VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Suhas MS (1BM23CS346)

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 **Suhas MS (1BM23CS346),** 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.

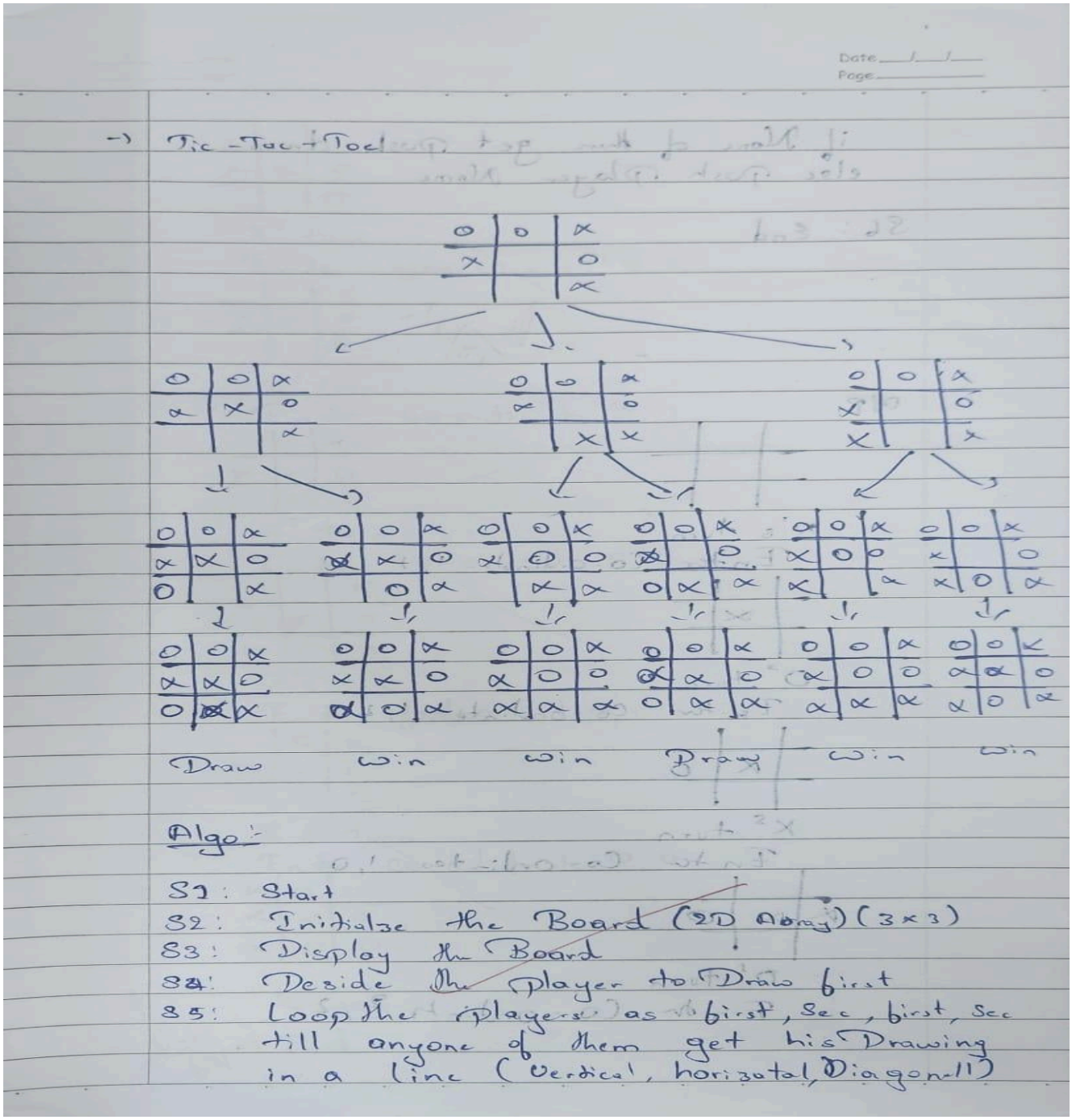
Prof. Sonika Sharma Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/suhasms369/AI_Lab_1BM23CS346.git

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

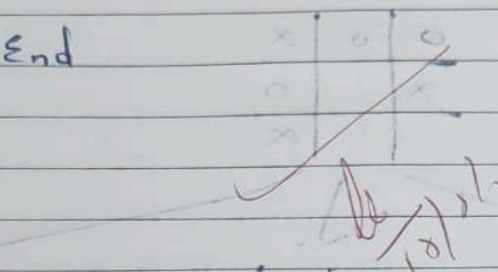
Algorithm:



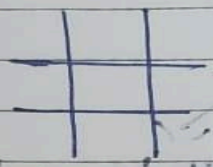
Date __/__/__
Page _____

if None of them get push draw
else push Player Name

S6: End

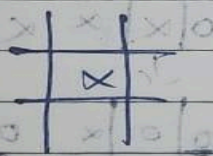


O/P



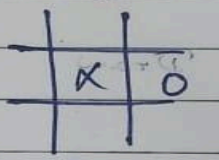
X's turn

Enter Co-ordinate 1, 1



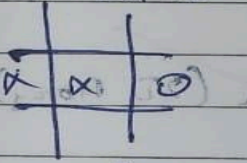
O's turn

Enter Co-ordinate 1, 2



X's turn

Enter Co-ordinates 1, 0



O's turn

Enter Co-ordinates 2, 2

Date / /
Page

	X	O
		O

X's turn

Enter ~~your~~ Co-ordinate 0, 2

		X
X	X	O
		O

O's turn

Enter Co-ordinate 2, 1

		X
X	X	O
	O	O

X's turn

Enter Co-ordinate 2, 0

		X
X	X	O
X	O	O

O's turn

Enter ~~good~~ co-ordinate 0, 1

	O	X
X	X	O
X	O	O

~~X~~'s turn

Enter co-odinate 0, 0

X	O	X
X	X	O
X	O	O

X won

End:

Code:

```
import random

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

def check_win(board, player):
    # Check rows
    for row in board:
        if all([cell == player for cell in row]):
            return True

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

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

    return False

def check_draw(board):
    for row in board:
        if " " in row:
            return False
    return True

def play_game():
    board = [[" " for _ in range(3)] for _ in range(3)]
    players = ["X", "O"]
    current_player = random.choice(players)

    print("Welcome to Tic Tac Toe!")
    print_board(board)

    while True:
        print(f"It's {current_player}'s turn.")
        try:
            move = input("Enter your move (row,column): ")
            row, col = map(int, move.split(','))

            if 0 <= row <= 2 and 0 <= col <= 2 and board[row][col] == " ":
                board[row][col] = current_player
                print_board(board)

                if check_win(board, current_player):
                    print(f"Player {current_player} wins!")
                    break
                elif check_draw(board):
                    print("It's a draw!")
                    break
```

```
                else:
                    current_player = "O" if current_player == "X" else "X"
            else:
                print("Invalid move. Try again.")
        except ValueError:
            print("Invalid input format. Please enter in the format 'row,column'.")
        except IndexError:
            print("Invalid input format. Please enter in the format 'row,column'.")


play_game()

print("END \n By : Suhas M S")
```

Output:

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

```
---------
  | X |
---------
O |   | O
---------
It's X's turn.
Enter your move (row,column): 1,0
O |   | X
---------
X | X |
---------
O |   | O
---------
It's O's turn.
Enter your move (row,column): 2,1
O |   | X
---------
X | X |
---------
O | O | O
---------
Player O wins!
END
 By : Suhas M S
```

→ Vaccum cleaner

Algo:-

S1: Start
S2: Cheak the room if its clean / Dirty
S3: if Dirty clean it
else Ask the user
either to go Room B if in Room A
or to go Room A if in Room B
or to stay.
S4: if ~~to continue~~ dirty
goto S2
else continue
S5: End:

O/P

Enter status of RoomA: dirty
Enter status of RoomB: dirty
which room to go: A
RoomA is dirty
Cleaning Now.
which room to move B:
RoomB is dirty
Cleaning now.
All room are clean
Total cost: 2

Code:

```
def vacuum_simulator():
    rooms = {}
    for room in ['A', 'B']:
        while True:
            status = input(f"Enter status of room {room} (C for clean, D for
dirty): ").strip().upper()
            if status in ['C', 'D']:
                rooms[room] = status
                break
            else:
                print("Invalid input. Please enter 'C' or 'D'.")

    total_cost = 0

    while True:
        if all(status == 'C' for status in rooms.values()):
            print("All rooms are clean. Exiting.")
            break

        move = input("Which room to move to? (A or B): ").strip().upper()
        if move not in rooms:
            print("Invalid room. Please enter 'A' or 'B'.")
            continue

        total_cost += 1

        if rooms[move] == 'C':
            print(f"Room {move} is clean. Continuing...")
        else:
            print(f"Room {move} is dirty. Cleaning now.")
            rooms[move] = 'C'

    print(f"Total cost of moves: {total_cost}")

vacuum_simulator()


print("End\nBy Suhas M S\n   1BM23CS346")
```

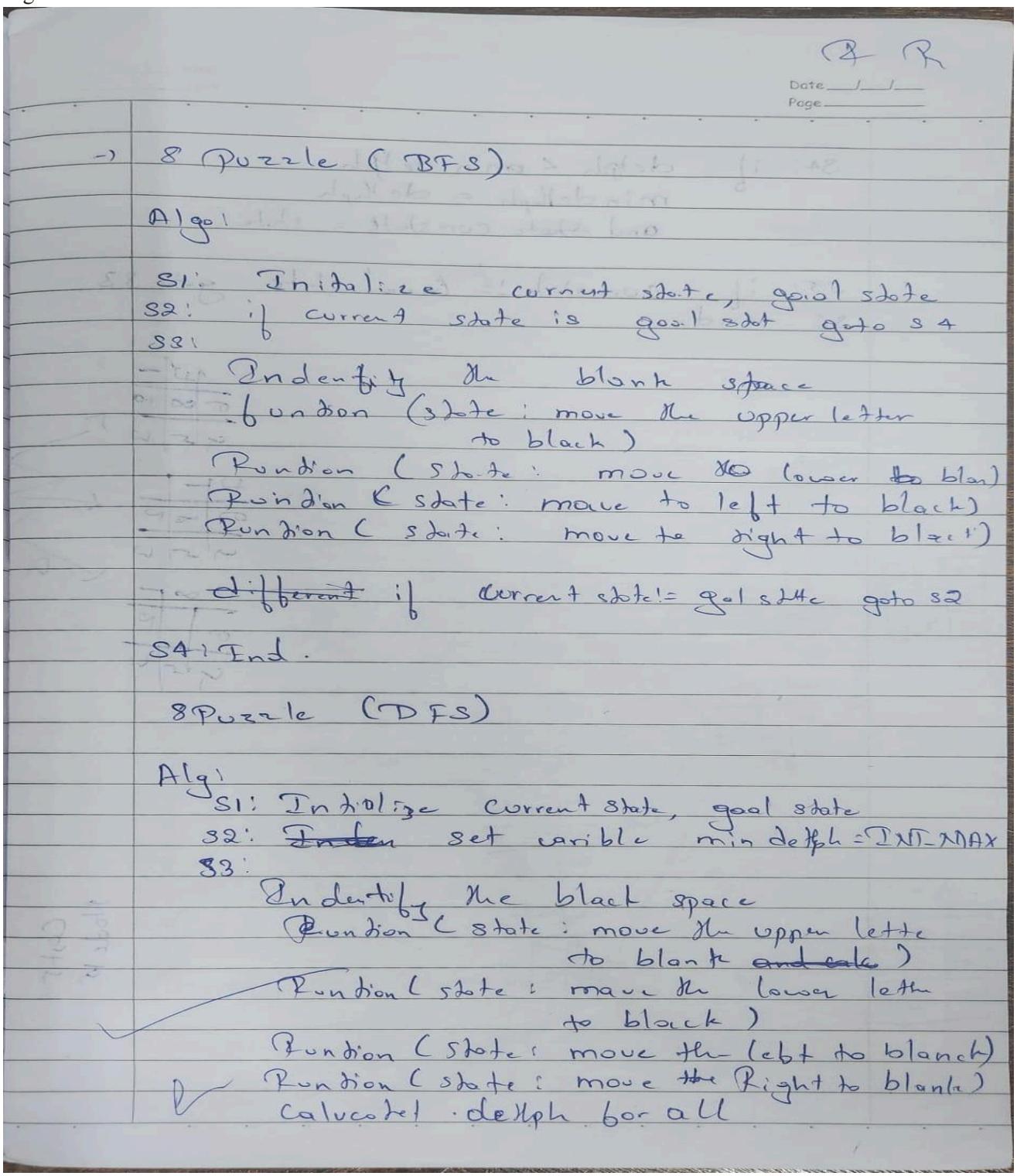
Output:

```
Enter status of room A (C for clean, D for dirty): d
Enter status of room B (C for clean, D for dirty): d
Which room to move to? (A or B): a
Room A is dirty. Cleaning now.
Which room to move to? (A or B): a
Room A is clean. Continuing...
Which room to move to? (A or B): b
Room B is dirty. Cleaning now.
All rooms are clean. Exiting.
Total cost of moves: 3
End
By Suhas M S
   1BM23CS346
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm

Algorithm:



8 Puzzle (BFS)

Algo:

S1: Initialize current state, goal state
S2: if current state is goal stat goto S4
S3:
- Indentify the blank space
- Function (state: move the upper letter to black)
- Function (state: move to (lower to blan)
- Function (state: move to left to black)
- Function (state: move to right to black)

- ~~different~~ if Current state != goal state goto S2

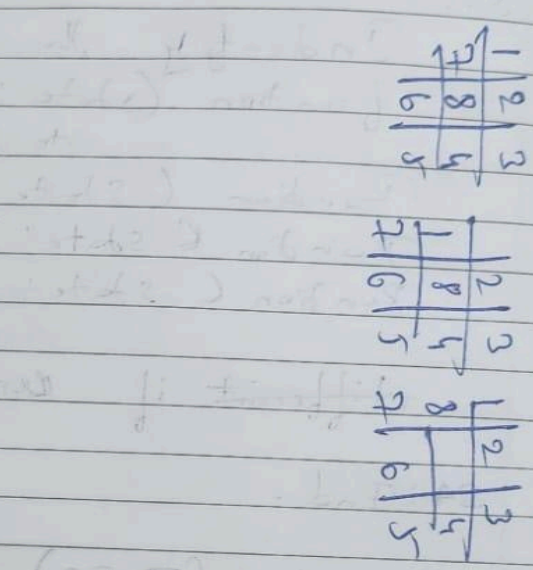
S4: End.

8Puzzle (DFS)

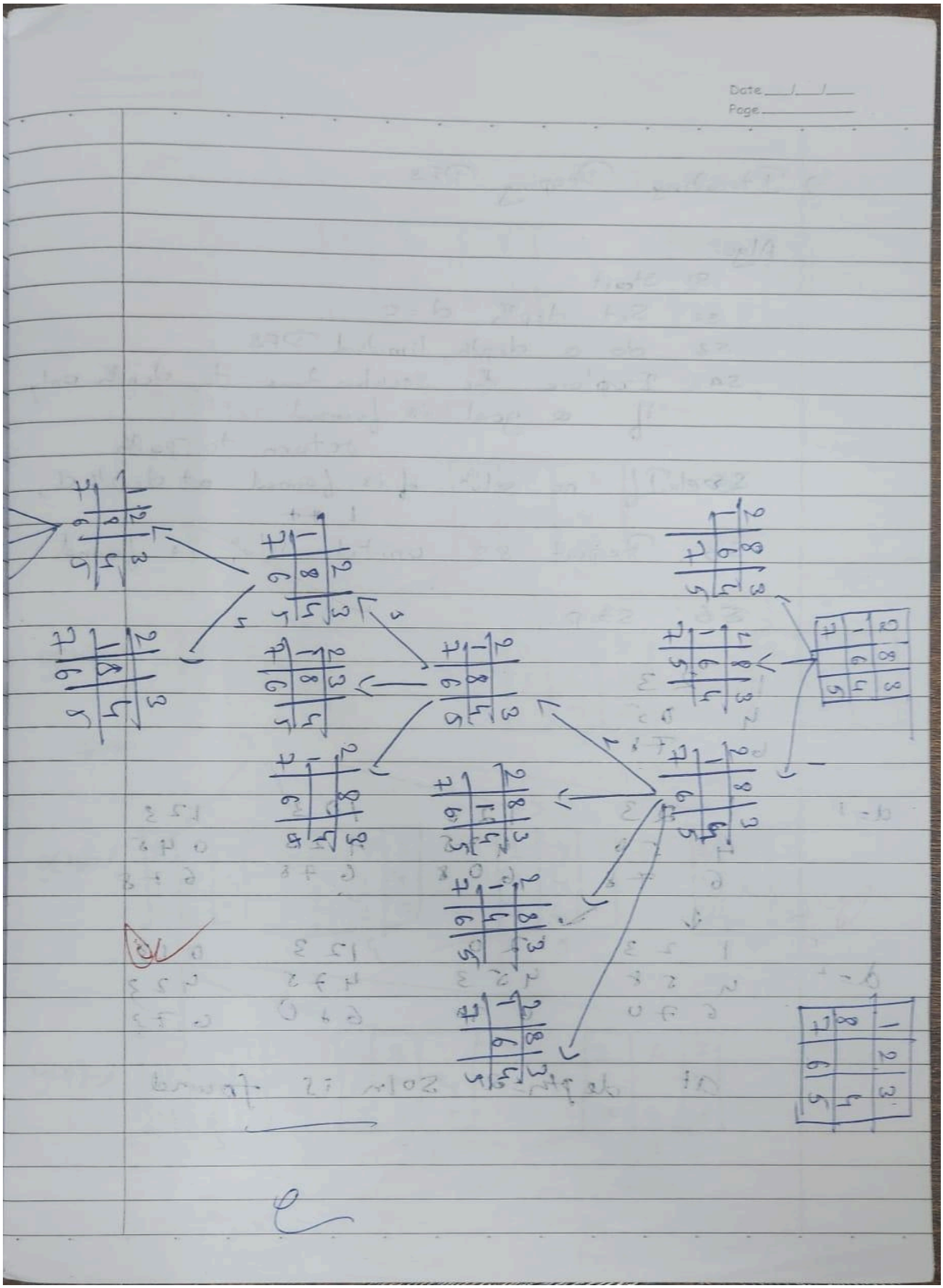
Alg:
S1: Initialize current state, goal state
S2: ~~Indep~~ set varible min depth = INT_MAX
S3:
Indentify the black space
Function (state: move the upper letter to blank ~~and calc~~)
Function (state: move the lower letter to black)
Function (state: move the (eft to blanch)
Function (state: move the Right to blank)
Calucatel depth for all

S4: if depth < min depth
min depth = depth
and ~~state~~ curr state = state

S5: if curr state != Goal state goto S3.
S6: end.



Costs
Node 15



Code:

```
import collections
from typing import Tuple, List, Optional

State = Tuple[Tuple[int, ...], ...]

def find_empty_tile(state: State) -> Tuple[int, int]:
    for r, row in enumerate(state):
        for c, val in enumerate(row):
            if val == 0:
                return r, c
    raise ValueError

def get_next_states(state: State) -> List[State]:
    er, ec = find_empty_tile(state)
    moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    out = []
    for dr, dc in moves:
        nr, nc = er + dr, ec + dc
        if 0 <= nr < 3 and 0 <= nc < 3:
            nb = [list(r) for r in state]
            nb[er][ec], nb[nr][nc] = nb[nr][nc], nb[er][ec]
            out.append(tuple(tuple(r) for r in nb))
    return out

def pretty_print(state: State) -> None:
    for row in state:
        print(" ".join(str(v) for v in row))
    print()

def solve_puzzle_dfs(initial_state: State, goal_state: State, max_depth:
Optional[int] = None) -> Optional[List[State]]:
    stack = [(initial_state, [initial_state], 0)]
    visited = {initial_state}
    while stack:
        cur, path, depth = stack.pop()
        if cur == goal_state:
            return path
        if max_depth is not None and depth >= max_depth:
            continue
        for nxt in get_next_states(cur):
            if nxt not in visited:
                visited.add(nxt)
                stack.append((nxt, path + [nxt], depth + 1))
    return None

initial_state_3x3: State = (
    (2, 8, 3),
    (1, 6, 4),
    (7, 0, 5),
)

goal_state_3x3: State = (
    (2, 8, 3),
    (0, 1, 4),
    (7, 6, 5),
)
```

```python
print("Attempting to solve the 3×3 puzzle with DFS...\n")
solution_path = solve_puzzle_dfs(initial_state_3x3, goal_state_3x3)

if solution_path:
    print(f"Solution found in {len(solution_path)-1} moves:\n")
    for i, st in enumerate(solution_path):
        print(f"Step {i}:")
        pretty_print(st)
else:
    print("No solution found.")

print("By \n  Suhas M S \n  1BM23CS346")
```

```
Attempting to solve the 3×3 puzzle with DFS...

Solution found in 2 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 8 3
0 1 4
7 6 5

By
  Suhas M S
  1BM23CS346
```

Date / /
Page

→ Iterating Deeping DFS:

Algo:

S1: Start

S2: Set depth d = 0

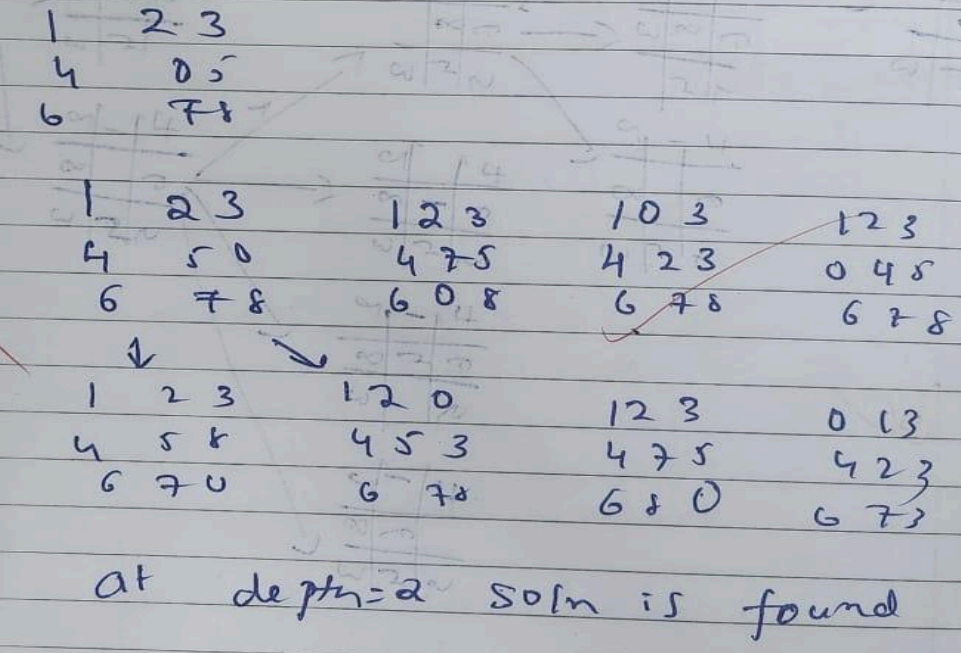
S3: do a depth limited DFS

S4: Explore the search tree to depth only if a goal is formed return to path

If no soln is formed at depth d, d++

S5: Repeat S3 until soln is found

S6: Stop!



at depth=2 soln is found

Code:

```
from collections import deque

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

MOVES = {
    'UP': (-1, 0),
    'DOWN': (1, 0),
    'LEFT': (0, -1),
    'RIGHT': (0, 1)
}

def find_zero(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def is_valid_pos(x, y):
    return 0 <= x < 3 and 0 <= y < 3

def swap_positions(state, pos1, pos2):
    state_list = [list(row) for row in state]
    x1, y1 = pos1
    x2, y2 = pos2
    state_list[x1][y1], state_list[x2][y2] = state_list[x2][y2],
state_list[x1][y1]
    return tuple(tuple(row) for row in state_list)

def get_neighbors(state):
    neighbors = []
    x, y = find_zero(state)
    for move in MOVES.values():
        new_x, new_y = x + move[0], y + move[1]
        if is_valid_pos(new_x, new_y):
            neighbors.append(swap_positions(state, (x, y), (new_x,
new_y)))
    return neighbors

def reconstruct_path(came_from, current):
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
    path.reverse()
    return path

def dls(state, depth_limit, came_from, visited):
    """Depth Limited Search"""
    if state == GOAL_STATE:
        return True
    if depth_limit <= 0:
        return False

    for neighbor in get_neighbors(state):
```

```
            if neighbor not in visited:
                visited.add(neighbor)
                came_from[neighbor] = state
                if dls(neighbor, depth_limit - 1, came_from, visited):
                    return True
        return False

def iddfs(start_state, max_depth=50):
    """Iterative Deepening DFS"""
    for depth in range(max_depth):
        came_from = {}
        visited = {start_state}
        if dls(start_state, depth, came_from, visited):
            return reconstruct_path(came_from, GOAL_STATE)
    return None

def print_state(state):
    for row in state:
        print(' '.join(str(x) if x != 0 else '_' for x in row))
    print()

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

    print("Initial State:")
    print_state(start_state)

    print("Solving with Iterative Deepening DFS...")
    iddfs_path = iddfs(start_state)
    if iddfs_path:
        print(f"Solution found in {len(iddfs_path) - 1} moves!")
        for state in iddfs_path:
            print_state(state)
    else:
        print("No solution found with IDDFS.")

    print("By:\n   Suhas MS\n   1BM24CS346\n")
```

```
Initial State:
2 8 3
1 6 4
7 _ 5

Solving with Iterative Deepening DFS...
Solution found in 5 moves!
2 8 3
1 6 4
7 _ 5

2 8 3
1 _ 4
7 6 5

2 _ 3
1 8 4
```

```
7 6 5

_ 2 3
1 8 4
7 6 5

1 2 3
_ 8 4
7 6 5

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

By:

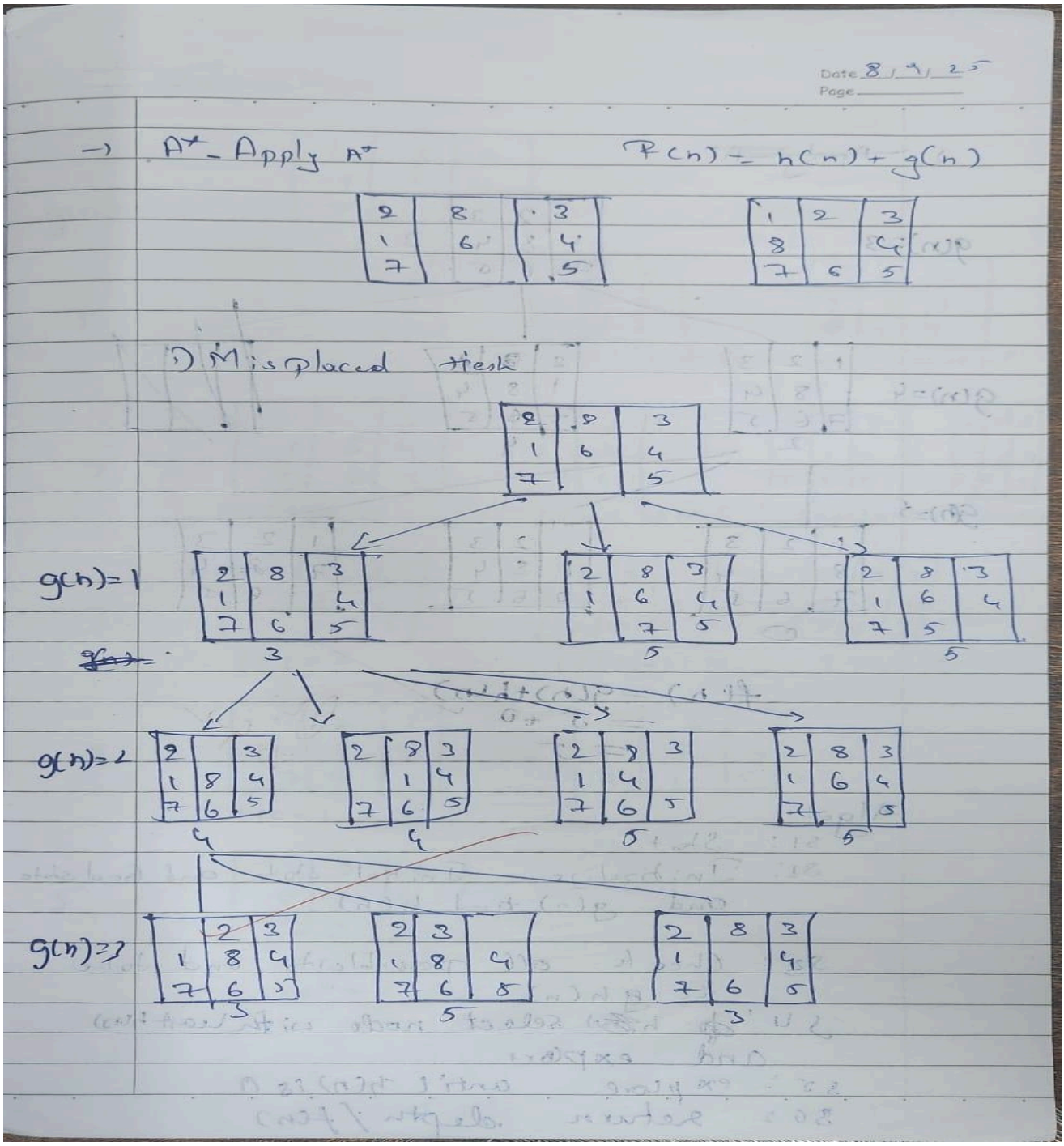
Suhas MS

1BM24CS346

Program 3

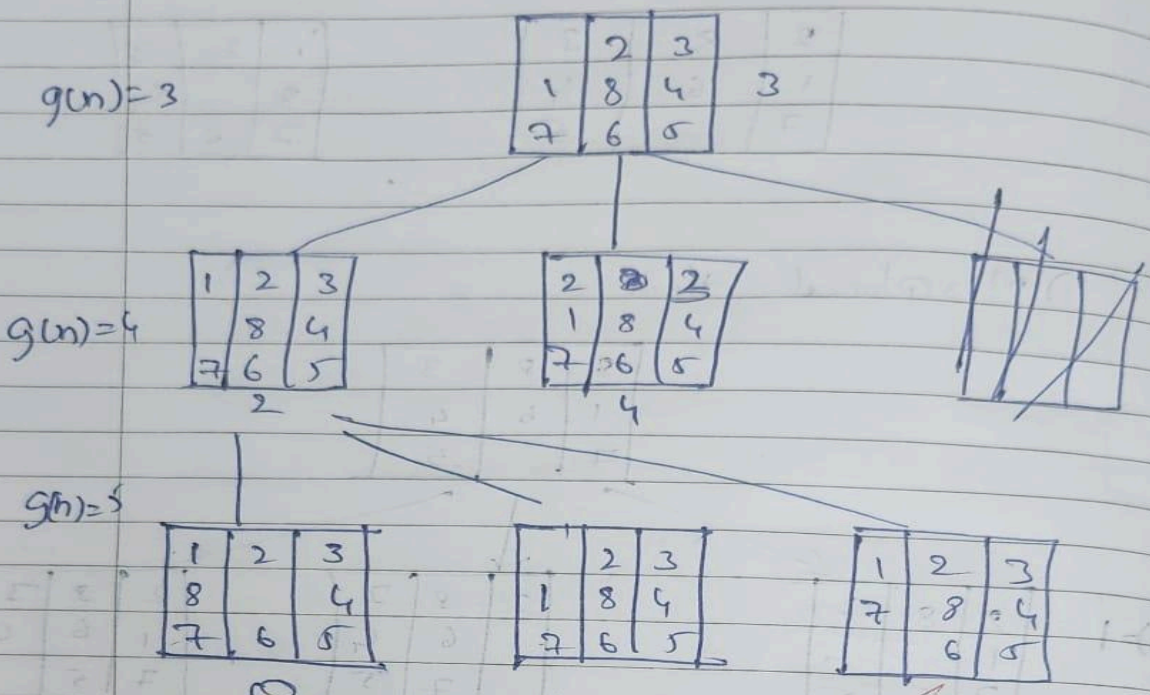
Implement A* search algorithm

Algorithm:



Date / /
Page

:) ~~Apply A*~~



$$f(n) = g(n) + h(n)$$
$$= 5 + 0$$
$$= 5$$

Algo!

S1: Start

S2: Initialize Initial state and Goal state and g(n) and h(n)

S3: Check all possibleits and take the h(n)

S4: select node with least h(n) and explore

S5: explore until h(n) is 0

S6: return depth / fcns

Manhattan

Date __/__/__
Page _____

gcnt=0

2	8	3
1	6	4
7		5

1

2	8	3
1		4
7	6	5

1+2+1=4

2	8	3
1	6	4
7	5	

1+2+1+1=5

2	8	3
1	6	4
	7	5

1+2+2+1=6

2

2		3
1	8	4
7	6	5

1+1+1=3

2	8	3
1	4	
7	6	5

1+2+1+1=5

2	8	3
	1	4
7	6	5

1+2+2=5

2	8	3
1	6	4
7	6	5

1+2+1+1=5

3

	2	3
1	8	4
7	6	5

1+1=2

2	3	
1	8	4
7	6	5

1+1+1=4

2	8	3
1		4
7	6	5

1+2+1=4

4

1	2	3
	8	4
7	6	5

1+0=1

2	3	
1	8	4
7	6	5

5

1	2	3
8		4
7	6	5

cost=5

Misplaced O/p:

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

Step 1: Moves: U
2 8 3
1 6 4
7 5

Step 2: Moves UU
2 3
1 8 4
7 6 5

Step 3: Moves UUL
2 3
1 8 4
7 6 5

Step 4: Moves UULD
1 2 3
8 4
7 6 5

Step 5: Moves UULDR
1 2 3
8 4
7 6 5

Code:

Mismatched

```
import heapq
import time

class PuzzleState:
    def __init__(self, board, goal, path="", cost=0):
        self.board = board
        self.goal = goal
        self.path = path
        self.cost = cost
        self.zero_pos = self.board.index(0)
        self.size = int(len(board) ** 0.5)

    def __lt__(self, other):

        return (self.cost + self.heuristic()) < (other.cost +
other.heuristic())

    def heuristic(self):

        misplaced = 0
        for i, tile in enumerate(self.board):
            if tile != 0 and tile != self.goal[i]:
                misplaced += 1
        return misplaced

    def get_neighbors(self):
        neighbors = []
        x, y = divmod(self.zero_pos, self.size)
        moves = {'U': (x - 1, y), 'D': (x + 1, y), 'L': (x, y -
1), 'R': (x, y + 1)}

        for move, (nx, ny) in moves.items():
            if 0 <= nx < self.size and 0 <= ny < self.size:
                new_zero_pos = nx * self.size + ny
                new_board = list(self.board)
                new_board[self.zero_pos],
new_board[new_zero_pos] = new_board[new_zero_pos],
new_board[self.zero_pos]
                neighbors.append(PuzzleState(tuple(new_board),
self.goal, self.path + move, self.cost + 1))
        return neighbors

def a_star(start, goal):
    start_state = PuzzleState(start, goal)
    frontier = []
    heapq.heappush(frontier, start_state)
    explored = set()
    parent_map = {start_state.board: None}
    move_map = {start_state.board: ""}

    while frontier:
        current_state = heapq.heappop(frontier)
```

```
        if current_state.board == goal:
            return reconstruct_path(parent_map, move_map,
current_state.board)

        explored.add(current_state.board)

        for neighbor in current_state.get_neighbors():
            if neighbor.board not in explored and neighbor.board
not in parent_map:
                parent_map[neighbor.board] = current_state.board
                move_map[neighbor.board] = neighbor.path[-1]
                heapq.heappush(frontier, neighbor)

    return None

def reconstruct_path(parent_map, move_map, state):
    path_boards = []
    path_moves = []
    while parent_map[state] is not None:
        path_boards.append(state)
        path_moves.append(move_map[state])
        state = parent_map[state]
    path_boards.append(state)
    path_boards.reverse()
    path_moves.reverse()
    return path_boards, path_moves

def print_board(board):
    size = int(len(board) ** 0.5)
    for i in range(size):
        row = board[i*size:(i+1)*size]
        print(" ".join(str(x) if x != 0 else " " for x in row))
    print()

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

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

    result = a_star(initial_state, final_state)
    if result:
        solution_boards, solution_moves = result
        print("Step-by-step solution:\n")
        for step_num, board in enumerate(solution_boards):
            moves_so_far = "".join(solution_moves[:step_num])
            print(f"Step {step_num}: Moves: {moves_so_far}")
            print_board(board)
            time.sleep(1)
    else:
        print("No solution found.")


    print("By \n  Suhas MS\n  1BM23CS346")
```

```
Step-by-step solution:

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

Step 1: Moves: U
2 8 3
1   4
7 6 5

Step 2: Moves: UU
2   3
1 8 4
7 6 5

Step 3: Moves: UUL
  2 3
1 8 4
7 6 5

Step 4: Moves: UULD
1 2 3
  8 4
7 6 5

Step 5: Moves: UULDR
1 2 3
8   4
7 6 5

By
  Suhas MS
  1BM23CS346
```

Manhattan

```python
import heapq
import time

class PuzzleState:
    def __init__(self, board, goal, path="", cost=0):
        self.board = board
        self.goal = goal
        self.path = path
        self.cost = cost
        self.zero_pos = self.board.index(0)
        self.size = int(len(board) ** 0.5)

    def __lt__(self, other):

        return (self.cost + self.heuristic()) < (other.cost +
```

```
other.heuristic())

    def heuristic(self):

        distance = 0
        for i, tile in enumerate(self.board):
            if tile != 0:
                goal_pos = self.goal.index(tile)
                distance += abs(i // self.size - goal_pos //
self.size) + abs(i % self.size - goal_pos % self.size)
        return distance

    def get_neighbors(self):
        neighbors = []
        x, y = divmod(self.zero_pos, self.size)
        moves = {'U': (x - 1, y), 'D': (x + 1, y), 'L': (x, y -
1), 'R': (x, y + 1)}

        for move, (nx, ny) in moves.items():
            if 0 <= nx < self.size and 0 <= ny < self.size:
                new_zero_pos = nx * self.size + ny
                new_board = list(self.board)

                new_board[self.zero_pos],
new_board[new_zero_pos] = new_board[new_zero_pos],
new_board[self.zero_pos]
                neighbors.append(PuzzleState(tuple(new_board),
self.goal, self.path + move, self.cost + 1))
        return neighbors

def a_star(start, goal):
    start_state = PuzzleState(start, goal)
    frontier = []
    heapq.heappush(frontier, start_state)
    explored = set()
    parent_map = {start_state.board: None}
    move_map = {start_state.board: ""}

    while frontier:
        current_state = heapq.heappop(frontier)

        if current_state.board == goal:
            return reconstruct_path(parent_map, move_map,
current_state.board)

        explored.add(current_state.board)

        for neighbor in current_state.get_neighbors():
            if neighbor.board not in explored and neighbor.board
not in parent_map:
                parent_map[neighbor.board] = current_state.board
                move_map[neighbor.board] = neighbor.path[-1]
                heapq.heappush(frontier, neighbor)

    return None

def reconstruct_path(parent_map, move_map, state):
    path_boards = []
```

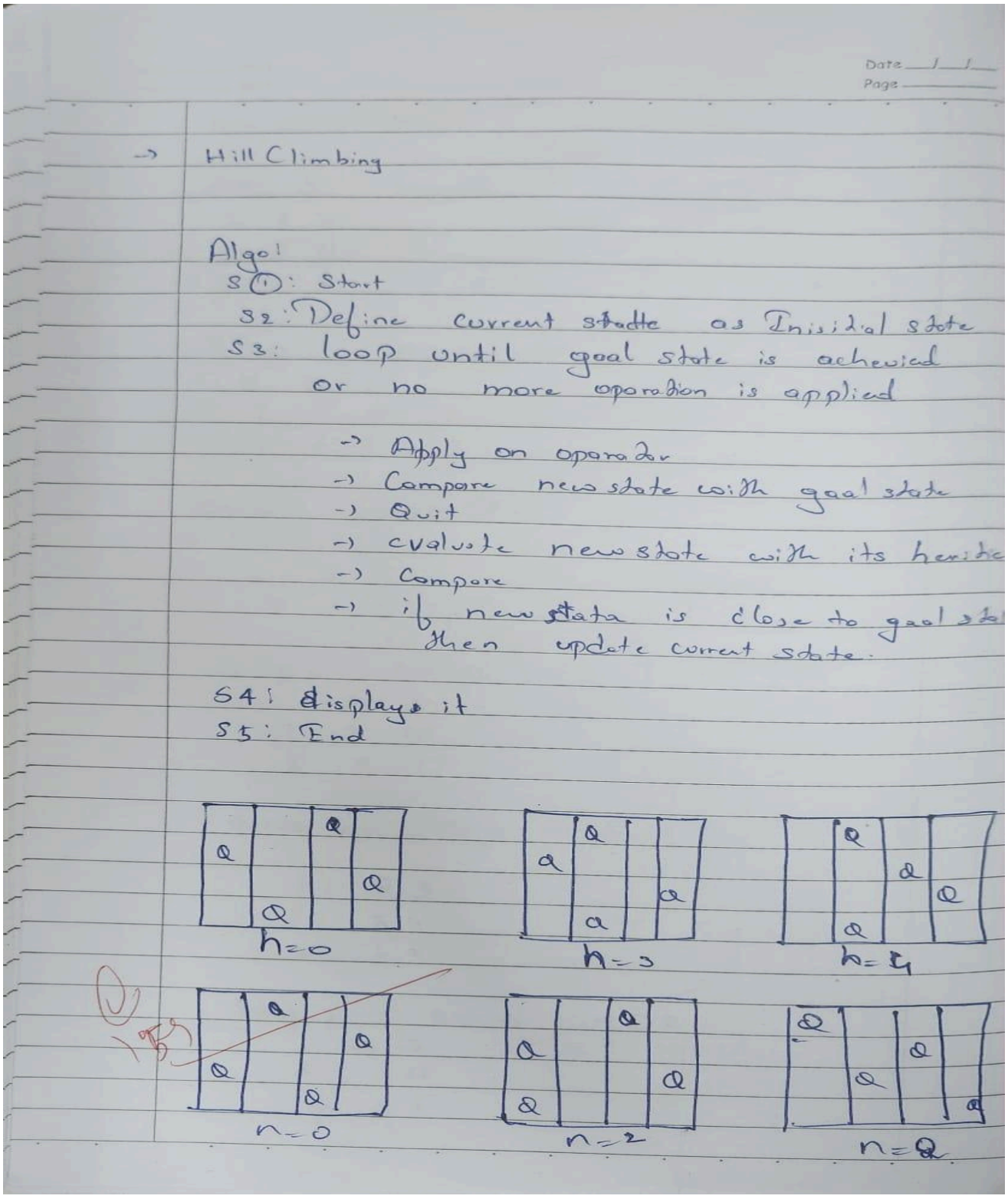
```
    path_moves = []
    while parent_map[state] is not None:
        path_boards.append(state)
        path_moves.append(move_map[state])
        state = parent_map[state]
    path_boards.append(state)
    path_boards.reverse()
    path_moves.reverse()
    return path_boards, path_moves

def print_board(board):
    size = int(len(board) ** 0.5)
    for i in range(size):
        row = board[i*size:(i+1)*size]
        print(" ".join(str(x) if x != 0 else " " for x in row))
    print()

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

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

    result = a_star(initial_state, final_state)
    if result:
        solution_boards, solution_moves = result
        print("Step-by-step solution:\n")
        for step_num, board in enumerate(solution_boards):
            moves_so_far = "".join(solution_moves[:step_num])
            print(f"Step {step_num}: Moves: {moves_so_far}")
            print_board(board)
            time.sleep(1)
    else:
        print("No solution found.")


    print("By \n  Suhas MS\n  1BM23CS346")
```

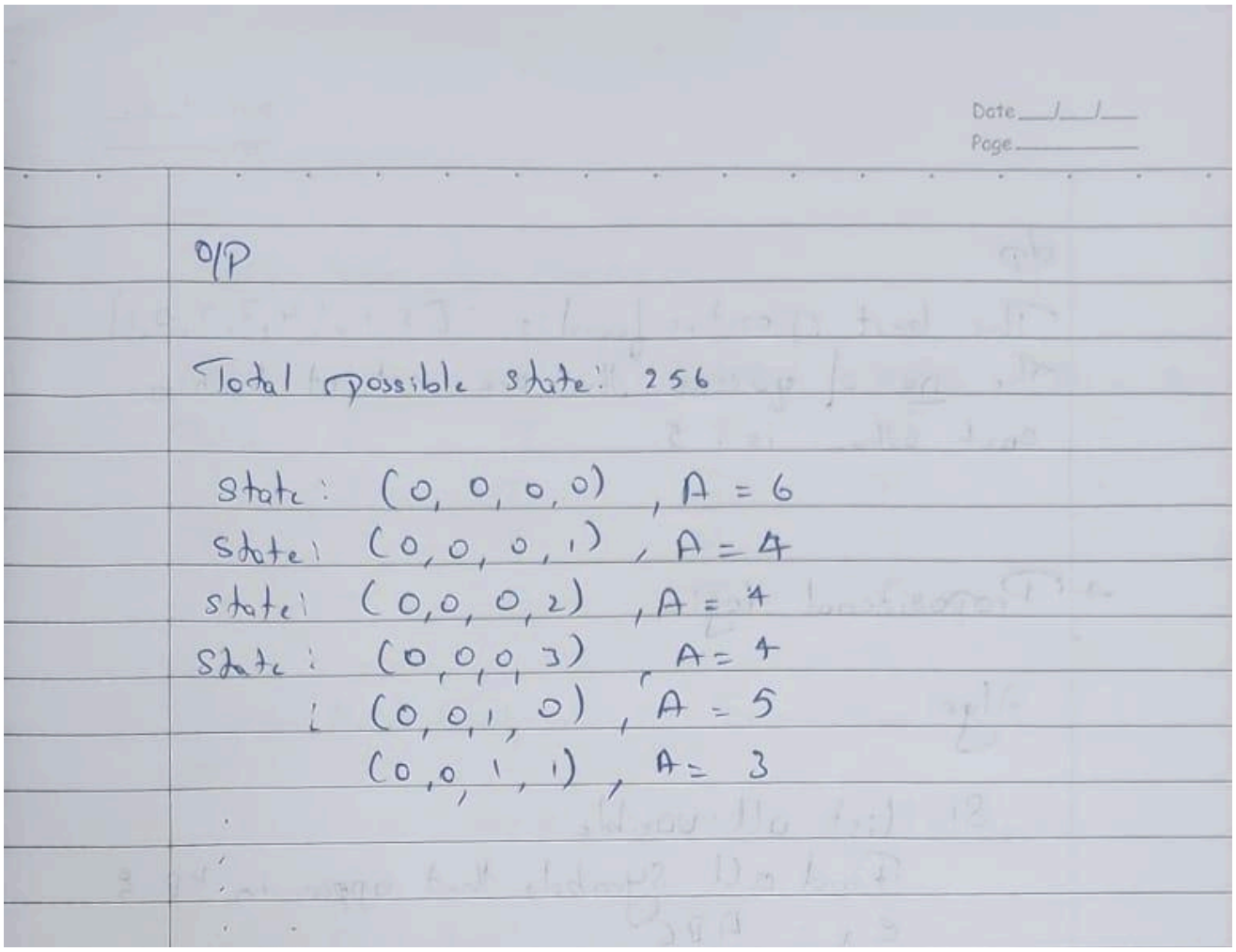
```
Step-by-step solution:

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

Step 1: Moves: U
2 8 3
1   4
7 6 5

Step 2: Moves: UU
2   3
1 8 4
7 6 5
```

```
Step 3: Moves: UUL
  2 3
1 8 4
7 6 5

Step 4: Moves: UULD
1 2 3
  8 4
7 6 5

Step 5: Moves: UULDR
1 2 3
8   4
7 6 5

By
  Suhas MS
  1BM23CS346
```

Program 4:
Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:





Code:

```
def calculate_conflicts(board):
    conflicts = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                conflicts += 1
    return conflicts

def print_board(board):
    n = len(board)
    for row in range(n):
        line = ['Q' if col == board[row] else '.' for col in range(n)]
        print(' '.join(line))
    print()

def hill_climbing_step_by_step(board):
    n = len(board)
    current_state = board[:]
    current_conflicts = calculate_conflicts(current_state)
```

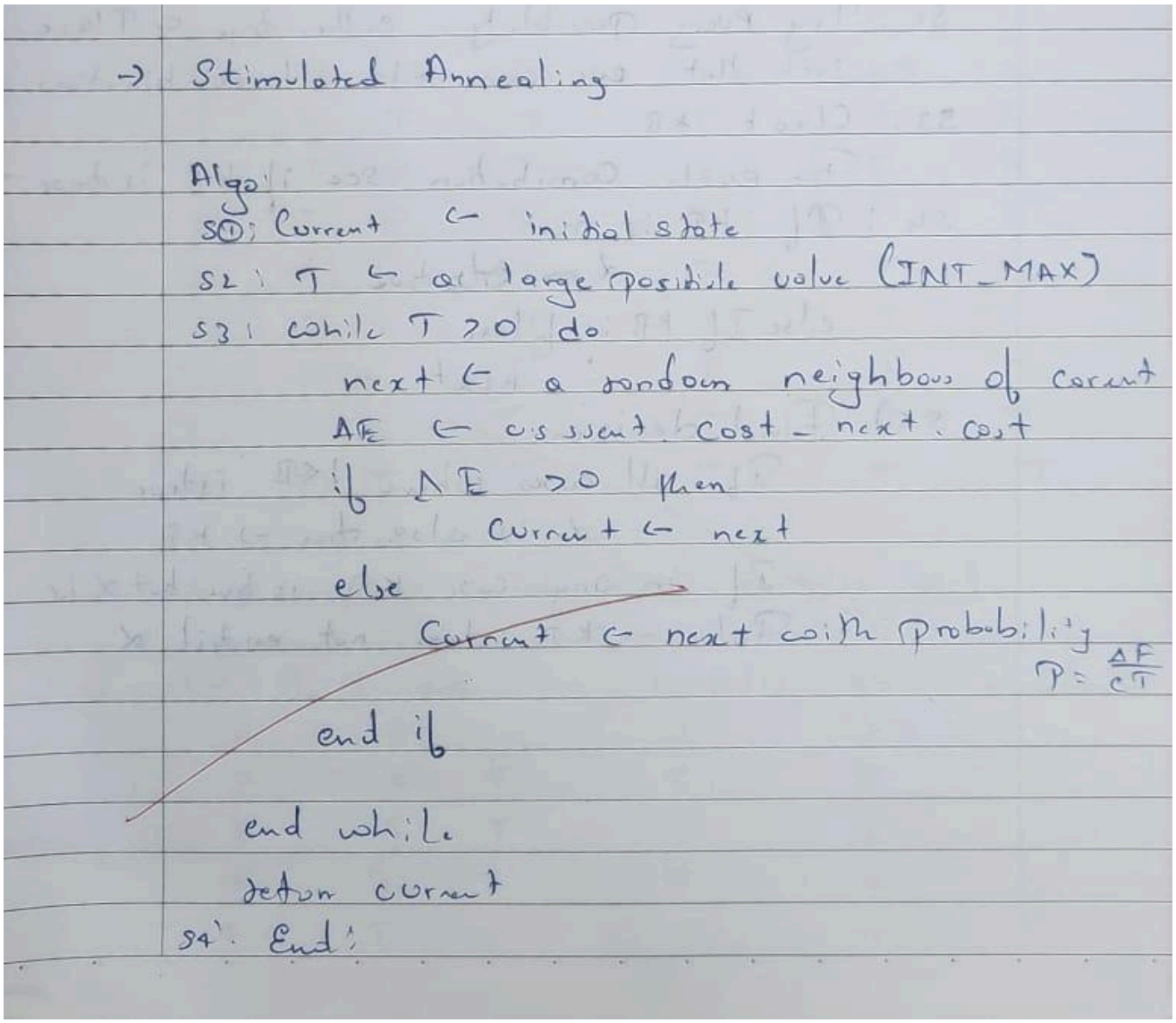
```
    step = 0
    print(f"Initial board with conflicts = {current_conflicts}:")
    print_board(current_state)

    while current_conflicts > 0:
        step += 1
        print(f"Step {step}:")
        best_state = current_state[:]
        best_conflicts = current_conflicts

        for row in range(n):
            original_col = current_state[row]
            for col in range(n):
                if col != original_col:
                    current_state[row] = col
                    conflicts = calculate_conflicts(current_state)

                    if conflicts < best_conflicts:
                        best_conflicts = conflicts
                        best_state = current_state[:]

            current_state[row] = original_col

        if best_conflicts == current_conflicts:
            print("No better neighbor found, stuck at local optimum.")
            break

        current_state = best_state
        current_conflicts = best_conflicts

        print(f"Board with conflicts = {current_conflicts}:")
        print_board(current_state)

    if current_conflicts == 0:
        print("Solution found!")
    else:
        print("No solution found.")
    return current_state

initial_board = [3, 0, 1, 2]
solution = hill_climbing_step_by_step(initial_board)

print("By:\n   Suhas MS\n   1BM23CS346")
```

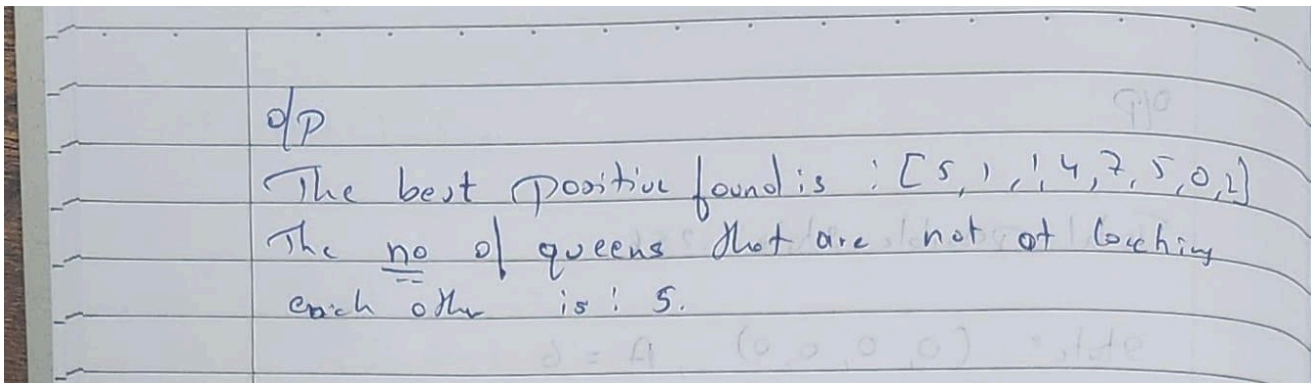
```
Initial board with conflicts = 4:
. . . Q
Q . . .
. Q . .
. . Q .

Step 1:
Board with conflicts = 2:
. . . Q
Q . . .
Q . . .
. . Q .
```

```
Step 2:
Board with conflicts = 1:
. . . Q
. Q . .
Q . . .
. . Q .

Step 3:
No better neighbor found, stuck at local optimum.
No solution found.
By:
   Suhas MS
   1BM23CS346
```

Program 5:
Simulated Annealing to Solve 8-Queens problem

Algorithm:





Code:

```
import random
import math

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()

def calculate_cost(state):
    cost = 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]) == j -
i:
                cost += 1
    return cost

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

def simulated_annealing(initial_state, initial_temp=1000,
cooling_rate=0.95, min_temp=1e-3, max_iter=1000):
    current_state = initial_state
    current_cost = calculate_cost(current_state)
    temperature = initial_temp
    path = [(current_state, current_cost, None)]

    print("Initial State:")
    print_board(current_state)
    print(f"Cost: {current_cost}\n")

    iteration = 0
    while temperature > min_temp and current_cost > 0 and iteration <
max_iter:
        neighbor, swap = get_neighbor(current_state)
        neighbor_cost = calculate_cost(neighbor)

        cost_diff = neighbor_cost - current_cost

        if cost_diff < 0 or math.exp(-cost_diff / temperature) >
random.random():
            current_state, current_cost = neighbor, neighbor_cost
            path.append((current_state, current_cost, swap))
            print(f"Iteration {iteration}: Swap columns {swap}")
            print_board(current_state)
            print(f"Cost: {current_cost}, Temperature:
```

```
{temperature:.4f}\n")

        temperature *= cooling_rate
        iteration += 1

    print("Terminated.")
    return path

def get_initial_state():
    print("Enter the initial positions of the 4 queens (row for each
column, 0-indexed):")
    positions = []
    for col in range(4):
        while True:
            try:
                pos = int(input(f"Column {col}: "))
                if 0 <= pos < 4:
                    positions.append(pos)
                    break
                else:
                    print("Invalid input. Enter a number between 0 and
3.")
            except ValueError:
                print("Invalid input. Please enter an integer.")
    return tuple(positions)

initial_state = get_initial_state()
solution_path = simulated_annealing(initial_state)

print("Final path:")
for i, (state, cost, swap) in enumerate(solution_path):
    print(f"Step {i}:")
    print_board(state)
    print(f"Cost: {cost}")
    if swap is not None:
        print(f"Swap columns: {swap}")
    print("--------------------")


print("By:\n   Suhas MS\n   1BM23CS346")
```

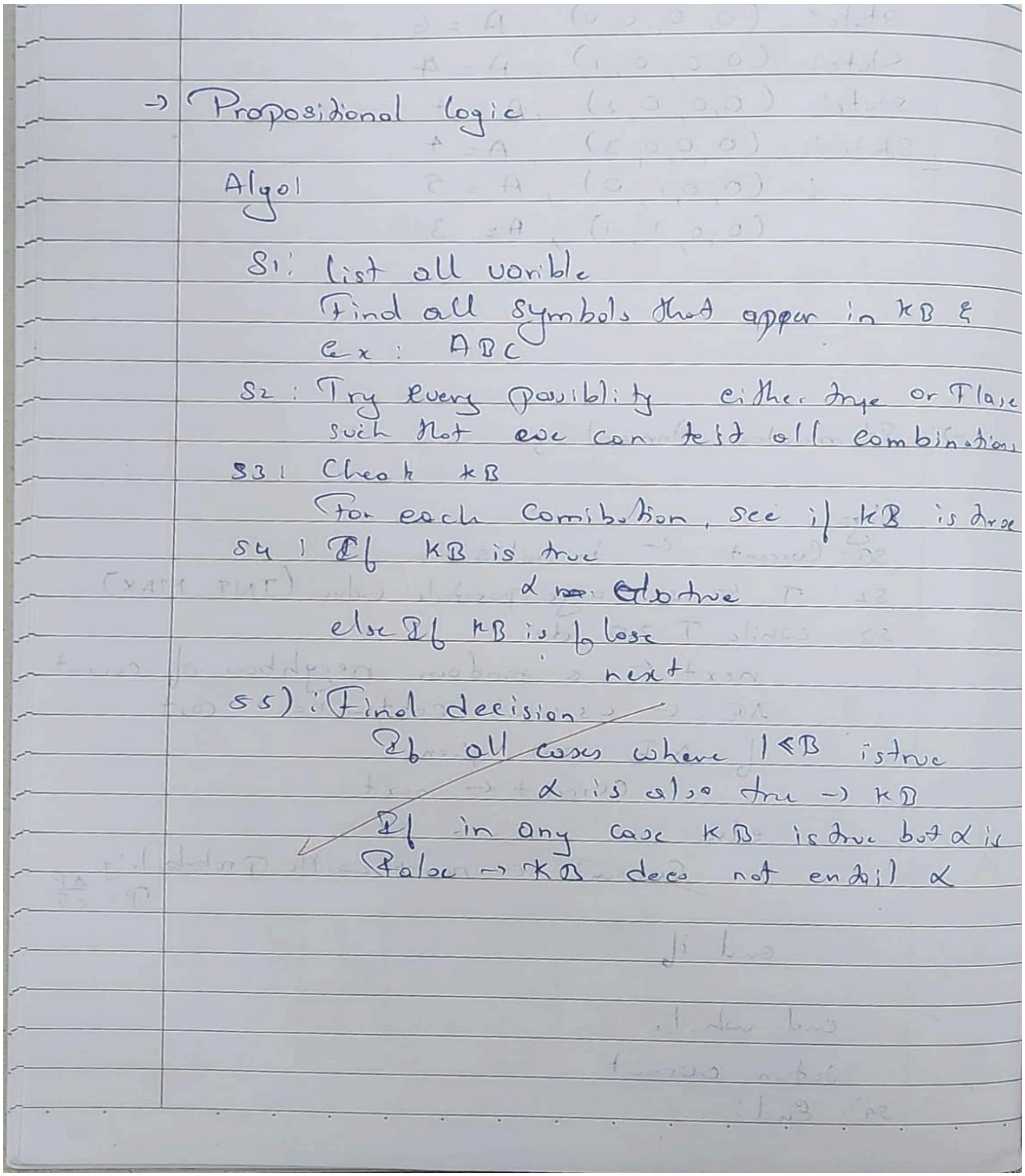
```
Enter the initial positions of the 4 queens (row for each column,
0-indexed):
Column 0: 3
Column 1: 2
Column 2: 1
Column 3: 0
Initial State:
. . . Q
. . Q .
. Q . .
Q . . .

Cost: 6

Iteration 0: Swap columns (2, 1)
. . . Q
```

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

Cost: 2, Temperature: 1000.0000

Iteration 1: Swap columns (1, 2)
. . . Q
. . Q .
. Q . .
Q . . .

Cost: 6, Temperature: 950.0000

Iteration 2: Swap columns (2, 3)
. . Q .
. . . Q
. Q . .
Q . . .

Cost: 2, Temperature: 902.5000

Iteration 3: Swap columns (3, 1)
. . Q .
. Q . .
. . . Q
Q . . .

Cost: 1, Temperature: 857.3750

Iteration 4: Swap columns (0, 1)
. . Q .
Q . . .
. . . Q
. Q . .

Cost: 0, Temperature: 814.5062

Terminated.
Final path:
Step 0:
. . . Q
. . Q .
. Q . .
Q . . .

Cost: 6
--------------------
Step 1:
. . . Q
. Q . .
. . Q .
Q . . .

Cost: 2
Swap columns: (2, 1)
--------------------
Step 2:
```

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

Cost: 6
Swap columns: (1, 2)
--------------------
Step 3:
. . Q .
. . . Q
. Q . .
Q . . .

Cost: 2
Swap columns: (2, 3)
--------------------
Step 4:
. . Q .
. Q . .
. . . Q
Q . . .

Cost: 1
Swap columns: (3, 1)
--------------------
Step 5:
. . Q .
Q . . .
. . . Q
. Q . .

Cost: 0
Swap columns: (0, 1)
--------------------
By:
   Suhas MS
   1BM23CS346
```

Program 6:
Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

Algorithm:



Date / /
Page

Truth table for connectives

P	Q	~R	P∧Q	P∨Q	P⇒Q
False	F	T	F	F	T
F	T	T	F	T	F
T	F	F	F	T	F
T	T	F	T	T	T

Propositional Inference: Enumeration Method

Ex: $\alpha = A \vee B$ $KB = (A \vee C) \wedge (B \vee C)$

Checking that $KB \models \alpha$

A	B	C	A∨C	B∨C	KB	α
F	F	F	F	T	F	F
F	F	T	T	T	T	T
F	T	F	F	T	F	F
F	T	T	T	T	T	T
T	F	F	T	T	T	T
T	F	T	T	T	T	T
T	T	F	T	T	T	T
T	T	T	T	T	T	T

$KB \models \alpha$ holds (KB entails α)

O/P Truth table For $\alpha = A \vee B$, $KB = (A \vee B) \wedge (B \vee C)$

A	B	C	α	KB
F	F	F	F	F
F	F	T	F	F
F	T	F	T	F
F	T	T	T	T

Consider S & T as variable

a : $+(S \vee T)$
b : $(S \wedge T)$
c : $T \vee \neg T$

i) a entails b
ii) b entails c

S	T	a	b	c
T	T	T	T	T
T	F	T	F	T
F	T	T	F	T
F	F	F	F	T

(i) As Row 2 $a = T$; $b = F$
And Row 3 $a = T$; $b = F$
∴ We can conclude a doesn't entails b

(ii) a ⊨ c
a ⊭ c
c is tautology
∴ Yes we can conclude a entails c.

22/9/22

Code:

```
import itertools
import pandas as pd
import re

def replace_implications(expr):
    pattern = r'([^=><]+?)\s*=>\s*([^=><]+?)(?=\s|$|[&|)])'
    while re.search(pattern, expr):
        expr = re.sub(pattern,
                      lambda m: f"(not {m.group(1).strip()} or
{m.group(2).strip()})",
                      expr,
                      count=1)
    return expr

def pl_true(sentence, model):
    expr = sentence.strip()
    expr = expr.replace("<=>", "==")
    expr = replace_implications(expr)
    for sym, val in model.items():
        expr = re.sub(rf'\b{sym}\b', str(val), expr)
    expr = re.sub(r'\s+', ' ', expr)
    expr = expr.replace(" and ", " and ").replace(" or ", " or ").replace(" not ",
" not ")

    return eval(expr)

def get_symbols(KB, alpha):
    symbols = set()
    for sentence in KB + [alpha]:
        for token in re.findall(r'\b[A-Za-z]+\b', sentence):
            if token not in ['and', 'or', 'not']:
                symbols.add(token)
    return sorted(list(symbols))

def tt_entails(KB, alpha):
    symbols = get_symbols(KB, alpha)
    rows = []
    entails = True

    for values in itertools.product([True, False], repeat=len(symbols)):
        model = dict(zip(symbols, values))

        try:
            kb_val = all(pl_true(sentence, model) for sentence in KB)
            alpha_val = pl_true(alpha, model)

            rows.append({**model, "KB": kb_val, "alpha": alpha_val})

            if kb_val and not alpha_val:
                entails = False
        except Exception as e:
            print(f"Error evaluating with model {model}: {e}")
            return False

    df = pd.DataFrame(rows)
    print("\n" + "="*50)
```

```
    print("                TRUTH TABLE")
    print("="*50)

    col_widths = {}
    for col in df.columns:
        col_widths[col] = max(len(str(col)), df[col].astype(str).str.len().max())

    table_width = sum(col_widths.values()) + len(df.columns) * 3 - 1
    print("┌" + "─" * table_width + "┐")

    header = "│"
    for col in df.columns:
        header += f" {col:^{col_widths[col]}} │"
    print(header)
    separator = "├"
    for col in df.columns:
        separator += "─" * (col_widths[col] + 2) + "┼"
    separator = separator[:-1] + "┤"
    print(separator)
    for _, row in df.iterrows():
        row_str = "│"
        for col in df.columns:
            value = str(row[col])
            row_str += f" {value:^{col_widths[col]}} │"
        print(row_str)
    print("└" + "─" * table_width + "┘")
    print("\n" + "="*50)
    result_text = f"KB ENTAILS ALPHA: {'✓ YES' if entails else '✗ NO'}"
    print(f"{result_text:^50}")
    print("="*50)
    return entails
print("Enter Knowledge Base (KB) sentences, separated by commas.")
print("Use symbols like A, B, C and operators: and, or, not, =>, <=>")
kb_input = input("KB: ").strip()
KB = [x.strip() for x in kb_input.split(",")]
alpha = input("Enter query (alpha): ").strip()
result = tt_entails(KB, alpha)
print(f"Result: {result}")
print("By:\n   Suhas MS\n   1BM23CS346")
```

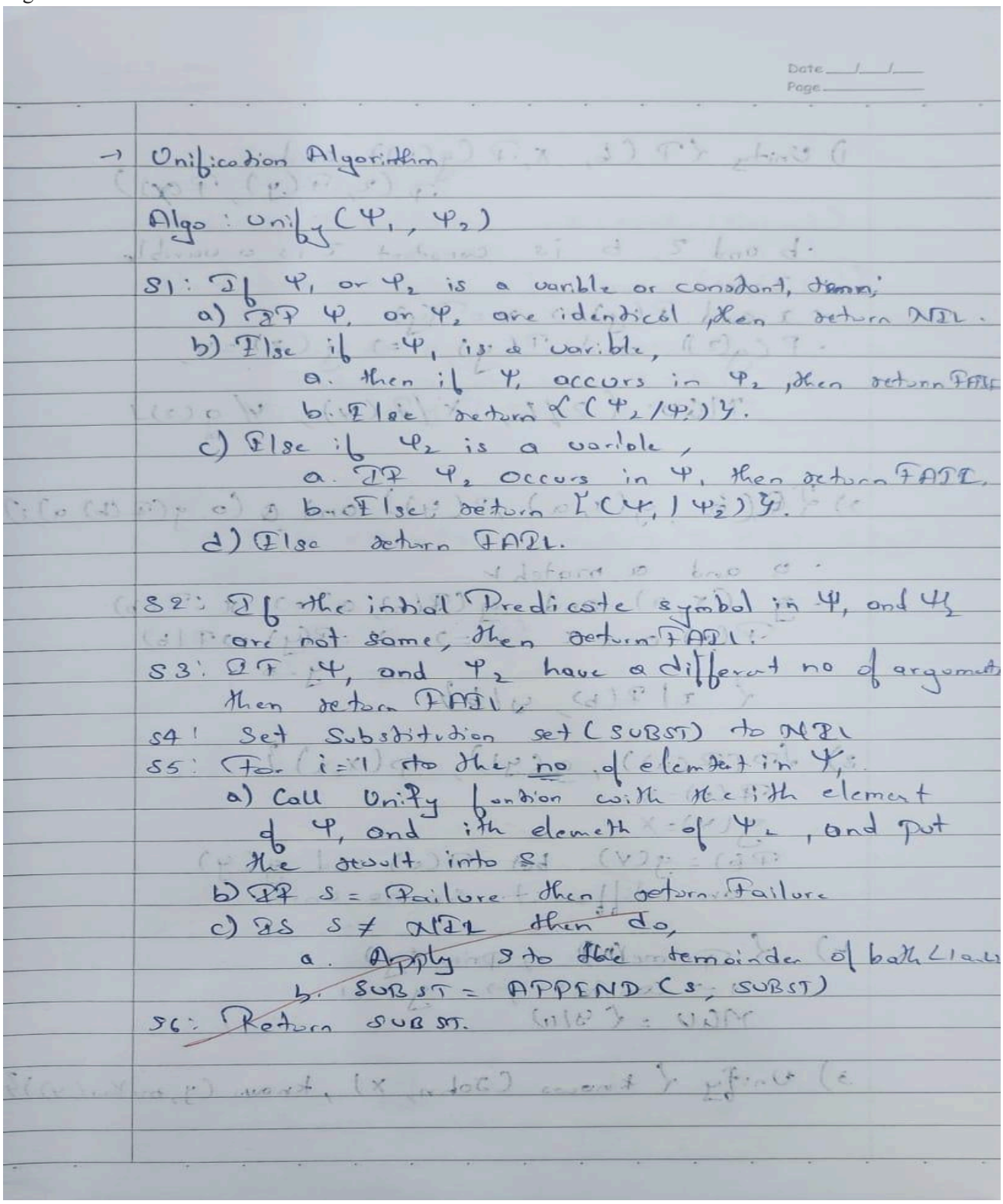
```
Enter Knowledge Base (KB) sentences, separated by commas.
Use symbols like A, B, C and operators: and, or, not, =>, <=>
KB: not (S or T)
Enter query (alpha): T or (not T)
```

S	T	KB	alpha
True	True	False	True
True	False	False	True
False	True	False	True
False	False	True	True

```
Result: True
By:
   Suhas MS
   1BM23CS346
```

Program 7:
Implement unification in first order logic

Algorithm:



Date / /
Page

1) Unify { P(b, x, F(g(z)) and P(z, F(y), F(x)) }

- b and z, b is constant z is a varible z = b
- x and F(y) → x = F(y)
- F(g(z)) and F(y) ⇒ g(z) = y

MGU: { z/b, x/F(y), y/g(z) }

2) { Q(a, g(x, a), f(y)) and Q(a, g(f(b), a), x) }

- a and a match
- g(x, a) and g(f(b), a) ⇒ x = f(b)
  f(y) and x ⇒ x = f(y) ⇒ f(b)
  y = b

{ x/f(b), y/b }

3) { P(F(a)), g(y), P(x, x) }

F(a) = x

g(y) = x

F(b) = g(y) but F(a) and g(y) are different so no unify.

4) { Prime(11), Prime(y) }

11 → constant ⇒ y = 11

MGU = { y/11 }

5) Unify { knows(John, x), knows(y, mother(y)

Date / /
Page

John = y
x = mother(y)
MGU = {y / John, x / mother(John)}

6) {knows(John, x), knows(y, Bill)}

First argument: John = y
Second argument: x = Bill
MGU = {y / John; x / Bill}

o/p

{P(b, x, f(g(z))), and P(z, f(y)), P(y)}
MGU:

z/b
~~x/{P(f(y), y)}~~
x/{P(y, z)}

Code:

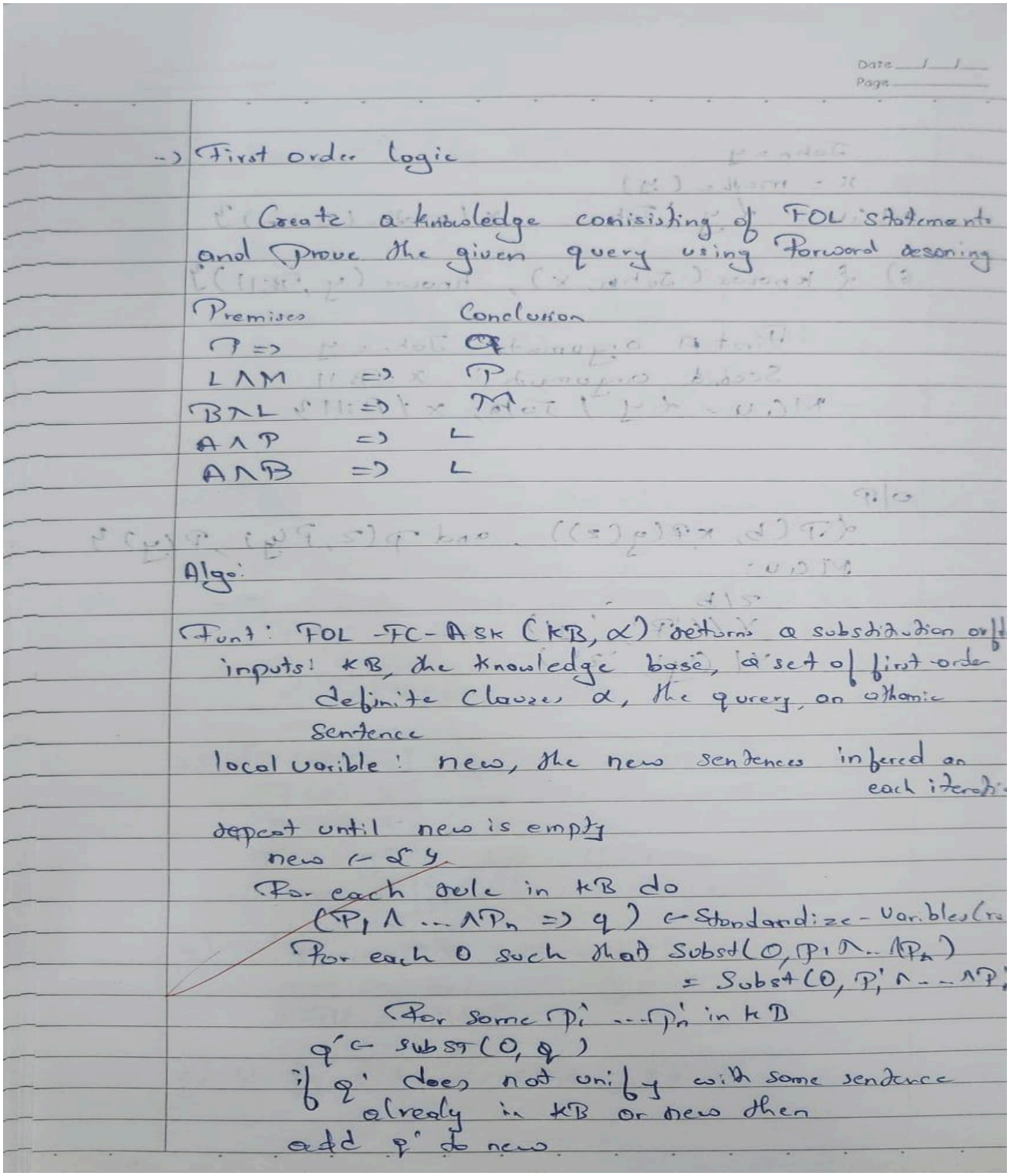
```
def is_variable(x):
    # Variables are single uppercase letters only, like 'X', 'Y',
'Z'
    return isinstance(x, str) and len(x) == 1 and x.isupper()

def is_compound(x):
    return isinstance(x, tuple) and len(x) > 0

def occurs_check(var, term, subst):
    if var == term:
        return True
    elif is_variable(term) and term in subst:
        return occurs_check(var, subst[term], subst)
    elif is_compound(term):
        return any(occurs_check(var, t, subst) for t in term[1:])
    return False

def unify(x, y, subst=None):
    if subst is None:
        subst = {}
    if x == y:
        return subst
    if is_variable(x):
        return unify_var(x, y, subst)
    elif is_variable(y):
        return unify_var(y, x, subst)
    elif is_compound(x) and is_compound(y):
        if x[0] != y[0] or len(x) != len(y):
            return None
        for x_arg, y_arg in zip(x[1:], y[1:]):
            subst = unify(x_arg, y_arg, subst)
            if subst is None:
                return None
        return subst
    else:
        return None

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

def print_substitution(subst):
    if subst is None:
        print("No unifier exists")
    else:
        print("MGU:")
        for var, val in subst.items():
            print(f"{var} / {val}")
```

```
# Terms:
expr1 = ('p', 'b', 'X', ('f', ('g', 'Z')))
expr2 = ('p', 'Z', ('f', 'Y'), ('f', 'Y'))

result = unify(expr1, expr2)
print_substitution(result)

print("By:\n   Suhas MS\n   1BM23CS346")
```

```
MGU:
Z / b
X / ('f', 'Y')
Y / ('g', 'Z')
By:
   Suhas MS
   1BM23CS346
```

Program 8:
Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

Algorithm:



-> First order logic

Create a knowledge consisting of FOL statements and Prove the given query using forward desoning

Premises		Conclusion
$\neg$	=>	Q
$L \wedge M$	=>	P
$B \wedge L$	=>	M
$A \wedge P$	=>	L
$A \wedge B$	=>	L

Algo:

Funt: FOL -FC- Ask (KB, α) returns a substitution or f
inputs: KB, the knowledge base, a set of first order definite Clauses α, the query, an atomic sentence
local varible: new, the new sentences inferred on each iterati

repeat until new is empty
new $\leftarrow$ {}
For each rule in KB do
$(P_1 \wedge \dots \wedge P_n \Rightarrow q) \leftarrow$ Standardize-Variables(ru
For each θ such that Subst(θ, $P_1 \wedge \dots \wedge P_n$)
= Subst(θ, $P_1' \wedge \dots \wedge P_n'$
For some $P_1' \dots P_n'$ in KB
$q' \leftarrow$ Subst(θ, q)
if q' does not unify with some sentence 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

Problem:

As per law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, and all the missiles were sold to it by Robert, who is an American citizen"

Prove that "Robert is criminal".

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

let's say P, Q, and r are variables

$American(P) \wedge weapon(Q) \wedge sells(P, Q, r) \wedge Hostile(r) \Rightarrow Criminal(P)$

Country A has some missiles

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

Existential Instantiation, introducing a new constant T_1

$Owns(A, T_1)$

$Missile(T_1)$

All of the missile were sold to country A by Robert

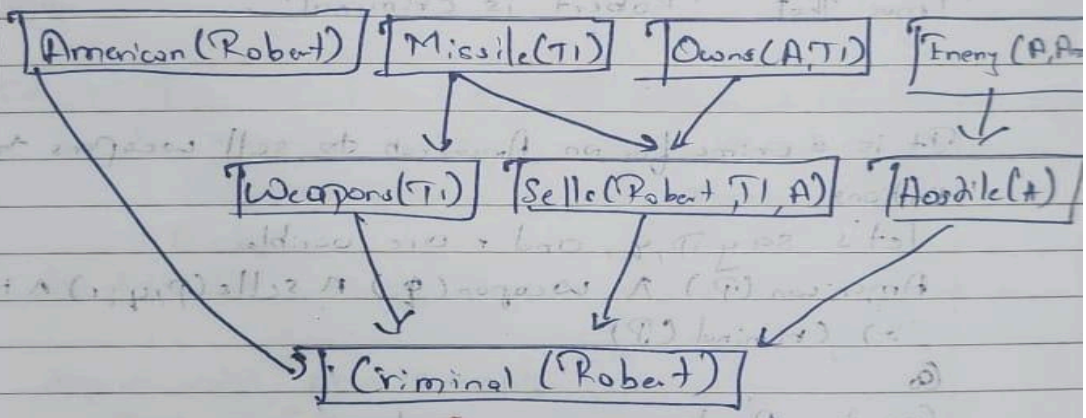
$\forall x \; Missiles(x) \wedge Owns(A, x) \Rightarrow Sells(Robert, x, A)$

Missiles are Weapons
Missile(x) => Weapon(x)

Enemy of America is known as hostile
∀x Enemy(x, America) => Hostile(x)

Robert is an American
American(Robert)

The country A, an enemy of America
Enemy(A, America)



American(p) ∧ Weapon(q) ∧ Sells(p,q,r) ∧
Hostile(r) => Criminal(p)

O/P: Inffered: Hostile(Country A)
Inffered: Weapon(M1)
Inffered: Criminal(Robert)
Goal: reached
Robert is criminal proved.

13/10/25

Code:

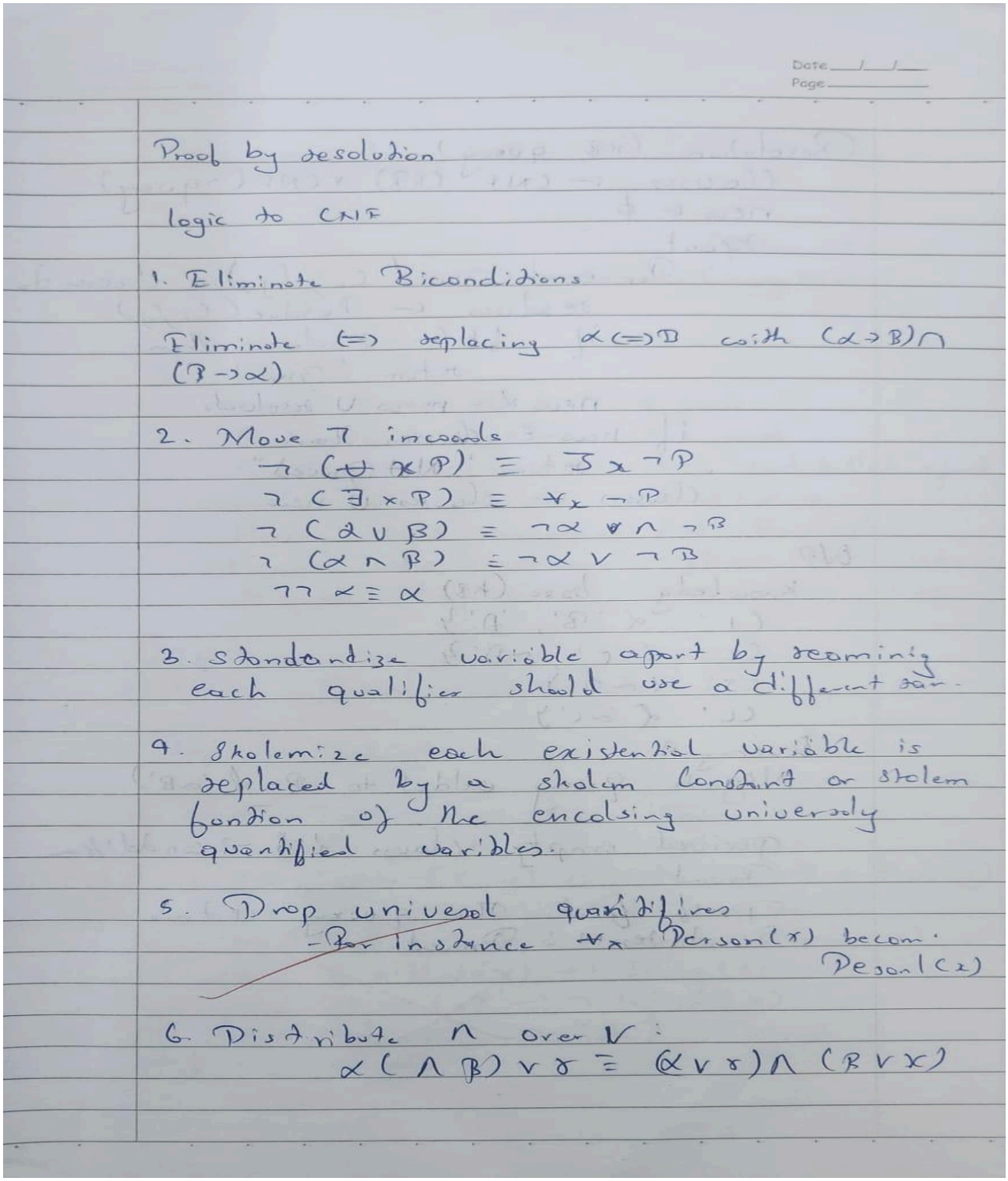
```
facts = {
    "American(Robert)",
    "Enemy(CountryA, America)",
    "Missile(M1)",
    "Has(CountryA, M1)",
    "Sold(Robert, CountryA, M1)"
}

rules = [
    ("Enemy(X, America)", "Hostile(X)"),
    ("American(Robert) & Sold(Robert, Y, Z) & Hostile(Y) &
Weapon(Z)", "Criminal(Robert)"),
    ("Missile(Z)", "Weapon(Z)")
]

import re

def forward_chaining(facts, rules, goal):
    new_facts = set(facts)
    while True:
        added = False
        for premise, conclusion in rules:
            premises = [p.strip() for p in premise.split('&')]
            substitutions = [{}]
            for p in premises:
                new_substitutions = []
                for sub in substitutions:
                    p_inst = substitute(p, sub)
                    matches = match_fact(p_inst, new_facts)
                    for m in matches:
                        merged = merge_substitutions(sub, m)
                        if merged is not None:
                            new_substitutions.append(merged)
                substitutions = new_substitutions
            for sub in substitutions:
                conclusion_inst = substitute(conclusion, sub)
                if conclusion_inst not in new_facts:
                    new_facts.add(conclusion_inst)
                    print(f"Inferred: {conclusion_inst}")
                    added = True
                    if conclusion_inst == goal:
                        print("Goal reached!")
                        return True
        if not added:
            break
    return goal in new_facts

def substitute(expr, sub):
    for var, val in sub.items():
        expr = re.sub(r'\b' + re.escape(var) + r'\b', val, expr)
    return expr

def match_fact(premise, facts):
    pattern = re.compile(r"(\w+)")
    m = pattern.match(premise)
    if not m:
```

```
        return []
    pred = m.group(1)
    args = [a.strip() for a in m.group(2).split(",")]
    matches = []
    for f in facts:
        fm = pattern.match(f)
        if not fm:
            continue
        fpred = fm.group(1)
        fargs = [a.strip() for a in fm.group(2).split(",")]
        if fpred != pred or len(fargs) != len(args):
            continue
        sub = {}
        failed = False
        for a, fa in zip(args, fargs):
            if is_variable(a):
                if a in sub and sub[a] != fa:
                    failed = True
                    break
                sub[a] = fa
            else:
                if a != fa:
                    failed = True
                    break
        if not failed:
            matches.append(sub)
    return matches

def merge_substitutions(s1, s2):
    s = s1.copy()
    for k, v in s2.items():
        if k in s and s[k] != v:
            return None
        s[k] = v
    return s

def is_variable(x):
    return bool(re.match(r'^[A-Z][A-Za-z0-9_]*$', x))

goal = "Criminal(Robert)"

if forward_chaining(facts, rules, goal):
    print("Robert is criminal: Proven")
else:
    print("Could not prove Robert is criminal")

print("By:\n   Suhas MS\n   1BM23CS346")
```

```
Inferred: Hostile(CountryA)
Inferred: Weapon(M1)
Inferred: Criminal(Robert)
Goal reached!
Robert is criminal: Proven
By:
   Suhas MS
   1BM23CS346
```

Program 9:
Create a knowledge base consisting of first order logic statements and prove the given query using Resolution

Algorithm:



Proof by resolution

logic to CNF

1. Eliminate Biconditions

Eliminate $\Leftrightarrow$ replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$

2. Move $\neg$ inwards

$\neg (\forall x\, P) \equiv \exists x \neg P$

$\neg (\exists x\, P) \equiv \forall x \neg P$

$\neg (\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$

$\neg (\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$

$\neg\neg\alpha \equiv \alpha$

3. Standardize variable apart by renaming each qualifier should use a different var.

4. Skolemize each existential variable is replaced by a skolem constant or skolem fonction of the enclosing universaly quantified variables.

5. Drop universal quantifiers

~~For Instance~~ $\forall x$ Person(x) becom: Person(x)

6. Distribute $\wedge$ over $\vee$:

$\alpha (\wedge \beta) \vee \gamma \equiv (\alpha \vee \gamma) \wedge (\beta \vee \chi)$

Date / /
Page

Resolution (KB, query)
clauses ← CNF (KB) ∧ CNF (¬query)
new ← ϕ
repeat
For each pair (Ci, Cj) in clauses
resolves ← Resolve (Ci, Cj)
if ϕ ∈ resolves then
return "Proved"
new ← new ∪ resolvents
if new ⊆ clauses then
return "Not Proved"
clauses ← clauses ∪ new

Q&P
Knowledge base (KB)
C1 : {'A', 'B'}
C2 : {'C', '~A'}
C3 : {'~B'}
C4 : {'~C'}
Query A
Negate query added to B {'~B'}

Derived empty clause {} → condition found
Proof found. The query is true
Final result : Proved.

Date __/__/__
Page _____

Prof by Resln that John likes Peanuts

Rep in FOL

(a) $\forall_x$ Food(x) → likes (John, x)

(b) Food (Apple) ∧ Food (Vegitable)

Proof By Resoln

John likes all kind of food

Apple and vegetable are good

Anything anyone eats and not killed is food

Anil eats Peanuts and still alive

Harry eats everything that Anil eats

Anyone who is alive implies not killed

Anyone who is not killed implies alive

To Prove

John likes Peanuts

Rep in FOL

$\forall_x$ Food (x) → likes (John, x)

Food (Apple) ∧ Food (Vegitable)

$\forall_x$ y eats (x, y) ∧ ¬ killed (x) → Food (y)

eats (Anil, Peanuts) ∧ Alive (Anil)

$\forall_x$: eats (Anil, x) → eats (Harry, x)

$\forall_x$ ¬ killed (x) → Alive (x)

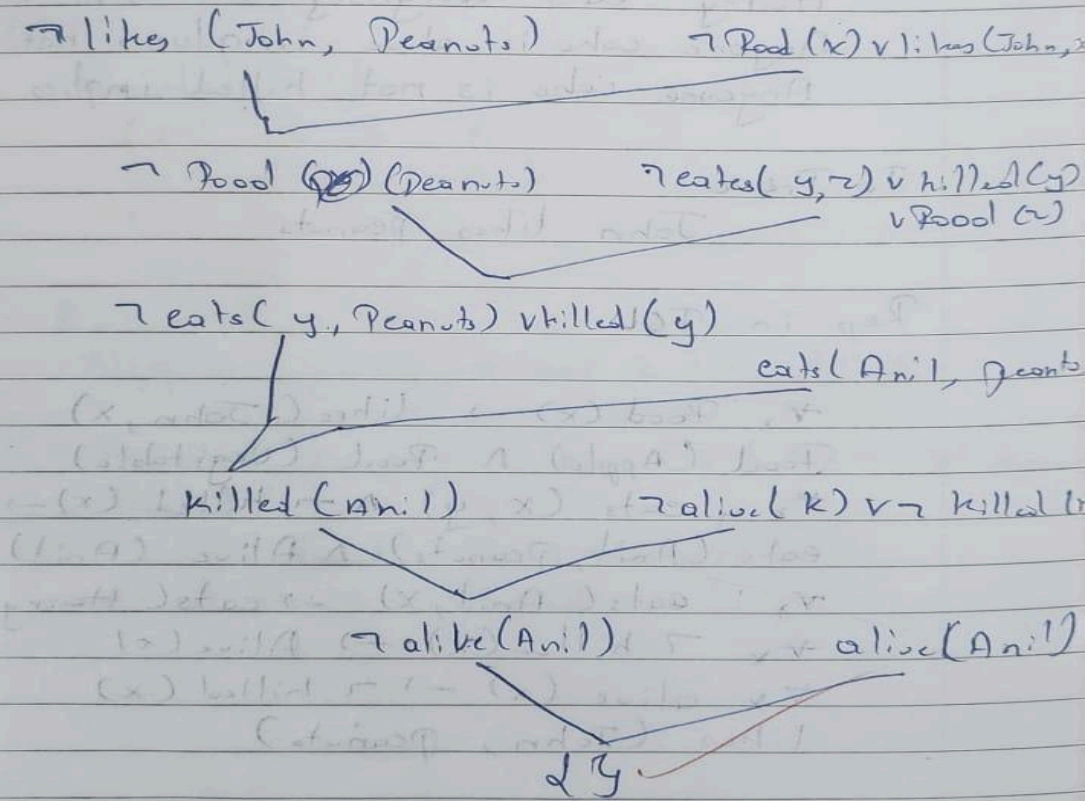
$\forall_x$ alive (x) → ¬ killed (x)

likes (John, Peanuts)

Date __/__/__
Page _____

FOL to CNF

¬ Food (x) ∨ likes (John, x)
Food (Apple)
Food (Vegetable)
¬ eats (y, z) ∨ killed (y) ∨ Food (z)
eats (Anil, Peanuts)
alive (Anil)
¬ eats (Anil, w) ∨ eats (Harry, w)
killed (g) ∨ alive (g)
¬ alive (k) ∨ ¬ killed (k)



Hence proved.

Code:

```
import re
import itertools

VAR_RE = re.compile(r'^[a-z](_\d+)?$')   # single-letter variable optionally
standardized (x_0, y_3)

def is_variable(token: str) -> bool:
    return bool(VAR_RE.fullmatch(token))

def parse_literal(text):
    text = text.strip()
    neg = False
    if text.startswith('¬') or text.startswith('~'):
        neg = True
        text = text[1:].strip()
    if '(' in text:
        pred = text[:text.index('(')].strip()
        args = [a.strip() for a in text[text.index('(')+1:-1].split(',')]
    else:
        pred = text
        args = []
    return {'neg': neg, 'pred': pred, 'args': args}

def clause_to_str(clause):
    if clause == []:
        return '⊥'
    parts = []
    for lit in clause:
        s = ('¬' if lit['neg'] else '') + (lit['pred'] + '(' + ',
'.join(lit['args']) + ')' if lit['args'] else lit['pred'])
        parts.append(s)
    return ' ∨ '.join(parts)

def standardize_apart_clause(clause, idx):
    # only rename variables (single-letter) to var_index form
    mapping = {}
    new_clause = []
    for lit in clause:
        new_args = []
        for a in lit['args']:
            if is_variable(a):
                if a not in mapping:
                    mapping[a] = f"{a}_{idx}"
                new_args.append(mapping[a])
            else:
                new_args.append(a)
        new_clause.append({'neg': lit['neg'], 'pred': lit['pred'], 'args':
new_args})
    return new_clause

# ----- Unification for flat args (no nested function terms) -----
def occurs_check(var, val, subs):
    # var and val are token strings
    if var == val:
        return True
```

```
    if is_variable(val) and val in subs:
        return occurs_check(var, subs[val], subs)
    return False

def apply_subs_token(tok, subs):
    if is_variable(tok):
        while tok in subs:
            tok = subs[tok]
        return tok
    return tok

def apply_subs_literal(lit, subs):
    new_args = [apply_subs_token(a, subs) for a in lit['args']]
    return {'neg': lit['neg'], 'pred': lit['pred'], 'args': new_args}

def unify_tokens(x, y, subs):
    # x,y are token strings (variables or constants)
    if x == y:
        return subs
    if is_variable(x):
        if x in subs:
            return unify_tokens(subs[x], y, subs)
        if occurs_check(x, y, subs):
            return None
        new = subs.copy()
        new[x] = y
        return new
    if is_variable(y):
        return unify_tokens(y, x, subs)
    # both constants and different => fail
    return None

def unify_arg_lists(a_list, b_list):
    if len(a_list) != len(b_list):
        return None
    subs = {}
    for a, b in zip(a_list, b_list):
        a_ap = a if not is_variable(a) else a
        b_ap = b if not is_variable(b) else b
        subs = unify_tokens(apply_subs_token(a_ap, subs), apply_subs_token(b_ap,
subs), subs)
        if subs is None:
            return None
    return subs

# ----- Resolution -----
def is_tautology_clause(clause):
    # clause is a list of literals (after substitution). If it contains A and ¬A
same args -> tautology
    seen = {}
    for lit in clause:
        key = (lit['pred'], tuple(lit['args']))
        if key in seen:
            if seen[key] != lit['neg']:
                return True
        else:
            seen[key] = lit['neg']
    return False
```

```
def resolve_pair(c1, c2):
    # c1, c2 are lists of literals (each literal dict)
    for i, l1 in enumerate(c1):
        for j, l2 in enumerate(c2):
            if l1['pred'] == l2['pred'] and l1['neg'] != l2['neg']:
                # try to unify their args
                subs = unify_arg_lists(l1['args'], l2['args'])
                if subs is None:
                    continue
                # apply substitution to the remainder of both clauses
                new_clause = []
                for k, lit in enumerate(c1):
                    if k == i: continue
                    new_clause.append(apply_subs_literal(lit, subs))
                for k, lit in enumerate(c2):
                    if k == j: continue
                    new_clause.append(apply_subs_literal(lit, subs))
                # remove duplicates (syntactic)
                uniq = []
                for lit in new_clause:
                    if not any(lit['pred']==u['pred'] and lit['neg']==u['neg'] and
lit['args']==u['args'] for u in uniq):
                        uniq.append(lit)
                if is_tautology_clause(uniq):
                    continue
                return uniq, subs, (i, j)
    return None, None, None

# ----- Build derivation tree nodes -----
class Node:
    def __init__(self, clause, parents=None, label=None):
        self.clause = clause
        self.parents = parents if parents else []
        self.label = label

def resolution_with_tree(initial_clauses, goal_clause):
    # standardize apart initial clauses
    clauses_nodes = []
    for idx, c in enumerate(initial_clauses):
        std = standardize_apart_clause(c, idx)
        clauses_nodes.append(Node(std, parents=[] , label=f"C{idx}"))

    # add negated goal as a fresh clause (standardize apart too)
    neg_goal = []
    # goal_clause is a clause list (we take its first literal if single-literal
goal)
    for lit in goal_clause:
        # negate each literal in goal clause (if goal is a single positive
literal user passed)
        neg_goal.append({'neg': not lit['neg'], 'pred': lit['pred'], 'args':
lit['args'][:]})
    neg_goal_std = standardize_apart_clause(neg_goal, len(clauses_nodes))
    goal_node = Node(neg_goal_std, parents=[], label="¬Goal")
    clauses_nodes.append(goal_node)

    # mapping from index -> Node
    idx = len(clauses_nodes)
```

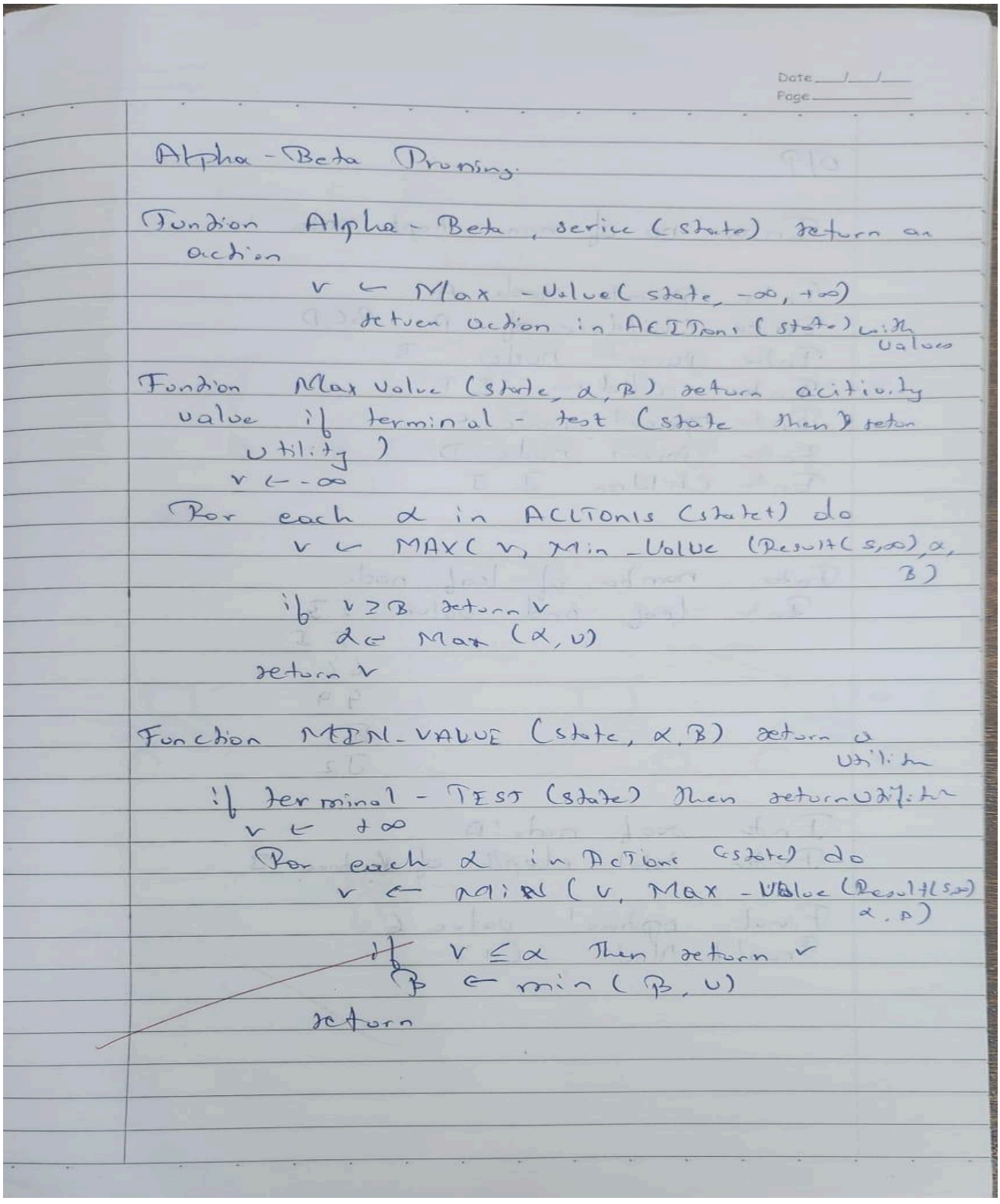
```
    seen_clauses = {clause_to_str(n.clause): i for i, n in
enumerate(clauses_nodes)}

    # perform breadth-first-ish resolution (pairwise), record parents as indices
    for a_index in range(len(clauses_nodes)):
        pass  # placeholder, we'll use dynamic loop below

    frontier_changed = True
    while True:
        new_added = False
        # iterate pairs over current clauses
        n = len(clauses_nodes)
        pairs = [(i,j) for i in range(n) for j in range(i+1, n)]
        for i,j in pairs:
            c1 = clauses_nodes[i].clause
            c2 = clauses_nodes[j].clause
            resolvent, subs, which = resolve_pair(c1, c2)
            if resolvent is None:
                continue
            s = clause_to_str(resolvent)
            if s in seen_clauses:
                continue
            # add node
            new_node = Node(resolvent, parents=[i, j], label=f"R{idx}")
            clauses_nodes.append(new_node)
            seen_clauses[s] = idx
            new_added = True
            idx += 1
            if resolvent == []:
                # build bottom-up tree node for ⊥
                root = new_node
                return clauses_nodes, seen_clauses, idx-1  # return nodes, map,
index of empty clause node
        if not new_added:
            return clauses_nodes, seen_clauses, None

# ----- ASCII print bottom-up (root bottom) -----
def print_bottom_up_tree(nodes, root_index):
    # recursively print node; ensure parents printed above
    def recurse(node_index, prefix="", is_last=True):
        node = nodes[node_index]
        connector = "└── " if is_last else "├── "
        print(prefix + connector + clause_to_str(node.clause))
        # if this node has parents, print them above (parents as children in
recursion so they appear above)
        parents = node.parents
        for k, pidx in enumerate(parents):
            recurse(pidx, prefix + ("    " if is_last else "│   "), k ==
len(parents)-1)
    recurse(root_index, "", True)

# ----- Runner -----
if __name__ == "__main__":
    print("="*70)
    print("FIRST-ORDER LOGIC RESOLUTION SYSTEM (FIXED)")
    print("="*70)
    print("Enter CNF clauses (one per line). End with a blank line.")
    raw = []
```

```
    while True:
        try:
            line = input().strip()
        except EOFError:
            break
        if line == "":
            break
        raw.append(line)
    clauses = [ [parse_literal(tok.strip()) for tok in re.split(r"∨", line) ] for
line in raw ]

    # read goal
    goal_line = input("\nEnter GOAL clause (single literal form): ").strip()
    goal_clause = [parse_literal(goal_line)]

    nodes, seen_map, root_idx = resolution_with_tree(clauses, goal_clause)
    if root_idx is None:
        print("\nNo empty clause could be derived — goal not entailed by KB.")
    else:
        print("\nDERIVATION TREE (bottom-up):")
        print_bottom_up_tree(nodes, root_idx)
        print("\nResolution complete — ⊥ derived.")
```

```
Enter CNF clauses (one per line). End with a blank line.
¬human(x) ∨ mortal(x)
human(Socrates)


Enter GOAL clause (single literal form): mortal(Socrates)

DERIVATION TREE (bottom-up):
└── ⊥
    ├── human(Socrates)
    └── ¬human(Socrates)
        ├── ¬human(x_0) ∨ mortal(x_0)
        └── ¬mortal(Socrates)

Resolution complete — ⊥ derived.
```

Program 10:
Implement Alpha-Beta Pruning

Algorithm:



O/P

Enter no of non - kept nodes : 4

Enter Parent node: A
Enter Children of A: B C D
Enter Parent nodes: B
Enter Children: E F
Enter Parent node: C
Enter Parent node: D
Enter Children I J

Enter number of leaf nodes
Enter leaf and value E 3
F 5
6 6
4 9
I 1
J 2

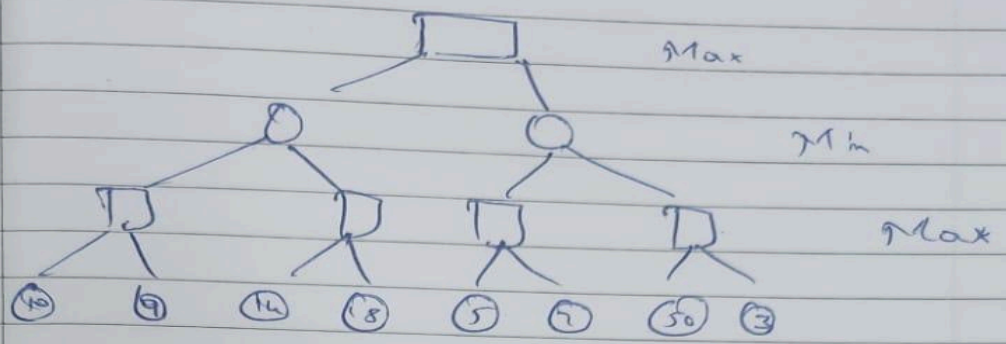
Enter root node: A
Enter total depth of the tree 3

Final optimal value 6
Pruned Node : C I J

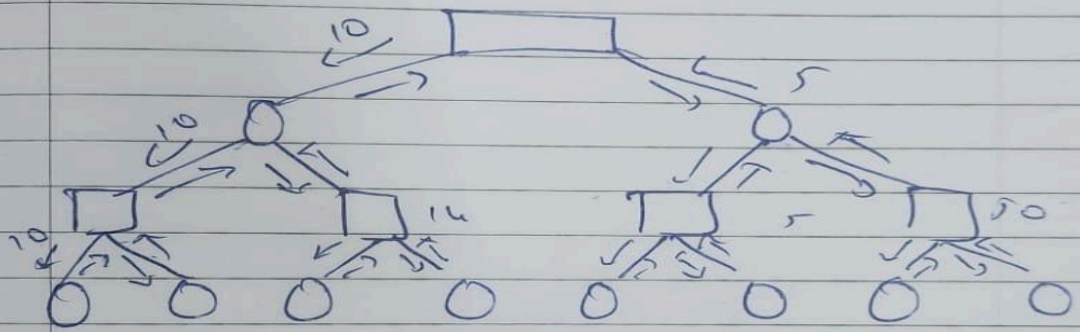
Date / /
Page

Problem

Apply the Alpha-Beta search



Sol:



Ans
6/11/25

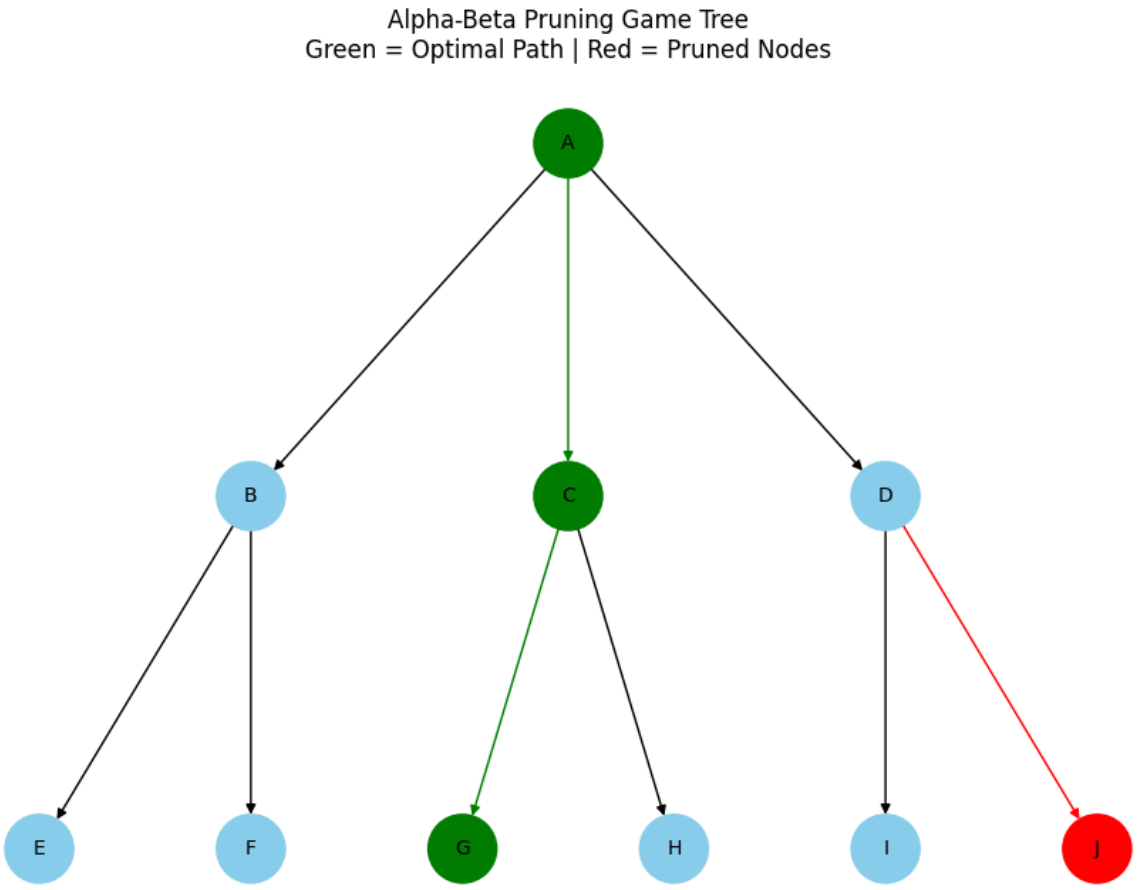
Code:

```
import networkx as nx
import matplotlib.pyplot as plt
import math

# --- Alpha-Beta Pruning ---
def alpha_beta(node, depth, alpha, beta, maximizing, tree, values, pruned_nodes,
path):
    # Leaf node
    if depth == 0 or node not in tree:
        return values.get(node, None)

    if maximizing:
        value = -math.inf
        for child in tree[node]:
            val = alpha_beta(child, depth - 1, alpha, beta, False, tree, values,
pruned_nodes, path)
            if val is None:
                continue
            value = max(value, val)
            alpha = max(alpha, value)
            if beta <= alpha:
                # Prune remaining children
                prune_index = tree[node].index(child) + 1
                for c in tree[node][prune_index:]:
                    pruned_nodes.append(c)
                break
        values[node] = value
        return value
    else:
        value = math.inf
        for child in tree[node]:
            val = alpha_beta(child, depth - 1, alpha, beta, True, tree, values,
pruned_nodes, path)
            if val is None:
                continue
            value = min(value, val)
            beta = min(beta, value)
            if beta <= alpha:
                prune_index = tree[node].index(child) + 1
                for c in tree[node][prune_index:]:
                    pruned_nodes.append(c)
                break
        values[node] = value
        return value


# --- Draw Game Tree ---
def draw_game_tree(G, path, pruned):
    pos = nx.nx_agraph.graphviz_layout(G, prog="dot")
    plt.figure(figsize=(9, 6))

    edge_colors = []
    for (u, v) in G.edges():
        if u in path and v in path:
            edge_colors.append('green')
```

```
        elif v in pruned:
            edge_colors.append('red')
        else:
            edge_colors.append('black')

    node_colors = []
    for node in G.nodes():
        if node in path:
            node_colors.append('green')
        elif node in pruned:
            node_colors.append('red')
        else:
            node_colors.append('skyblue')

    nx.draw(
        G, pos, with_labels=True,
        node_color=node_colors,
        edge_color=edge_colors,
        node_size=1200,
        font_size=10
    )

    plt.title("Alpha-Beta Pruning Game Tree\nGreen = Optimal Path | Red = Pruned
Nodes")
    plt.show()


# --- Main Program ---
def main():
    tree = {}
    G = nx.DiGraph()

    n = int(input("Enter number of non-leaf nodes: "))
    for _ in range(n):
        parent = input("\nEnter parent node: ").strip()
        children = input("Enter children of " + parent + " (space separated):
").split()
        tree[parent] = children
        for c in children:
            G.add_edge(parent, c)

    leaf_count = int(input("\nEnter number of leaf nodes: "))
    values = {}
    for _ in range(leaf_count):
        leaf, val = input("Enter leaf node and its value (e.g. E 3): ").split()
        values[leaf] = int(val)

    root = input("\nEnter root node: ").strip()
    depth = int(input("Enter total depth of tree: "))

    pruned_nodes = []
    path = []

    print("\n--------------------------------")
    result = alpha_beta(root, depth, -math.inf, math.inf, True, tree, values,
pruned_nodes, path)
    print(f"Final Optimal Value: {result}")
    print(f"Pruned Nodes: {pruned_nodes}")
```

```
    print("--------------------------------")

    draw_game_tree(G, path=[root, 'C', 'G'], pruned=pruned_nodes)


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

```
Enter number of non-leaf nodes: 4

Enter parent node: A
Enter children of A (space separated): B C D

Enter parent node: B
Enter children of B (space separated): E F

Enter parent node: C
Enter children of C (space separated): G H

Enter parent node: D
Enter children of D (space separated): I J

Enter number of leaf nodes: 6
Enter leaf node and its value (e.g. E 3): E 3
Enter leaf node and its value (e.g. E 3): F 5
Enter leaf node and its value (e.g. E 3): G 6
Enter leaf node and its value (e.g. E 3): H 9
Enter leaf node and its value (e.g. E 3): I 1
Enter leaf node and its value (e.g. E 3): J 2

Enter root node: A
Enter total depth of tree: 3

--------------------------------
Final Optimal Value: 6
Pruned Nodes: ['J']
--------------------------------
```


Alpha-Beta Pruning Game Tree
Green = Optimal Path | Red = Pruned Nodes
A
B
C
D
E
F
G
H
I
J