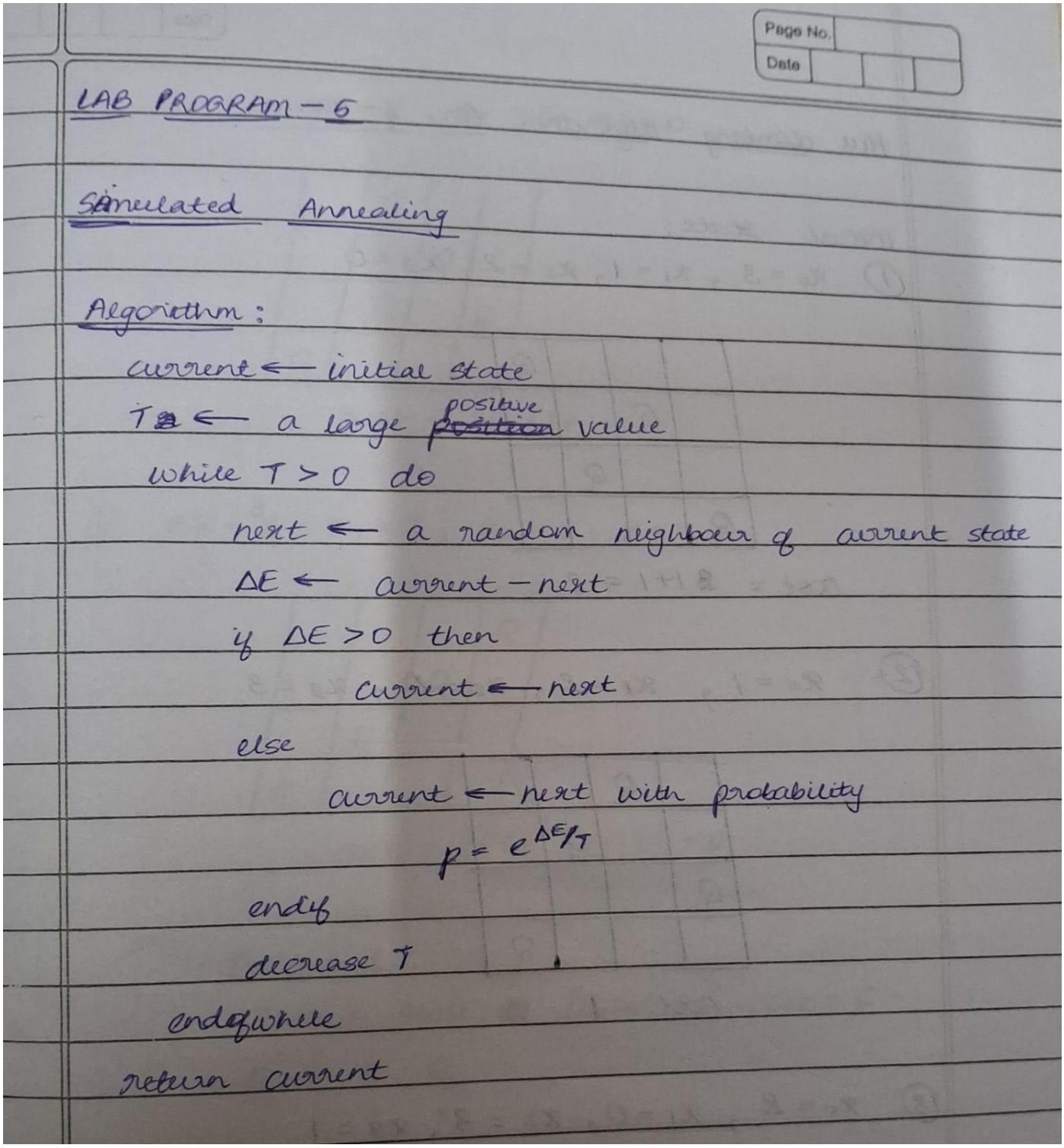
OUTPUT:

```
Sinchana Hemanth 1BM23CS330
Hill Climbing for 4-Queens Problem

Attempt 1:
Start state: [0, 2, 1, 1]
Final state (heuristic=1):
Q . . .
. . . Q
. Q . .
. . Q .

Attempt 2:
Start state: [1, 3, 3, 0]
Final state (heuristic=0):
. . Q .
Q . . .
. . . Q
. Q . .

✅ Solution found after 2 attempts!

Final Solution Board:
. . Q .
Q . . .
. . . Q
. Q . .
```

Program 5

8 Queens Problem using Simulated Annealing

ALGORITHM:



LAB PROGRAM – 5

Simulated Annealing

Algorithm :

current ← initial state

T ← a large positive value

while T > 0 do

next ← a random neighbour of current state

ΔE ← current − next

if ΔE > 0 then

current ← next

else

current ← next with probability

$p = e^{\Delta E/T}$

endif

decrease T

endofwhile

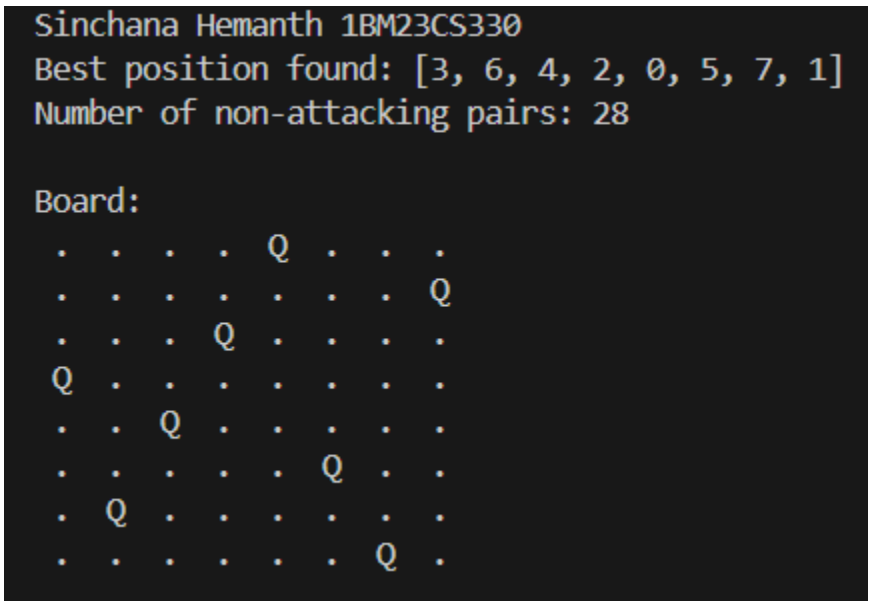
return current

CODE:

```
import random
import math
print("Sinchana Hemanth 1BM23CS330")
def cost(state):
    attacks = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                attacks += 1
    return attacks

def get_neighbor(state):
    neighbor = state[:]
    i, j = random.sample(range(len(state)), 2)
    neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
    return neighbor

def simulated_annealing(n=8, max_iter=10000):
    current = list(range(n))
    random.shuffle(current)
    current_cost = cost(current)

    temperature = 100.0
    cooling_rate = 0.95

    best = current[:]
    best_cost = current_cost

    for _ in range(max_iter):
        if temperature <= 0 or best_cost == 0:
            break

        neighbor = get_neighbor(current)
        neighbor_cost = cost(neighbor)
        delta = current_cost - neighbor_cost

        if delta > 0:
            current, current_cost = neighbor, neighbor_cost
            if neighbor_cost < best_cost:
                best, best_cost = neighbor, neighbor_cost
        else:
            probability = math.exp(delta / temperature)
            if random.random() < probability:
                current, current_cost = neighbor, neighbor_cost

        temperature *= cooling_rate

    return best, best_cost
```

```
def print_board(state):
    n = len(state)
    for row in range(n):
        line = ""
        for col in range(n):
            if state[col] == row:
                line += " Q "
            else:
                line += " . "
        print(line)
    print()

if __name__ == "__main__":
    n = 8
    solution, cost_val = simulated_annealing(n)

    print("Best position found:", solution)
    print(f"Number of non-attacking pairs: {n*(n-1)//2 - cost_val}")
    print("\nBoard:")
    print_board(solution)
```

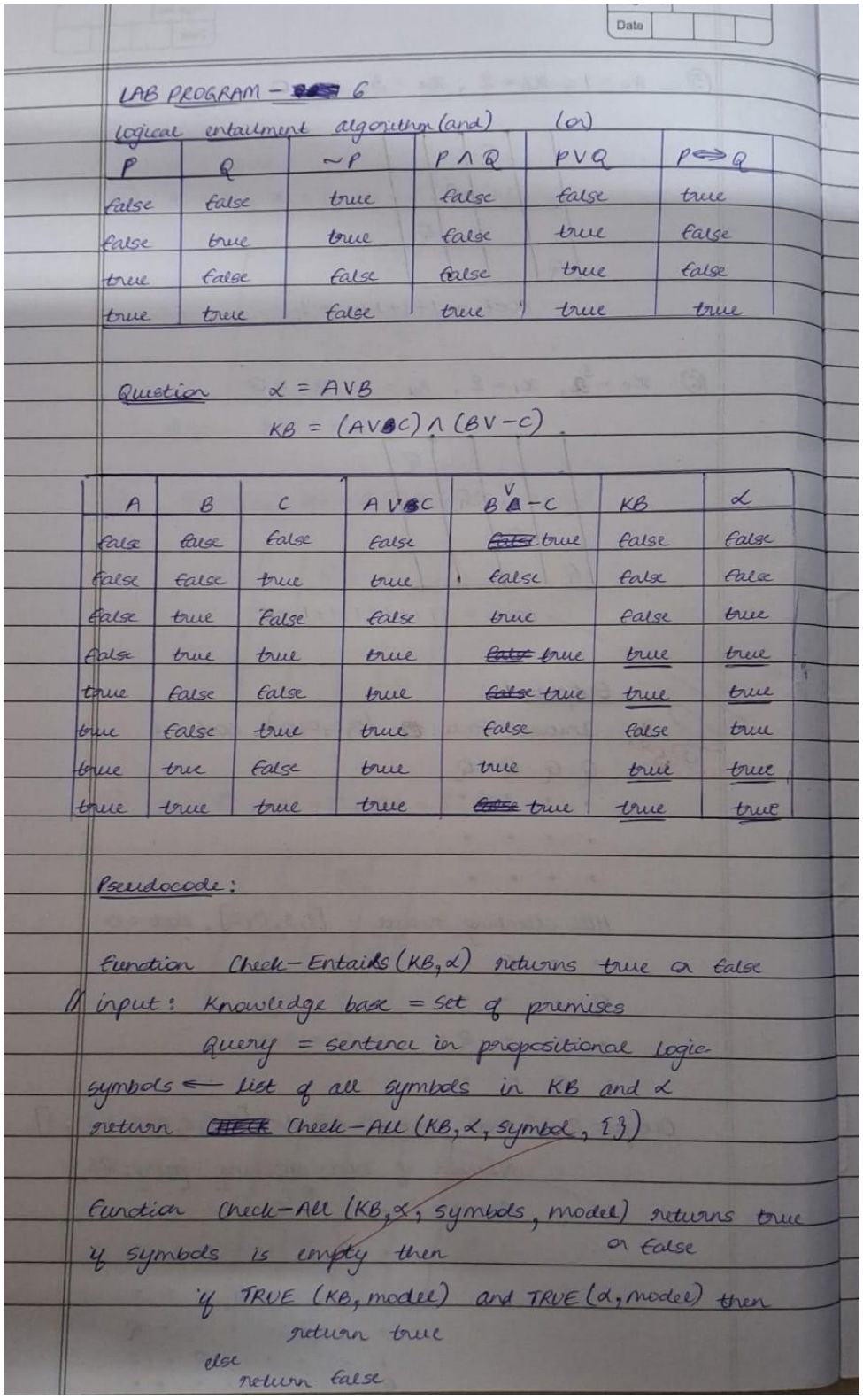
OUTPUT:



Program 6

Implement truth table enumeration algorithm for deciding propositional entailment.

ALGORITHM:



Date

LAB PROGRAM – 6

Logical entailment algorithm (and) (or)

P	Q	~P	P ∧ Q	P ∨ Q	P ⇔ Q
false	false	true	false	false	true
false	true	true	false	true	false
true	false	false	false	true	false
true	true	false	true	true	true

Question $\alpha = A \vee B$

$KB = (A \vee C) \wedge (B \vee \neg C)$

A	B	C	A ∨ C	B ∨ ¬C	KB	α
false	false	false	false	true	false	false
false	false	true	true	false	false	false
false	true	false	false	true	false	true
false	true	true	true	true	true	true
true	false	false	true	true	true	true
true	false	true	true	false	false	true
true	true	false	true	true	true	true
true	true	true	true	true	true	true

Pseudocode:

Function Check-Entails (KB, α) returns true or false

// input: Knowledge base = set of premises

Query = sentence in propositional logic

symbols ← list of all symbols in KB and α

return Check-All (KB, α, symbol, {})

Function Check-All (KB, α, symbols, model) returns true or false

if symbols is empty then

if TRUE (KB, model) and TRUE (α, model) then

return true

else

return false

else:

P ← first symbol in symbols

rest ← remaining sym

return (Check-All(KB, α, rest, model ∪ {P = true})) and (Check-All(KB, α, rest, model ∪ {P = false}))

consider S ET as variables and following

a ← ~(S ∨ T)

b ← (S ∧ T)

c ← T ∨ ~T

Write truth table and show whether:

(i) a entails b

(ii) a entails c

S	T	a: ~(S∨T)	b: S∧T	c: T ∨ ~T
false	false	true	false	true
false	true	false	false	true
true	false	false	false	true
true	true	false	true	true

a entails b: false

a entails c: true

22/09

CODE:

```
import pandas as pd
from itertools import product
import re

def tokenize(sentence):
    token_pattern = r'\w+|[()∨∧¬]'
    return re.findall(token_pattern, sentence)

def pl_true(sentence, model):
    tokens = tokenize(sentence)
    logical_ops = {'and', 'or', 'not', '∨', '∧', '¬'}
    evaluated_tokens = []
    for token in tokens:
        if token == '∨':
            evaluated_tokens.append('or')  # replace symbol with python 'or'
        elif token == '∧':
            evaluated_tokens.append('and')  # replace symbol with python 'and'
        elif token == '¬':
            evaluated_tokens.append('not')  # replace symbol with python 'not'
        elif token.lower() in logical_ops:
            evaluated_tokens.append(token.lower())
        elif token in model:
            evaluated_tokens.append(str(model[token]))
        else:
            evaluated_tokens.append(token)
    eval_sentence = ' '.join(evaluated_tokens)
    try:
        return eval(eval_sentence)
    except Exception as e:
        print(f"Error evaluating sentence: {eval_sentence}")
        raise e

def tt_entails(kb, alpha, symbols):
    truth_table = []
    for model in product([False, True], repeat=len(symbols)):
        model_dict = dict(zip(symbols, model))
        kb_value = pl_true(kb, model_dict)
        alpha_value = pl_true(alpha, model_dict)
        row = {
            'A': model_dict.get('A', False),
            'B': model_dict.get('B', False),
            'C': model_dict.get('C', False),
            'A ∨ C': model_dict.get('A', False) or model_dict.get('C', False),
            'B ∨ ¬C': model_dict.get('B', False) or not model_dict.get('C', False),
            'KB': kb_value,
            'α': alpha_value
        }
        truth_table.append(row)
        if kb_value and not alpha_value:
            return False, pd.DataFrame(truth_table)
```

```
    return True, pd.DataFrame(truth_table)

def get_symbols(kb, alpha):
    return sorted(set(re.findall(r'[A-Z]', kb + alpha)))

kb = "(A ∨ C) ∧ (B ∨ ¬C)"
alpha = "A ∨ B"
symbols = get_symbols(kb, alpha)
result, truth_table = tt_entails(kb, alpha, symbols)

def highlight_kb_alpha(row):
    if row['KB'] and row['α']:
        return ['background-color: lightgreen' if col in ['KB', 'α'] else '' for col in row.index]
    else:
        return ['' for _ in row.index]

print("Sinchana Hemanth 1BM23CS330")
styled_table = truth_table.style.apply(highlight_kb_alpha, axis=1)
display(styled_table)

if result:
    print("\nKB entails α")
else:
    print("\nKB does not entail α")
```

OUTPUT:

Sinchana Hemanth 1BM23CS330

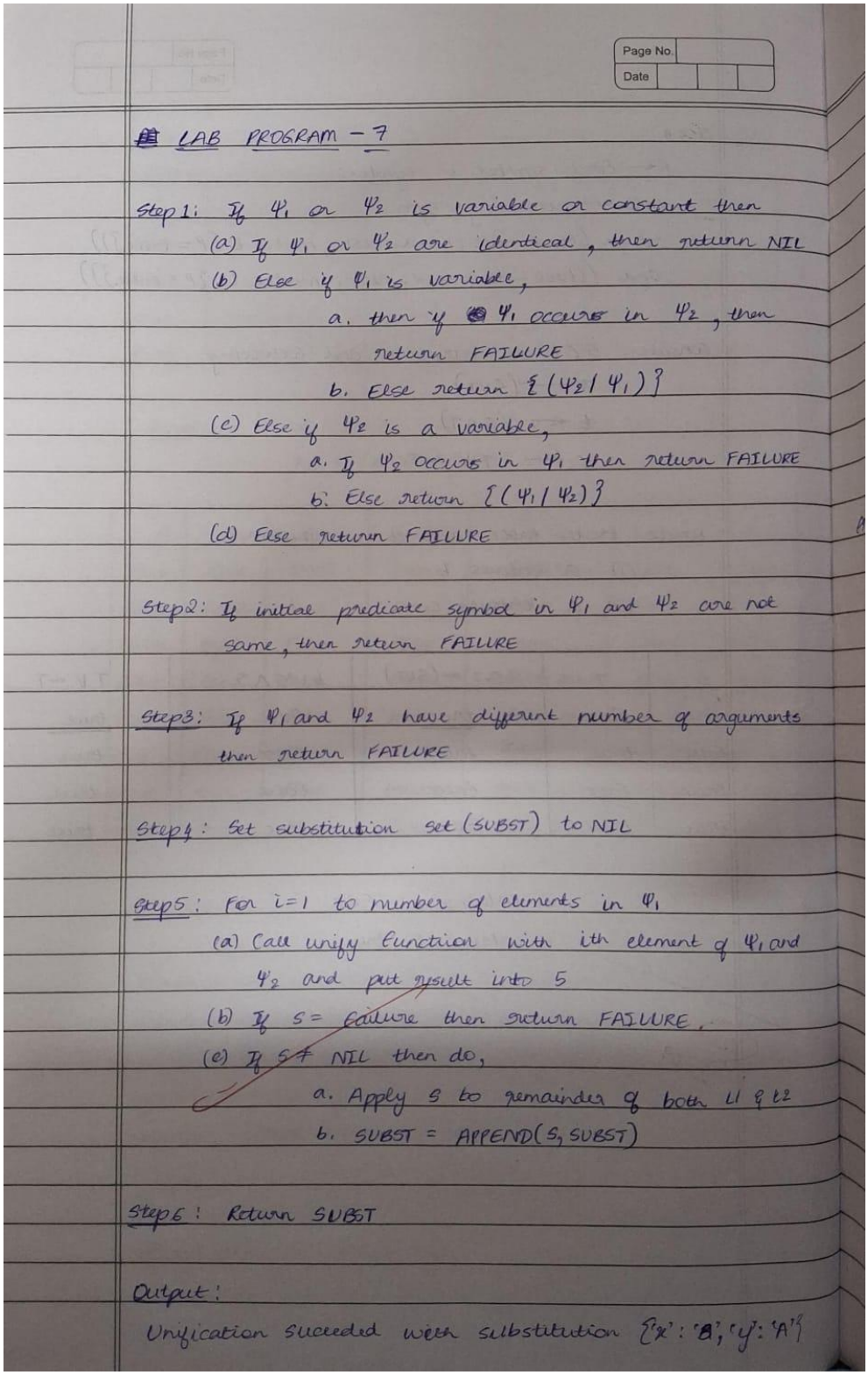
	A	B	C	A ∨ C	B ∨ ¬C	KB	α
0	False	False	False	False	True	False	False
1	False	False	True	True	False	False	False
2	False	True	False	False	True	False	True
3	False	True	True	True	True	True	True
4	True	False	False	True	True	True	True
5	True	False	True	True	False	False	True
6	True	True	False	True	True	True	True
7	True	True	True	True	True	True	True

KB entails α

Program 7

Implement unification in first order logic.

ALGORITHM:



LAB PROGRAM – 7

Step 1: If Ψ_1 or Ψ_2 is variable or constant then
(a) If Ψ_1 or Ψ_2 are identical, then return NIL
(b) Else if Ψ_1 is variable,
a. then if Ψ_1 occurs in Ψ_2, then return FAILURE
b. Else return $\{(\Psi_2 / \Psi_1)\}$
(c) Else if Ψ_2 is a variable,
a. If Ψ_2 occurs in Ψ_1 then return FAILURE
b. Else return $\{(\Psi_1 / \Psi_2)\}$
(d) Else return FAILURE

Step2: If initial predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE

Step3: If Ψ_1 and Ψ_2 have different number of arguments then return FAILURE

Step4: Set substitution set (SUBST) to NIL

Step5: For i=1 to number of elements in Ψ_1
(a) Call unify function with ith element of Ψ_1 and Ψ_2 and put result into S
(b) If S = failure then return FAILURE,
(c) If S ≠ NIL then do,
a. Apply S to remainder of both L1 & L2
b. SUBST = APPEND(S, SUBST)

Step6: Return SUBST

Output:
Unification succeeded with substitution {'x': 'B', 'y': 'A'}

Page No.
Date

Solve the following

1. Find MGU of $\{p(b, x, f(g(z)))$ and $p(z, f(y), f(y))\}$

comparing both predicates

① $b \leftrightarrow z$
② $x \leftrightarrow f(y)$
③ $f(g(z)) \leftrightarrow f(y)$

substituting $b = z$ from ①
from ③ $f(g(b)) = f(y) \longrightarrow y = g(b)$

substituting $x = f(y)$ from ② after substituting
$x = f(g(b))$

unifiers $= \{b/z, x/f(y), y/g(z)\}$

② Find MGU of $\{Q(a, g(x,a), f(y))$ and $Q(a, g(f(b),a), x)\}$

2. $Q(a, g(x,a), f(y))$ & $Q(a, g(f(b),a), x)$

$x/f(b)$

$Q(a, g(f(b),a), f(y))$

$f(y)/x$

$Q(a, g(f(b),a), x)$

③ Find MGU of $\{p(f(a), g(y))$, $p(x,x)\}$ ~~Unify prime~~

3. $P(f(a), g(y))$ $P(x,x)$ ~~$= \{prime(11), prime(y)\}$~~

$x/f(a)$ } no unifier
$P(x, g(y))$ }

~~unifiers sub: $\{y/11\}$~~

$x/g(y)$

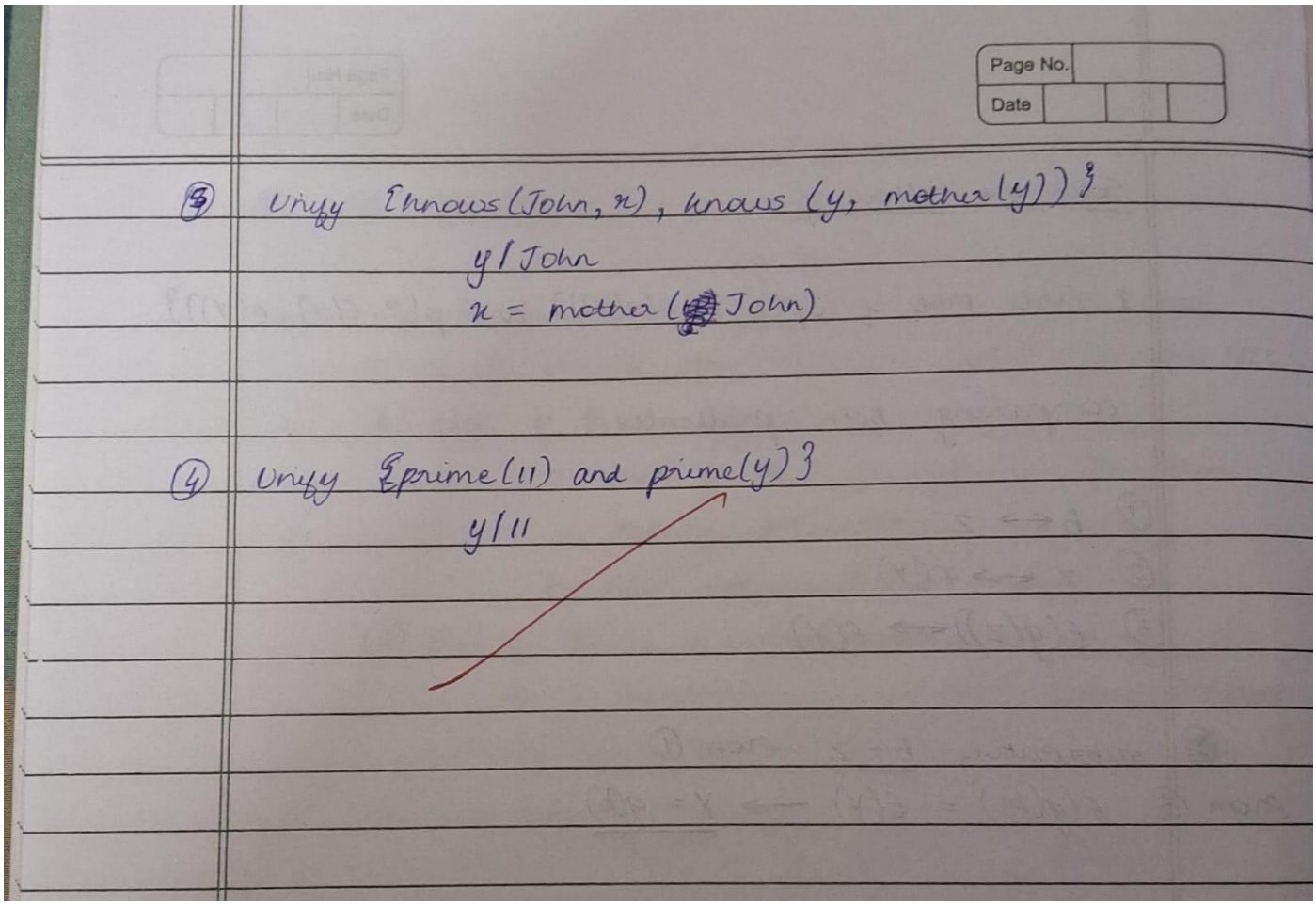
$P(x,x)$

⑤ Unify $\{knows(John, x)$, $knows(y, Bill)\}$

5. $\{knows(John, x)$, $knows(y, Bill)\}$

sub: John/y or y/John

x/Bill



CODE:

```
print('Sinchana Hemanth  1BM23CS330')
def unify(x, y, subst=None):
    if subst is None:
        subst = {}

    if is_variable(x) or is_constant(x):
        if x == y:
            return subst
        elif is_variable(x):
            return unify_var(x, y, subst)
        elif is_variable(y):
            return unify_var(y, x, subst)
        else:
            return None
    if is_compound(x) and is_compound(y):
        if x[0] != y[0] or len(x[1]) != len(y[1]):
            return None
        for xi, yi in zip(x[1], y[1]):
            subst = unify(xi, yi, subst)
            if subst is None:
                return None
```

```
        return subst
    return None

def is_variable(x):
    return isinstance(x, str) and x.islower() and x.isalpha()

def is_constant(x):
    return isinstance(x, str) and x.isupper() and x.isalpha()

def is_compound(x):
    return isinstance(x, tuple) and len(x) == 2 and isinstance(x[0], str) and isinstance(x[1], list)

def unify_var(var, x, subst):
    if var in subst:
        return unify(subst[var], x, subst)
    elif x in subst:
        return unify(var, subst[x], subst)
    elif occurs_check(var, x, subst):
        return None
    else:
        subst[var] = x
        return subst

def occurs_check(var, x, subst):
    if var == x:
        return True
    elif is_variable(x) and x in subst:
        return occurs_check(var, subst[x], subst)
    elif is_compound(x):
        return any(occurs_check(var, arg, subst) for arg in x[1])
    else:
        return False
x = ("P", ["x", "A"])
y = ("P", ["B", "y"])
result = unify(x, y)
if result is not None:
    print("Unification succeeded with substitution:", result)
else:
    print("Unification failed.")
```

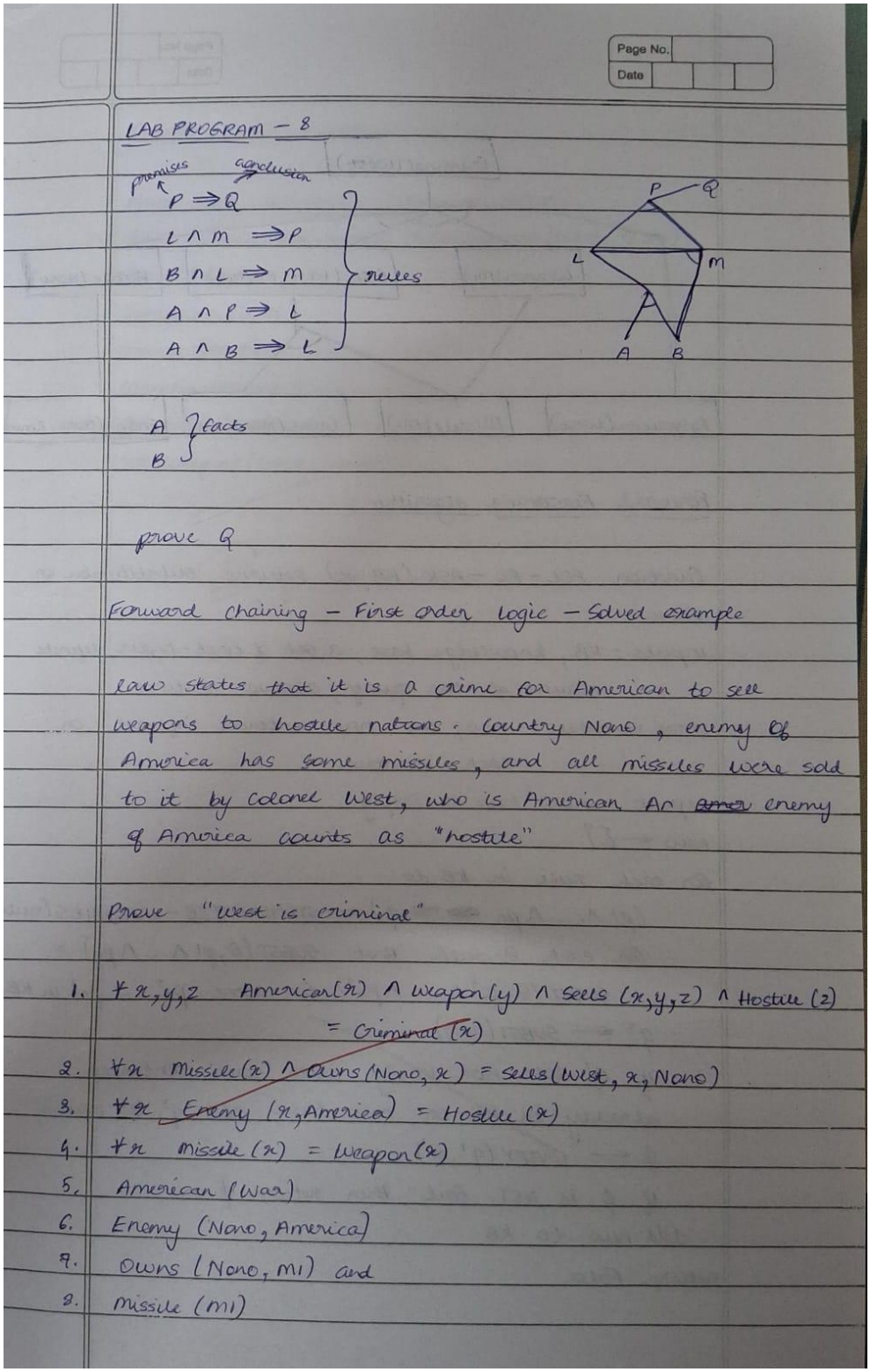
OUTPUT:

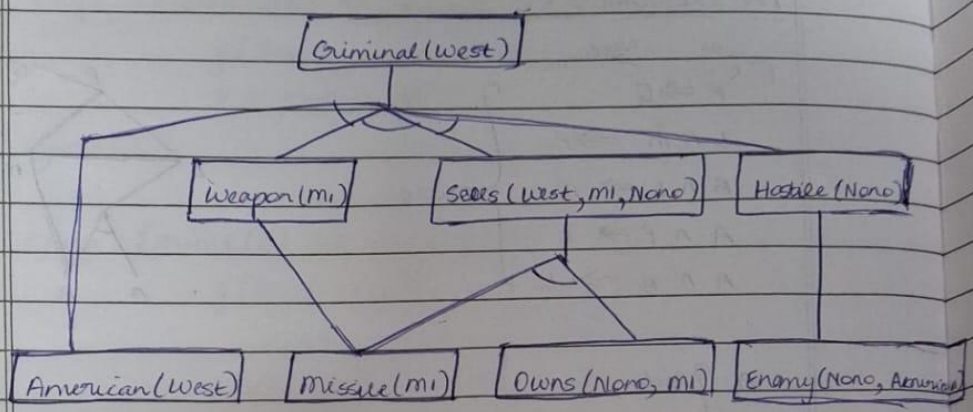
```
Sinchana Hemanth  1BM23CS330
Unification succeeded with substitution: {'x': 'B', 'y': 'A'}
```

Program 8

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

ALGORITHM:





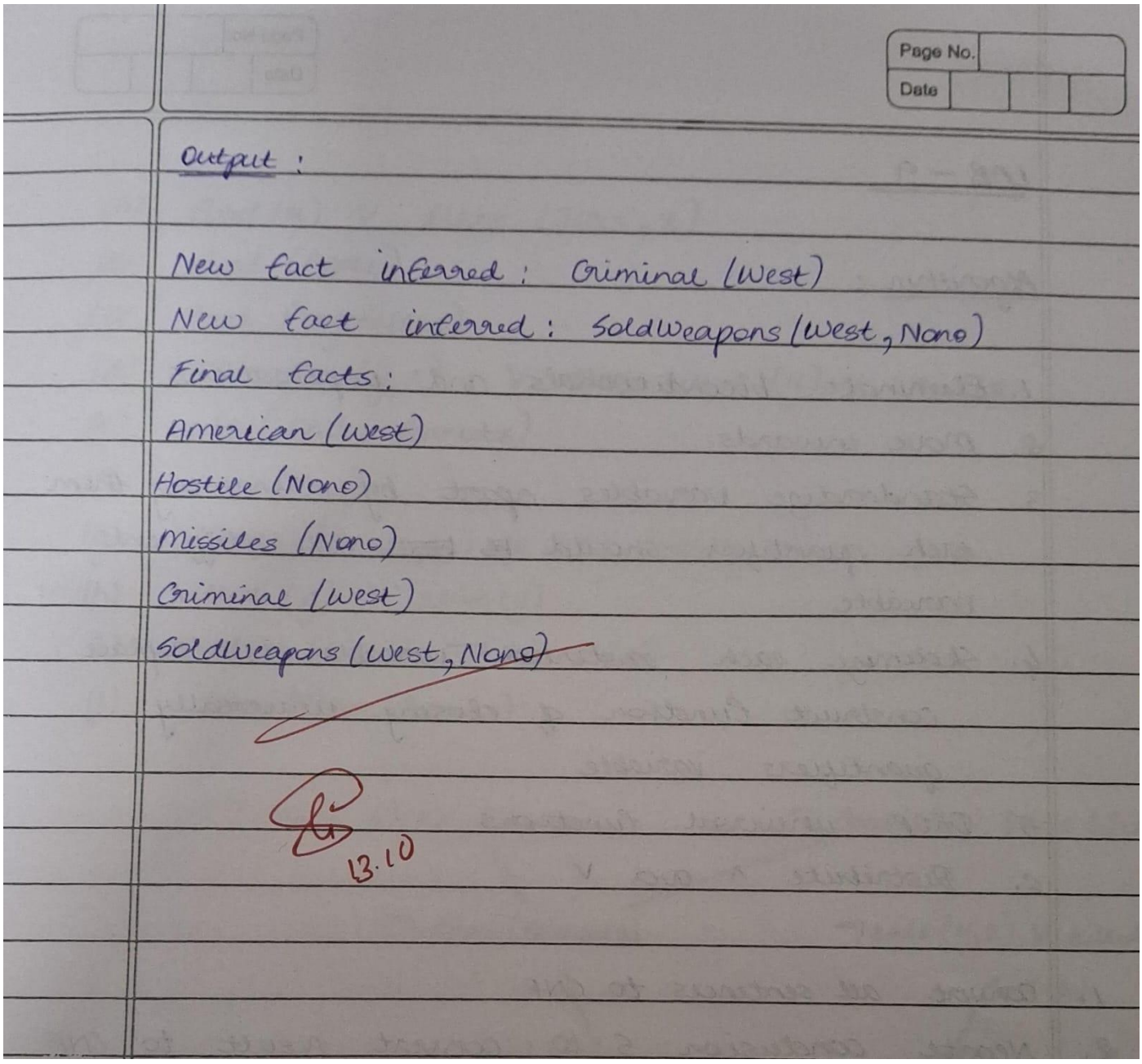
Forward Reasoning algorithm

function FOL-FC-ASK (KB, α) returns substitution or false

inputs: KB, knowledge base, a set of first-order definite clauses
α, the query, atomic sentence

local variables: new, the new sentences inferred on each iteration

repeat until new is empty
new $\leftarrow$ [ ]
for each rule in KB do
$(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES(rule)
for each θ such that SUBST($\theta, p_1 \wedge \ldots \wedge p_n$) = SUBST($\theta, p_1' \wedge \ldots \wedge p_n'$) for some $p_1' \ldots p_n'$ in KB
$q' \leftarrow$ SUBST(θ, q)
if q' does not unify with some sentences already in KB or new then add q' to new
$\phi \leftarrow$ UNIFY(q', α)
if ϕ is NOT fail then return ϕ
add new to KB
return false



CODE:

```
print('Sinchana Hemanth 1BM23CS330')
facts = {
    'American(West)': True,
    'Hostile(Nono)': True,
    'Missiles(Nono)': True,
}
def rule1(facts):
    if facts.get('American(West)', False) and facts.get('Hostile(Nono)', False):
        return 'Criminal(West)'
```

```
    return None

def rule2(facts):
    if facts.get('Missiles(Nono)', False) and facts.get('Hostile(Nono)', False):
        return 'SoldWeapons(West, Nono)'

def forward_chaining(facts, rules):
    new_facts = facts.copy()
    inferred = True
    while inferred:
        inferred = False
        for rule in rules:
            result = rule(new_facts)
            if result and result not in new_facts:
                new_facts[result] = True
                inferred = True
                print(f"New fact inferred: {result}")
    return new_facts
rules = [rule1, rule2]

inferred_facts = forward_chaining(facts, rules)

print("\nFinal facts:")
for fact in inferred_facts:
    print(fact)
```

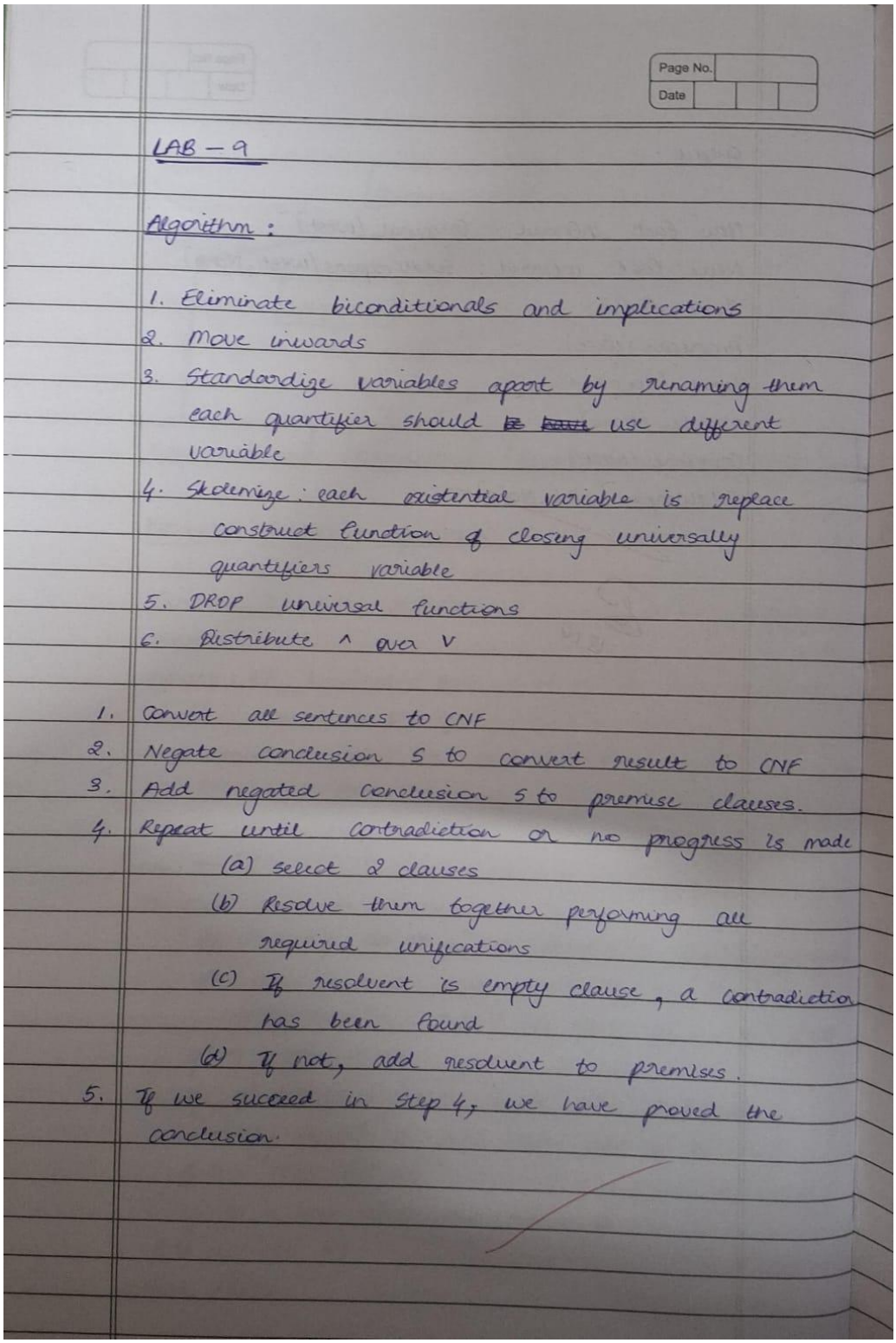
OUTPUT:

```
Sinchana Hemanth 1BM23CS330
New fact inferred: Criminal(West)
New fact inferred: SoldWeapons(West, Nono)

Final facts:
American(West)
Hostile(Nono)
Missiles(Nono)
Criminal(West)
SoldWeapons(West, Nono)
```

Program 9

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

ALGORITHM:



LAB – 9

Algorithm :

1. Eliminate biconditionals and implications
2. Move inwards
3. Standardize variables apart by renaming them each quantifier should ~~be~~ ~~have~~ use different variable
4. Skolemize : each existential variable is replace construct function of closing universally quantifiers variable
5. DROP universal functions
6. Distribute ∧ over ∨

1. Convert all sentences to CNF
2. Negate conclusion S to convert result to CNF
3. Add negated conclusion S to premise clauses.
4. Repeat until contradiction or no progress is made
   (a) select 2 clauses
   (b) Resolve them together performing all required unifications
   (c) If resolvent is empty clause, a contradiction has been found
   (d) If not, add resolvent to premises.
5. If we succeed in Step 4, we have proved the conclusion.

(a) food(x) ∨ likes (John, x)

(b) food (Apple)

(c) food (vegetables)

(d) ¬ eats (y, z) ∨ killed (y) ∨ food(z)

(e) eats (Anil, Peanuts)

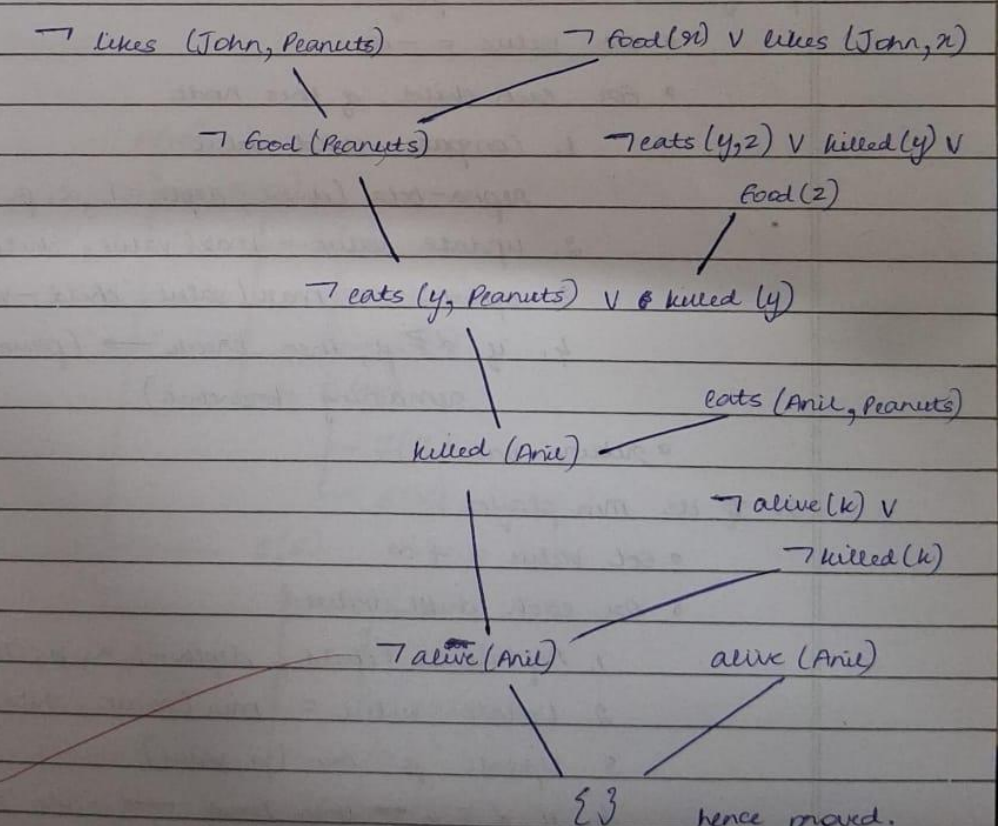
(f) alive (Anil)

(g) ¬ eats (Anil, w) ∨ eats (Harry, w)

(h) killed (g) ∨ alive (g)

(i) ¬ alive (k) ∨ ¬ killed (k)

(j) likes (John, Peanuts)



CODE:

```
from collections import deque
import itertools
import copy
import pprint
print('Sinchana Hemanth 1BM23CS330')
class Var:
    def __init__(self, name):
        self.name = name
    def __repr__(self):
        return f"Var({self.name})"
    def __eq__(self, other):
        return isinstance(other, Var) and self.name == other.name
    def __hash__(self):
        return hash(('Var', self.name))

class Const:
    def __init__(self, name):
        self.name = name
    def __repr__(self):
        return f"Const({self.name})"
    def __eq__(self, other):
        return isinstance(other, Const) and self.name == other.name
    def __hash__(self):
        return hash(('Const', self.name))

class Func:
    def __init__(self, name, args):
        self.name = name
        self.args = args
    def __repr__(self):
        return f"Func({self.name}, {self.args})"
    def __eq__(self, other):
        return isinstance(other, Func) and self.name == other.name and self.args == other.args
    def __hash__(self):
        return hash(('Func', self.name, tuple(self.args)))

class Literal:
    def __init__(self, predicate, args, negated=False):
        self.predicate = predicate
        self.args = tuple(args)
        self.negated = negated
    def negate(self):
        return Literal(self.predicate, list(self.args), not self.negated)
    def __repr__(self):
        sign = "~" if self.negated else ""
        args = ",".join(map(term_to_str, self.args))
        return f"{sign}{self.predicate}({args})"
    def __eq__(self, other):
        return (self.predicate, self.args, self.negated) == (other.predicate, other.args, other.negated)
    def __hash__(self):
```

```
        return hash((self.predicate, self.args, self.negated))

def clause_to_str(cl):
    return " OR ".join(map(str, cl)) if cl else "EMPTY"

def term_to_str(t):
    if isinstance(t, Var):
        return t.name
    if isinstance(t, Const):
        return t.name
    if isinstance(t, Func):
        return f"{t.name}({','.join(term_to_str(a) for a in t.args)})"
    return str(t)

def apply_subst_term(term, subst):
    if isinstance(term, Var):
        if term in subst:
            return apply_subst_term(subst[term], subst)
        else:
            return term
    elif isinstance(term, Const):
        return term
    elif isinstance(term, Func):
        return Func(term.name, [apply_subst_term(a, subst) for a in term.args])
    else:
        return term

def apply_subst_literal(lit, subst):
    return Literal(lit.predicate, [apply_subst_term(a, subst) for a in lit.args], lit.negated)

def apply_subst_clause(clause, subst):
    return frozenset(apply_subst_literal(l, subst) for l in clause)

def occurs_check(var, term, subst):
    term = apply_subst_term(term, subst)
    if term == var:
        return True
    if isinstance(term, Func):
        return any(occurs_check(var, arg, subst) for arg in term.args)
    return False

def unify_terms(x, y, subst):
    # returns updated subst or None on failure
    x = apply_subst_term(x, subst)
    y = apply_subst_term(y, subst)
    if isinstance(x, Var):
        if x == y:
            return subst
        if occurs_check(x, y, subst):
            return None
        new = subst.copy()
        new[x] = y
```

```
        return new
    if isinstance(y, Var):
        return unify_terms(y, x, subst)
    if isinstance(x, Const) and isinstance(y, Const):
        return subst if x.name == y.name else None
    if isinstance(x, Func) and isinstance(y, Func) and x.name == y.name and len(x.args) == len(y.args):
        for a, b in zip(x.args, y.args):
            subst = unify_terms(a, b, subst)
            if subst is None:
                return None
        return subst
    return None

def unify_literals(l1, l2):
    if l1.predicate != l2.predicate or l1.negated == l2.negated or len(l1.args) != len(l2.args):
        return None
    subst = {}
    for a, b in zip(l1.args, l2.args):
        subst = unify_terms(a, b, subst)
        if subst is None:
            return None
    return subst

_var_count = 0
def standardize_apart(clause):
    global _var_count
    varmap = {}
    new_literals = []
    for lit in clause:
        new_args = []
        for t in lit.args:
            new_args.append(_rename_term_vars(t, varmap))
        new_literals.append(Literal(lit.predicate, new_args, lit.negated))
    return frozenset(new_literals)

def _rename_term_vars(term, varmap):
    global _var_count
    if isinstance(term, Var):
        if term.name not in varmap:
            _var_count += 1
            varmap[term.name] = Var(f"{term.name}_{_var_count}")
        return varmap[term.name]
    if isinstance(term, Const):
        return term
    if isinstance(term, Func):
        return Func(term.name, [_rename_term_vars(a, varmap) for a in term.args])
    return term

def resolve(ci, cj):
    resolvents = set()
    ci = standardize_apart(ci)
    cj = standardize_apart(cj)
```

```
    for li in ci:
        for lj in cj:
            if li.predicate == lj.predicate and li.negated != lj.negated and len(li.args) == len(lj.args):
                subst = unify_literals(li, lj)
                if subst is not None:
                    # build resolvent: (Ci - {li}) U (Cj - {lj}) with subst applied
                    new_clause = set(apply_subst_literal(l, subst) for l in (ci - {li}) | (cj - {lj}))
                    # remove tautologies: a clause containing P and ~P after subst
                    preds = {}
                    taut = False
                    for l in new_clause:
                        key = (l.predicate, tuple(map(term_to_str, l.args)))
                        if key in preds and preds[key] != l.negated:
                            taut = True
                            break
                        preds[key] = l.negated
                    if not taut:
                        resolvents.add(frozenset(new_clause))
    return resolvents

def fol_resolution(kb_clauses, query_clause, max_iterations=20000):
    """
    kb_clauses: set/list of clauses (each clause is frozenset of Literal)
    query_clause: single Literal (to be proved), will be negated and added to KB
    Returns True if contradiction (empty clause) is derived.
    """
    negated_query = [query_clause.negate()]
    clauses = set(kb_clauses)
    for l in negated_query:
        clauses.add(frozenset([l]))

    new = set()
    processed_pairs = set()
    queue = list(clauses)

    iterations = 0
    while True:
        pairs = []
        clause_list = list(clauses)
        n = len(clause_list)
        for i in range(n):
            for j in range(i+1, n):
                pairs.append((clause_list[i], clause_list[j]))

        something_added = False
        for (ci, cj) in pairs:
            pair_key = (ci, cj)
            if pair_key in processed_pairs:
                continue
            processed_pairs.add(pair_key)
            resolvents = resolve(ci, cj)
            iterations += 1
```

```
            if iterations > max_iterations:
                return False, "max_iterations_exceeded"
            for r in resolvents:
                if len(r) == 0:
                    return True, "Derived empty clause (success)"
                if r not in clauses and r not in new:
                    new.add(r)
                    something_added = True
        if not something_added:
            return False, "No new clauses — failure (KB does not entail query)"
        clauses.update(new)
        new = set()

def C(name): return Const(name)
def V(name): return Var(name)
def F(name, *args): return Func(name, list(args))
def L(pred, args, neg=False): return Literal(pred, args, neg)

x = V('x')
y = V('y')

kb = set()

kb.add(frozenset([L('Food', [x], neg=True), L('Likes', [C('John'), x], neg=False)]))

kb.add(frozenset([L('Food', [C('apple')], neg=False)]))
kb.add(frozenset([L('Food', [C('vegetable')], neg=False)]))

kb.add(frozenset([L('Eats', [x,y], neg=True), L('Killed', [y], neg=False), L('Food', [y], neg=False)]))

kb.add(frozenset([L('Eats', [C('Anil'), C('peanuts')], neg=False)]))
kb.add(frozenset([L('Alive', [C('Anil')], neg=False)]))

kb.add(frozenset([L('Eats', [C('Anil'), x], neg=True), L('Eats', [C('Harry'), x], neg=False)]))

kb.add(frozenset([L('Alive', [x], neg=True), L('Killed', [x], neg=True)]))

kb.add(frozenset([L('Killed', [x], neg=True), L('Alive', [x], neg=False)]))

query = L('Likes', [C('John'), C('peanuts')], neg=False)

def show_kb(kb):
    print("Knowledge base clauses:")
    for c in kb:
        print("  ", clause_to_str(c))
    print()

if __name__ == "__main__":
    print("FOL resolution prover (basic example)\n")
    show_kb(kb)
    print("Query:", query)
    print("Negated query clause will be added to KB and resolution attempted.\n")
```

```
success, info = fol_resolution(kb, query, max_iterations=20000)
print("Result:", success, "|", info)
```

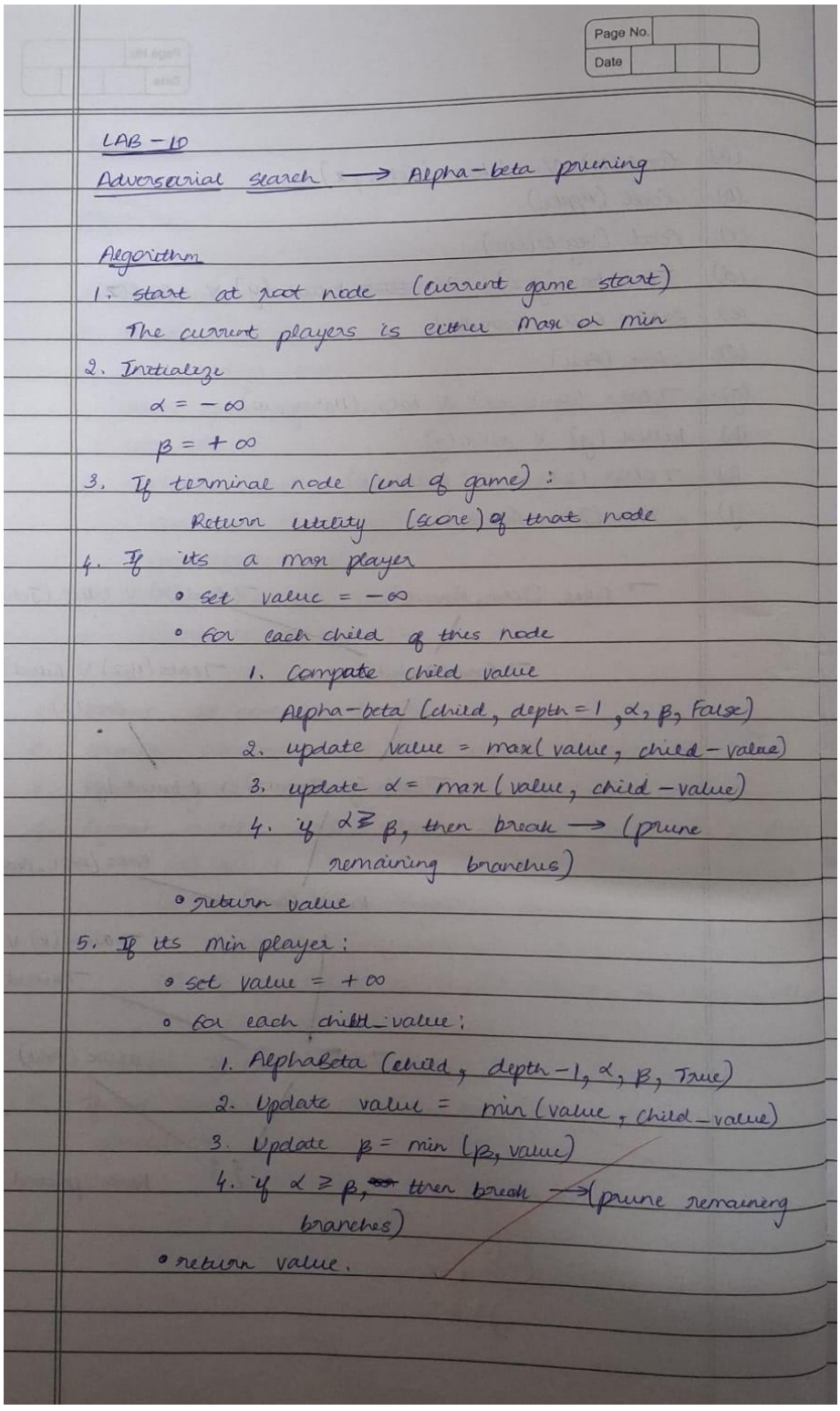
OUTPUT:

```
Sinchana Hemanth 1BM23CS330
FOL resolution prover (basic example)

Knowledge base clauses:
   ~Killed(x) OR Alive(x)
   Food(apple)
   ~Alive(x) OR ~Killed(x)
   Eats(Anil,peanuts)
   Food(vegetable)
   ~Eats(Anil,x) OR Eats(Harry,x)
   Alive(Anil)
   Likes(John,x) OR ~Food(x)
   Killed(y) OR Food(y) OR ~Eats(x,y)

Query: Likes(John,peanuts)
Negated query clause will be added to KB and resolution attempted.

Result: True | Derived empty clause (success)
```

Program 10

Implement Alpha-Beta Pruning

ALGORITHM:



LAB - 10

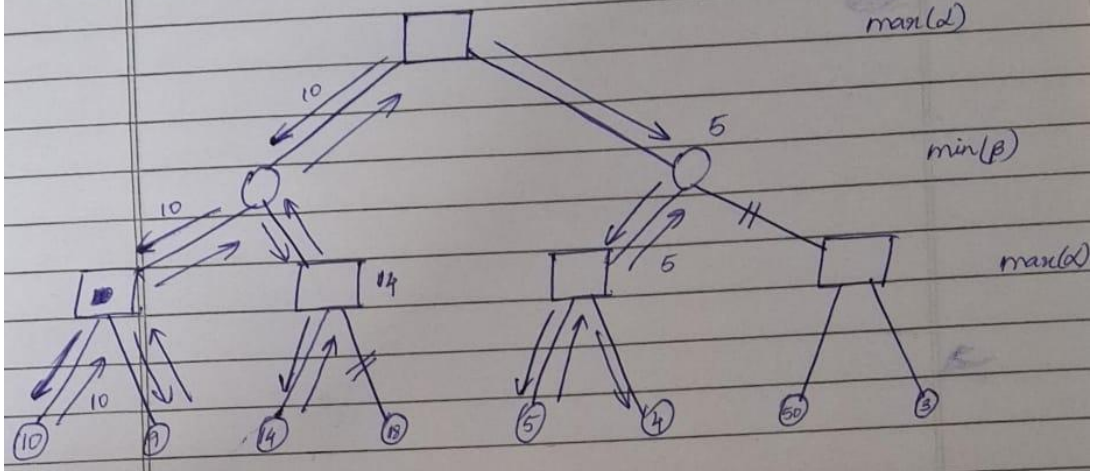
Adversarial search → Alpha-beta pruning

Algorithm

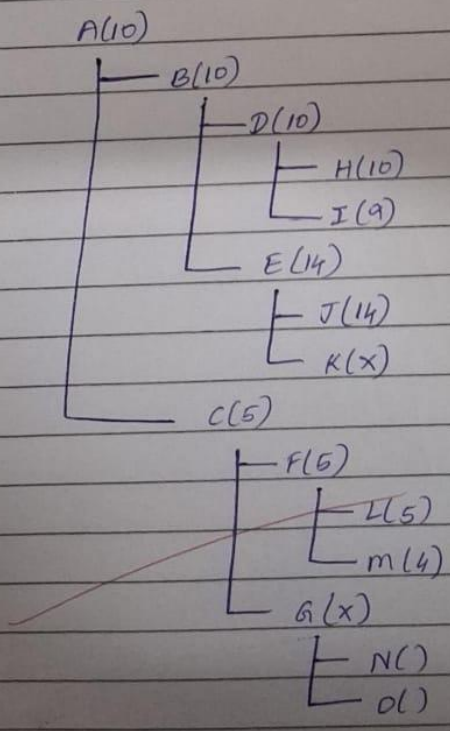
1. start at root node (current game start)
   The current players is either Max or min
2. Initialize
   $\alpha = -\infty$
   $\beta = +\infty$
3. If terminal node (end of game):
   Return utility (score) of that node
4. If its a max player
   - set value = $-\infty$
   - for each child of this node
     1. Compute child value
        Alpha-beta (child, depth = 1, α, β, False)
     2. update value = max(value, child - value)
     3. update α = max(value, child - value)
     4. if $\alpha \geq \beta$, then break → (prune remaining branches)
   - return value
5. If its min player:
   - set value = $+\infty$
   - for each child: value:
     1. AlphaBeta (child, depth - 1, α, β, True)
     2. Update value = min (value, child - value)
     3. Update β = min (β, value)
     4. if $\alpha \geq \beta$, then break → (prune remainerg branches)
   - return value.

Page No.
Date

4. ~~If $\alpha \geq \beta$ then break => (prune remaining branches)~~



Output:



22/1/24

CODE:

```
print('Sinchana Hemanth 1BM23CS330')
def alpha_beta(node_index, depth, max_depth, alpha, beta, is_max, values, explored, pruned, path):
    """
    node_index : index of current node in conceptual tree
    depth      : current depth (0 = root)
    max_depth  : total depth of tree
    values     : list of leaf node values
    """

    total_leaves = len(values)

    if depth == max_depth:
        leaf_index = node_index - (2 ** max_depth - 1)
        if 0 <= leaf_index < total_leaves:
            val = values[leaf_index]
            explored.append((list(path), val))
            return val
        else:
            return 0
    if is_max:
        value = float('-inf')
        for i in range(2):
            child_index = node_index * 2 + i + 1
            path.append(child_index)
            value = max(value, alpha_beta(child_index, depth + 1, max_depth,
                                          alpha, beta, False, values, explored, pruned, path))
            path.pop()
            alpha = max(alpha, value)
            if beta <= alpha:
                pruned.append((node_index, child_index, 'Beta cutoff'))
                break
        return value
    else:
        value = float('inf')
        for i in range(2):
            child_index = node_index * 2 + i + 1
            path.append(child_index)
            value = min(value, alpha_beta(child_index, depth + 1, max_depth,
                                          alpha, beta, True, values, explored, pruned, path))
            path.pop()
            beta = min(beta, value)
            if beta <= alpha:
                pruned.append((node_index, child_index, 'Alpha cutoff'))
                break
        return value

if __name__ == "__main__":
    values = [3, 5, 6, 9, 1, 2, 0, -1]
    max_depth = 3
    explored, pruned = [], []
```

```
print("Leaf node values:", values)
result = alpha_beta(0, 0, max_depth, float('-inf'), float('inf'),
                    True, values, explored, pruned, [0])

print("\nValue of root node (MAX mode):", result)
print("\nExplored leaf paths:")
for p, val in explored:
    print(f"Path {p} -> Value {val}")

print("\nPruned branches:")
for item in pruned:
    print(item)
```

OUTPUT:

```
Sinchana Hemanth 1BM23CS330
Leaf node values: [3, 5, 6, 9, 1, 2, 0, -1]

Value of root node (MAX mode): 5

Explored leaf paths:
Path [0, 1, 3, 7] -> Value 3
Path [0, 1, 3, 8] -> Value 5
Path [0, 1, 4, 9] -> Value 6
Path [0, 2, 5, 11] -> Value 1
Path [0, 2, 5, 12] -> Value 2

Pruned branches:
(4, 9, 'Beta cutoff')
(2, 5, 'Alpha cutoff')
```