VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Tejas Joshi(1WA23CS016)

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 **Tejas Joshi(1WA23CS016),** 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.

Dr. K. R. Mamatha Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

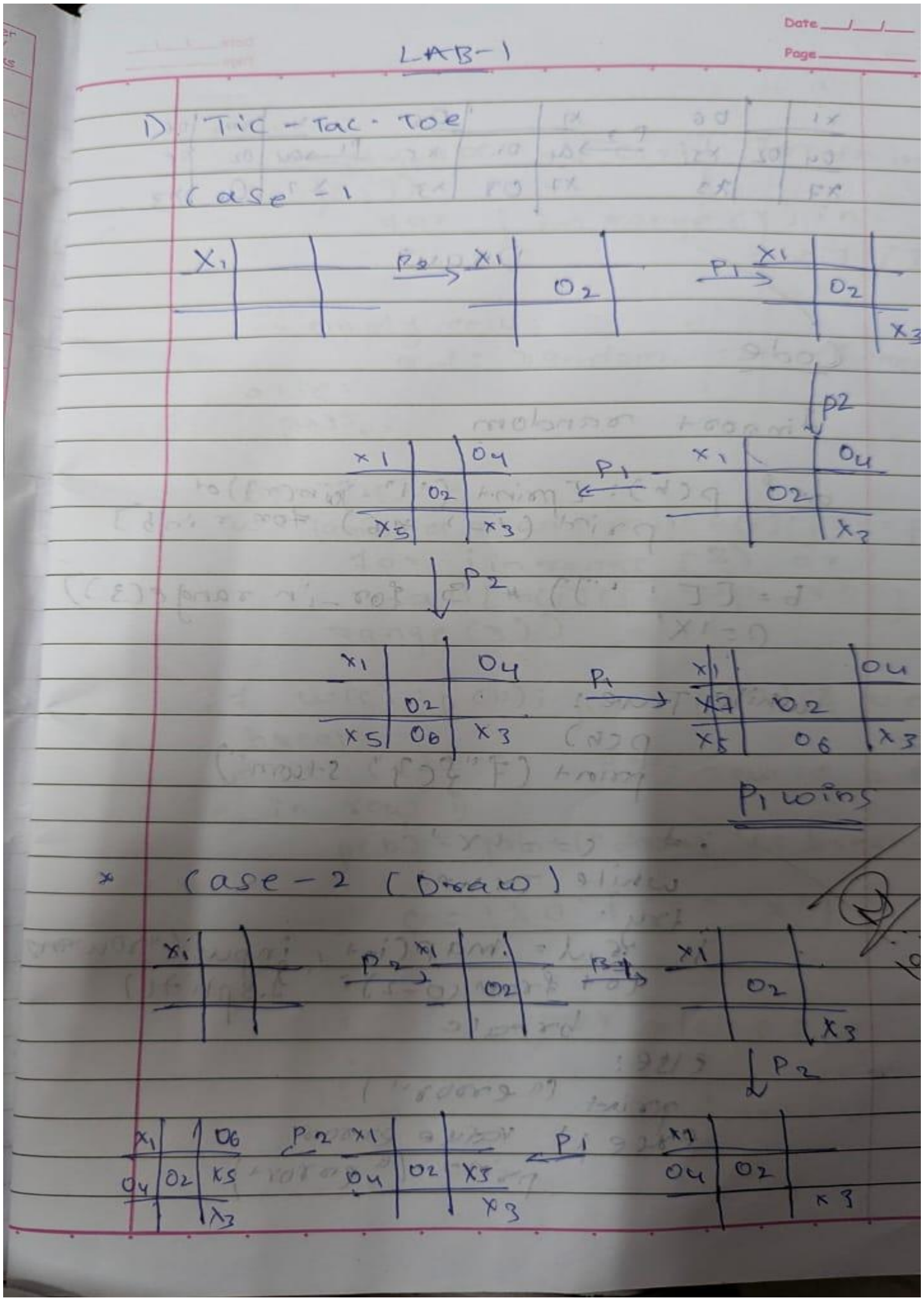
Index

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

Program 1

Implement Tic –Tac –Toe Game

Implement Vacuum Cleaner agent



X1		O6
O4	O2	X5
X7		X3

P2 →

X1		O6
O4	O2	X5
X7	O8	X3

P1 →

X	X9	O6
O4	O2	X5
X7	O8	X3

Draw

Code

```
import random

def p(b): [print ('|'.join(r)) or
        print ('-' *6) for r in b]

b = [[' '] * 3 for _ in range(3)]
c = 'X'

while True:
    p(b)
    print (f"{c}' s turn")

    if c == 'X':
    while True:
    try:
      r,c = map(int, input("row and
      col from (0-2): ").split())
      break

else:
   empty.block = [(i,j) for i in
range(3)
        for j in range(3) in
                  b[i][j] = ' ']

if empty cells:
        r,c = random. choice(empt-cells)

else:
  pass..

b[r][c] = c
w = lambda p: any (all(b[i][j] == p
      for j in range (3)) or
         all[b[i][j] == p for j in
   range (3))

if w(c): p(b); print (f"{c} win");
 break
if all (e != ' ' for row in b for
e in row )
      p(b): print ("draw "); break.

         c = 'XO'[c == 'X']
```

Output

```
 | |          enter r|c: 0 2
-----
 | |              |X|
-----
```

O' turn

	X	O

enter r|c : 1,1

	X	O
	X	

O' turn

	X	O
	X	O

enter r|c : 2,1

	X	O
	X	O
	X	

<u>X wins!</u>

O' turn

	X	O

enter r/c : 1,1

	X	O
	X	

O' turn

	X	O
	X	O

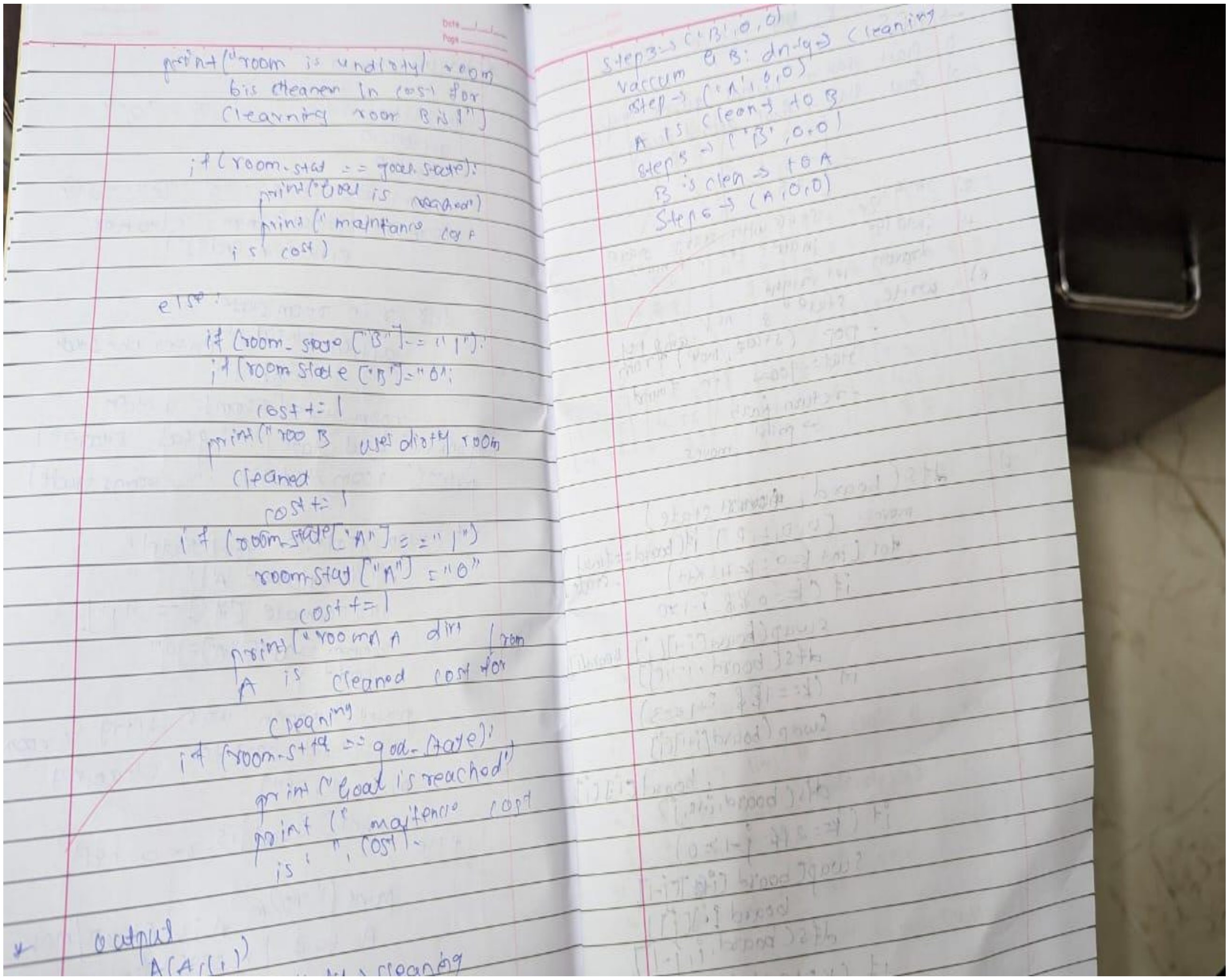
enter r/c : 2,1

	X	O
	X	O
	X	

<u>X wins!</u>

* Vaccum cleaner

```
goal state = { 'A' : '0', 'D' : "0" }
action = 0
cost = 0
room state = { "A" : "0", "B" : "0" }
location : input (" Enter cleaner
        either (A and B) ")

for r in room state
    action = input (" enter the state
        0 or 1 ")
    room-state [room] = action
print (" Goal state is: ", goal- state")
print (" room state is ", rooms state)

if (room- state != goal. state):
    if (location == 'A'):
        if (roomate ["A"] == "1"):
            room. state ["A"] = "0"
            cost += 1
            print (" room A was dirty in room
            A is cleand cost for cleaning
            room A is 1")
            print (" Goal is reached")
    else:
        print (" room A is clean/mov
        A to B | n cost for movin
        A to B is 1")
        cost += 1
```



print("room is undirty room b is cleaned in cost for cleaning room B is")

if (room_stat == goal_state):
print("Goal is reached")
print("maintance cost is", cost)

else:
if (room_state["B"] == "1"):
if (room_state["B"] = "0":
cost += 1
print("room B was dirty room cleaned
cost += 1
if (room_state["A"] == "1")
room_state["A"] = "0"
cost += 1
print("room A dirty room A is cleaned cost for cleaning
if (room_state == goal_state):
print("Goal is reached")
print("maitenance cost is", cost)

* output
A(A,1,1)
cleaning

step3 -> ('B', 0, 0)
vaccum @ B: dirty -> cleaning
step -> ('A', 0, 0)
A is clean -> to B
step5 -> ('B', 0, 0)
B is clean -> to A
step6 -> (A, 0, 0)

Code:

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

def print_cell_mapping():
    mapping = [
        "1 | 2 | 3",
        "- - - - - - -",
        "4 | 5 | 6",
        "- - - - - - -",
        "7 | 8 | 9"
    ]
    print("\nCell Mapping:")
    for line in mapping:
```

```
        print(line)
    print()

def check_winner(board):
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != " ":
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != " ":
            return board[0][i]

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

def tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = "X"
    print_cell_mapping()

    while True:
        print_board(board)
        move = int(input(f"Player {current_player}, enter the cell number (1-9): ")) - 1
        row, col = divmod(move, 3)

        if 0 <= move < 9 and board[row][col] == " ":
            board[row][col] = current_player
        else:
            print("Invalid move! Try again.")
            continue

        winner = check_winner(board)
        if winner:
            print_board(board)
            print(f"Player {winner} wins!")
            break

        if is_board_full(board):
            print_board(board)
            print("It's a draw!")
            break
        current_player = "O" if current_player == "X" else "X"
```

tic_tac_toe()
Output :

```
Cell Mapping:
1 | 2 | 3
- - - - - - -
4 | 5 | 6
- - - - - - -
7 | 8 | 9

  |   |
---------
  |   |
---------
  |   |
---------
Player X, enter the cell number (1-9): 1
X |   |
---------
  |   |
---------
  |   |
---------
Player O, enter the cell number (1-9): 2
X | O |
---------
  |   |
---------
  |   |
---------
Player X, enter the cell number (1-9): 5
X | O |
---------
  | X |
---------
  |   |
---------
Player O, enter the cell number (1-9): 3
X | O | O
---------
  | X |
---------
  |   |
---------
Player X, enter the cell number (1-9): 9
X | O | O
---------
  | X |
```

```
---------
 |  | X
---------
Player X wins!
```

Code :

```
import random

rooms = [1, 1]
print(f"Initial room states: {rooms}")

cleaner_location = random.randint(0, 1)
print(f"Cleaner starts in room: {cleaner_location + 1}")

cleaned_count = 0
cost_count = 0

while cleaned_count < 2:
  print(f"\nCleaner is in room: {cleaner_location + 1}")

  if rooms[cleaner_location]:
    print("Room is dirty.")
    clean_confirm = input("Clean? (y/n): ")

    if clean_confirm == 'y':
      rooms[cleaner_location] = 0
      cleaned_count += 1
      cost_count += 1
      print("Room is now clean.")
    else:
      print("Room not cleaned.")

  else:
    print("Room is already clean.")

  print(f"Current room states: {rooms}")
  print(f"Current cost: {cost_count}")

  if cleaned_count < 2:
    while 1 == 1:
     move_to = int(input("Enter the room number to move to (1 or 2): ")) - 1
     if move_to in [0, 1]:
      cleaner_location = move_to
      cost_count += 1
      break
     else:
      print("Invalid room number. Please enter 1 or 2.")
```

```
print("\nAll rooms are clean. Program ends.")
print(f"Total cost: {cost_count}")
```

Output :

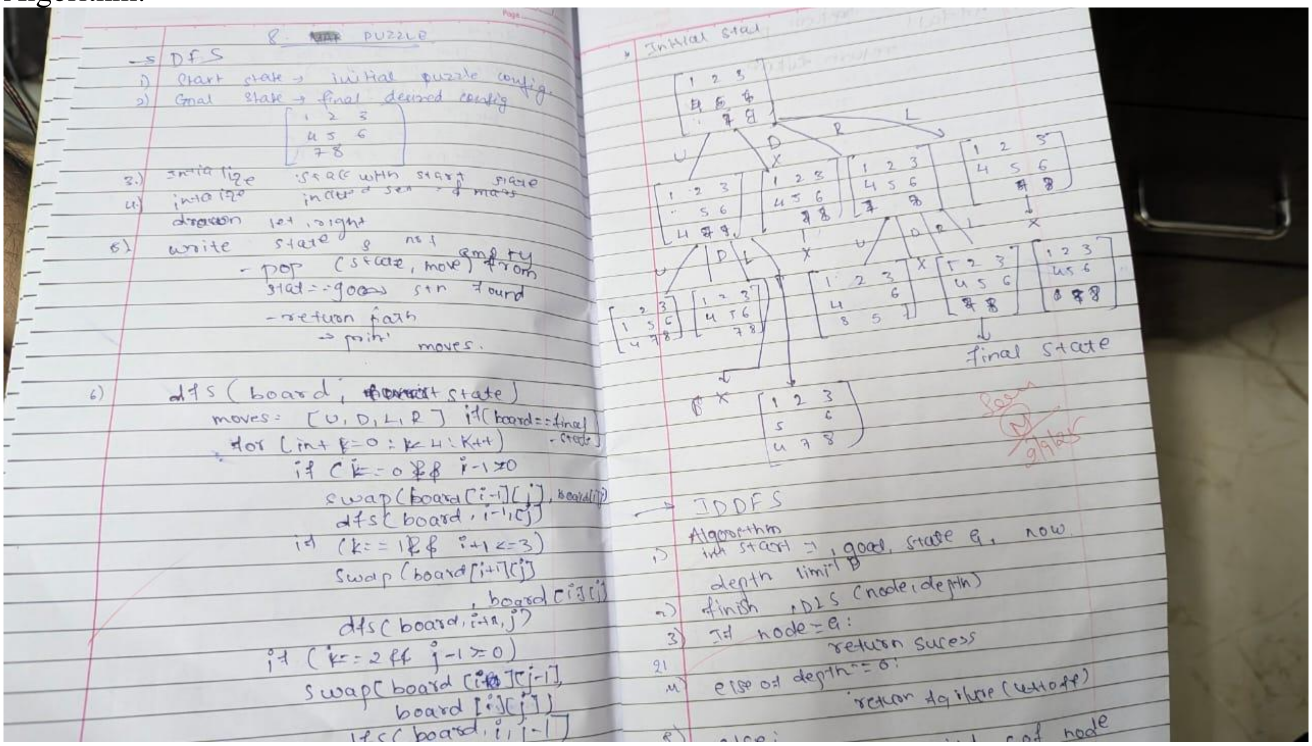
```
Initial room states: [1, 1]
Cleaner starts in room: 2

Cleaner is in room: 2
Room is dirty.
Clean? (y/n): y
Room is now clean.
Current room states: [1, 0]
Current cost: 1
Enter the room number to move to (1 or 2): 1

Cleaner is in room: 1
Room is dirty.
Clean? (y/n): y
Room is now clean.
Current room states: [0, 0]
Current cost: 3

All rooms are clean. Program ends.
Total cost: 3
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)

Implement Iterative deepening search algorithm

Algorithm:



Code:

Eight Puzzle

```
def swap(arr, x1, y1, x2, y2):
    arr[x1][y1], arr[x2][y2] = arr[x2][y2], arr[x1][y1]

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

zero_pos = [1, 0]

def print_puzzle(arr):
    for row in arr:
        print(row)
    print("-" * 10)

moves = {
    'up': (-1, 0),
    'down': (1, 0),
    'left': (0, -1),
    'right': (0, 1)
}
```

```
while final != arr:
    print_puzzle(arr)
    move = input("Enter move (up, down, left, right): ").lower()

    dx, dy = moves[move]
    new_x, new_y = zero_pos[0] + dx, zero_pos[1] + dy

    if 0 <= new_x < 3 and 0 <= new_y < 3:
      swap(arr, zero_pos[0], zero_pos[1], new_x, new_y)
      zero_pos[0], zero_pos[1] = new_x, new_y
    else:
      print("Invalid move.")
print("Complete")
```

Output :

```
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
----------
Enter move (up, down, left, right): right
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
----------
Enter move (up, down, left, right): down
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
----------
Enter move (up, down, left, right): right
Complete
```

Eight puzzle with heuristic

```
import copy
def swap(arr, x1, y1, x2, y2):
    arr[x1][y1], arr[x2][y2] = arr[x2][y2], arr[x1][y1]

def get_zero_pos(arr):
    for r in range(len(arr)):
        for c in range(len(arr[r])):
            if arr[r][c] == 0:
                return [r, c]
    return None
def print_puzzle(arr):
    for row in arr:
        print(row)
```

```
    print("-" * 10)

def count_mismatched_tiles(current_arr, final_arr):
    count = 0
    for r in range(len(current_arr)):
        for c in range(len(current_arr[r])):
            if current_arr[r][c] != final_arr[r][c] and current_arr[r][c] != 0:
                count += 1
    return count

def solve_puzzle_heuristic(initial_arr, final_arr):
    open_list = [(initial_arr, [initial_arr], get_zero_pos(initial_arr))]
    closed_set = set()
    moves = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0, 1)}
    while open_list:
        open_list.sort(key=lambda item: count_mismatched_tiles(item[0], final_arr))
        current_arr, path, zero_pos = open_list.pop(0)
        if current_arr == final_arr:
            return path
        current_tuple = tuple(tuple(row) for row in current_arr)
        if current_tuple in closed_set:
            continue
        closed_set.add(current_tuple)
        for move, (dx, dy) in moves.items():
            new_x, new_y = zero_pos[0] + dx, zero_pos[1] + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_arr = copy.deepcopy(current_arr)
                swap(new_arr, zero_pos[0], zero_pos[1], new_x, new_y)
                new_path = path + [new_arr]
                new_zero_pos = [new_x, new_y]
                open_list.append((new_arr, new_path, new_zero_pos))
    return None

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

print("Initial State:")
print_puzzle(initial_state)

solution_path = solve_puzzle_heuristic(initial_state, final_state)

if solution_path:
    print("Solution Found:")
    for step in solution_path:
        print_puzzle(step)
    print("Complete")
else:
```

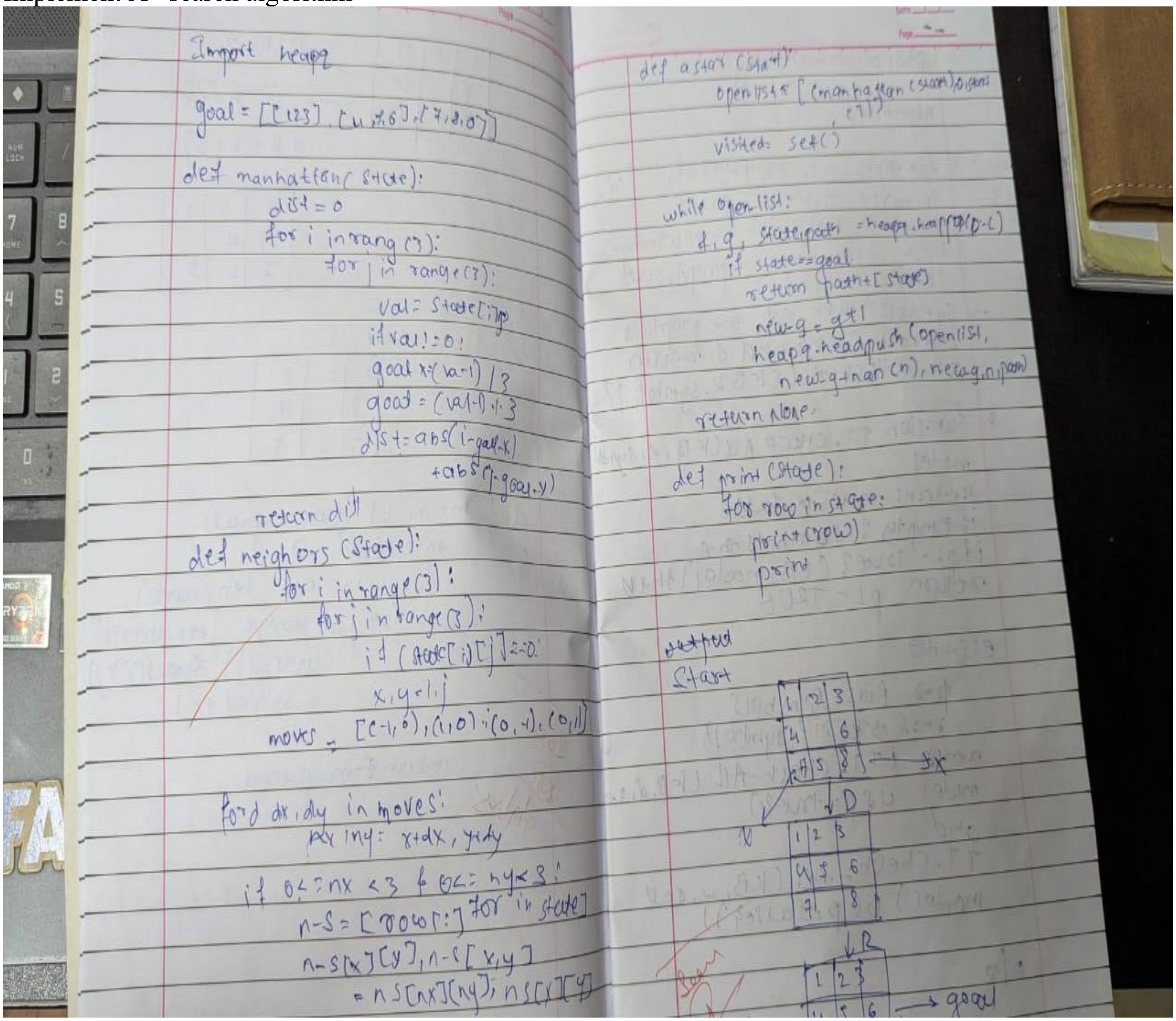
```
    print("No solution exists for this puzzle.")
```

Output :

```
Initial State:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
----------
Solution Found:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
----------
[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]
----------
Complete
```

Program 3

Implement A* search algorithm



```
Import heapq

goal = [[1 2 3] . [4, 5, 6] , [7, 8, 0]]

def manhattan( state):
        dist = 0
        for i in rang (3):
            for j in range(3):
                val = state[i][j]
                if val != 0 :
                goal x = (va-1) / 3
                goal = (val-1) % 3
                dist = abs( i-goal-x)
                    +abs(j-goal.y)
        return dist
def neighbors (state):
        for i in range(3) :
            for j in range(3):
                if ( state[i][j] == 0:
                x, y = i, j
        moves = [(-1, 0), (1, 0) ; (0, -1), (0, 1)]

    for dx, dy in moves:
        nx, ny = x+dx, y+dy

    if 0 <= nx < 3 & 0 <= ny < 3 :
        n-s = [row[:] for in state]
        n-s[x][y], n-s[x, y]
            = ns[nx][ny], ns[x][y]

def astar (start):
        openlist = [ (manhattan (start), 0, start
                        [])]
        visited = set()

while open-list:
    f, g, state, path = heapq.heappop(o-l)
        if state == goal
            return path+[state]
        new-g = g+1
        heapq.heappush (openlist,
            new-g+man (n), new-g, n, path)
    return None.

def print (state):
    for row in state:
        print(row)
        print

output
Start
```

1	2	3
4		6
7	5	8

→ X

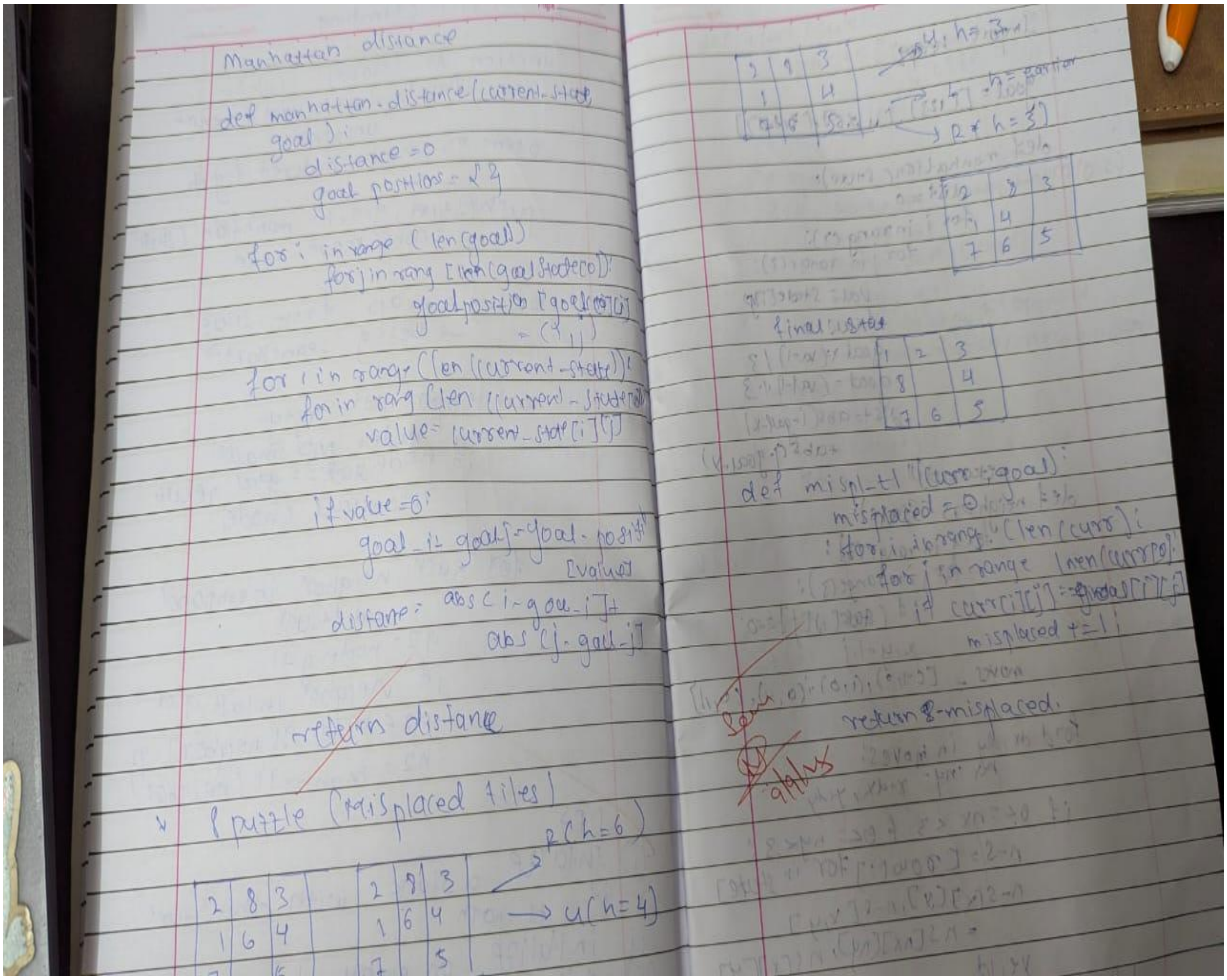
↓ D

1	2	3
4	5	6
7		8

↓ R

1	2	3
4	5	6

→ goal



Code :

```
def find_empty(board):
    """Find position of empty tile (0)"""
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:
                return i, j
    return 0, 0

def is_goal(board, goal_board):
    """Check if current state is goal state"""
    return board == goal_board

def copy_board(board):
```

```
    """Create a copy of the board"""
    new_board = []
    for i in range(3):
        row = []
        for j in range(3):
            row.append(board[i][j])
        new_board.append(row)
    return new_board

def get_neighbors(board):
    """Get all possible next states"""
    neighbors = []
    empty_row, empty_col = find_empty(board)

    # Try all 4 directions: up, down, left, right
    directions = [(-1, 0, "UP"), (1, 0, "DOWN"), (0, -1, "LEFT"), (0, 1, "RIGHT")]

    for dr, dc, move_name in directions:
        new_row = empty_row + dr
        new_col = empty_col + dc

        # Check if move is within bounds
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            # Create new board by copying current board
            new_board = copy_board(board)

            # Swap empty tile with adjacent tile
            new_board[empty_row][empty_col] = new_board[new_row][new_col]
            new_board[new_row][new_col] = 0

            neighbors.append((new_board, move_name))

    return neighbors

def misplaced_tiles_heuristic(board, goal_board):
    """Count number of tiles in wrong position"""
    count = 0

    for i in range(3):
        for j in range(3):
            if board[i][j] != 0 and board[i][j] != goal_board[i][j]:
                count += 1

    return count

def manhattan_distance_heuristic(board, goal_board):
    """Calculate Manhattan distance for each tile"""
```

```
    # Goal positions for each number
    goal_positions = {}
    for i in range(3):
        for j in range(3):
            goal_positions[goal_board[i][j]] = (i, j)

    total_distance = 0

    for i in range(3):
        for j in range(3):
            if board[i][j] != 0:  # Don't calculate for empty tile
                goal_row, goal_col = goal_positions[board[i][j]]
                distance = abs(i - goal_row) + abs(j - goal_col)
                total_distance += distance

    return total_distance

def print_board(board):
    """Print the board nicely"""
    print(" ┌───────────┐ ")
    for row in board:
        print(" │ ", end="")
        for cell in row:
            if cell == 0:
                print("   ", end="")
            else:
                print(f" {cell} ", end="")
        print(" │ ")
    print(" └───────────┘ ")

def board_to_string(board):
    """Convert board to string for comparison"""
    result = ""
    for row in board:
        for cell in row:
            result += str(cell)
    return result

def find_min_f_cost(open_list):
    """Find node with minimum f cost in open list"""
    min_f = float('inf')
    min_index = -1

    for i in range(len(open_list)):
        if open_list[i]['f_cost'] < min_f:
            min_f = open_list[i]['f_cost']
            min_index = i
```

```
    return min_index

def reconstruct_path(node):
    """Reconstruct path from goal to start"""
    path = []
    current = node

    while current['parent'] is not None:
        path.append({'board': current['board'], 'move': current['move']})
        current = current['parent']

    path.reverse()
    return path

def a_star_solver(start_board, goal_board, heuristic_type):
    """A* algorithm implementation"""

    # Choose heuristic function
    if heuristic_type == 1:
        heuristic_func = misplaced_tiles_heuristic
        heuristic_name = "Misplaced Tiles"
    else:
        heuristic_func = manhattan_distance_heuristic
        heuristic_name = "Manhattan Distance"

    print(f"\nUsing {heuristic_name} heuristic")
    print("="*40)

    # Check if already solved
    if is_goal(start_board, goal_board):
        print("Puzzle already solved!")
        return

    # Initialize open and closed lists
    open_list = []
    closed_list = []

    # Create start node
    start_node = {
        'board': start_board,
        'g_cost': 0,
        'h_cost': heuristic_func(start_board, goal_board),
        'f_cost': 0,
        'parent': None,
        'move': "START"
    }
```

```
start_node['f_cost'] = start_node['g_cost'] + start_node['h_cost']

open_list.append(start_node)
nodes_expanded = 0

while len(open_list) > 0:
    # Find node with minimum f cost
    current_index = find_min_f_cost(open_list)
    current_node = open_list.pop(current_index)
    closed_list.append(current_node)
    nodes_expanded += 1

    # Check if goal reached
    if is_goal(current_node['board'], goal_board):
        print(f"Solution found! Nodes expanded: {nodes_expanded}")
        print(f"Solution length: {current_node['g_cost']} moves\n")

        # Reconstruct and print path
        path = reconstruct_path(current_node)

        print("Solution path:")
        print("Initial state:")
        print_board(start_board)

        for step, state in enumerate(path):
            print(f"\nStep {step + 1}: Move {state['move']}")
            print_board(state['board'])

        return

    # Get neighbors
    neighbors = get_neighbors(current_node['board'])

    for neighbor_board, move in neighbors:
        # Check if neighbor is in closed list
        neighbor_string = board_to_string(neighbor_board)
        in_closed = False

        for closed_node in closed_list:
            if board_to_string(closed_node['board']) == neighbor_string:
                in_closed = True
                break

        if in_closed:
            continue

        # Calculate costs
```

```
            g_cost = current_node['g_cost'] + 1
            h_cost = heuristic_func(neighbor_board, goal_board)
            f_cost = g_cost + h_cost

            # Check if neighbor is in open list with better cost
            in_open = False
            for open_node in open_list:
                if board_to_string(open_node['board']) == neighbor_string:
                    if g_cost < open_node['g_cost']:
                        open_node['g_cost'] = g_cost
                        open_node['f_cost'] = f_cost
                        open_node['parent'] = current_node
                        open_node['move'] = move
                    in_open = True
                    break

            # Add to open list if not already there
            if not in_open:
                neighbor_node = {
                    'board': neighbor_board,
                    'g_cost': g_cost,
                    'h_cost': h_cost,
                    'f_cost': f_cost,
                    'parent': current_node,
                    'move': move
                }
                open_list.append(neighbor_node)

    print("No solution found!")

def get_puzzle_input():
    """Get puzzle input from user"""
    print("Enter your 3x3 puzzle:")
    print("Use 0 for empty tile")
    print("Enter each row (3 numbers separated by spaces):")

    board = []
    for i in range(3):
        while True:
            try:
                row_input = input(f"Row {i+1}: ").strip()
                row = list(map(int, row_input.split()))

                if len(row) != 3:
                    print("Please enter exactly 3 numbers")
                    continue
```

```
            # Check if numbers are valid (0-8)
            valid = True
            for num in row:
                if num < 0 or num > 8:
                    print("Numbers must be between 0 and 8")
                    valid = False
                    break

            if valid:
                board.append(row)
                break

        except ValueError:
            print("Please enter valid integers")

    return board

def main():
    """Main function"""
    print("="*50)
    print("     A* SLIDING PUZZLE SOLVER")
    print("="*50)

    # Get puzzle input
    puzzle = get_puzzle_input()

    print("\nYour puzzle:")
    print_board(puzzle)

    # Get goal state input
    print("\nEnter your 3x3 goal state:")
    goal_puzzle = get_puzzle_input()

    print("\nYour goal state:")
    print_board(goal_puzzle)

    # Choose heuristic
    print("\nChoose heuristic:")
    print("1. Misplaced Tiles")
    print("2. Manhattan Distance")

    while True:
        try:
            choice = int(input("Enter choice (1 or 2): "))
            if choice in [1, 2]:
                break
            else:
```

```
            print("Please enter 1 or 2")
        except ValueError:
            print("Please enter a valid number")

    # Solve puzzle
    a_star_solver(puzzle, goal_puzzle, choice)

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

Output:
Enter your 3x3 puzzle:

Start state:
2 8 3
1 6 4
7 5
Total states visited: 7
Solution found!
Moves: U U L D R
Number of moves: 5

Move 1: U
2 8 3
1 4
7 6 5
g(n) = 1, h(n) = 3, f(n) = g(n) + h(n) = 4
Move 2: U
2 3
1 8 4
7 6 5
g(n) = 2, h(n) = 3, f(n) = g(n) + h(n) = 5
Move 3: L
 2 3
1 8 4
7 6 5
g(n) = 3, h(n) = 2, f(n) = g(n) + h(n) = 5
Move 4: D
1 2 3
 8 4
7 6 5
g(n) = 4, h(n) = 1, f(n) = g(n) + h(n) = 5
Move 5: R
1 2 3
27
8 4
7 6 5

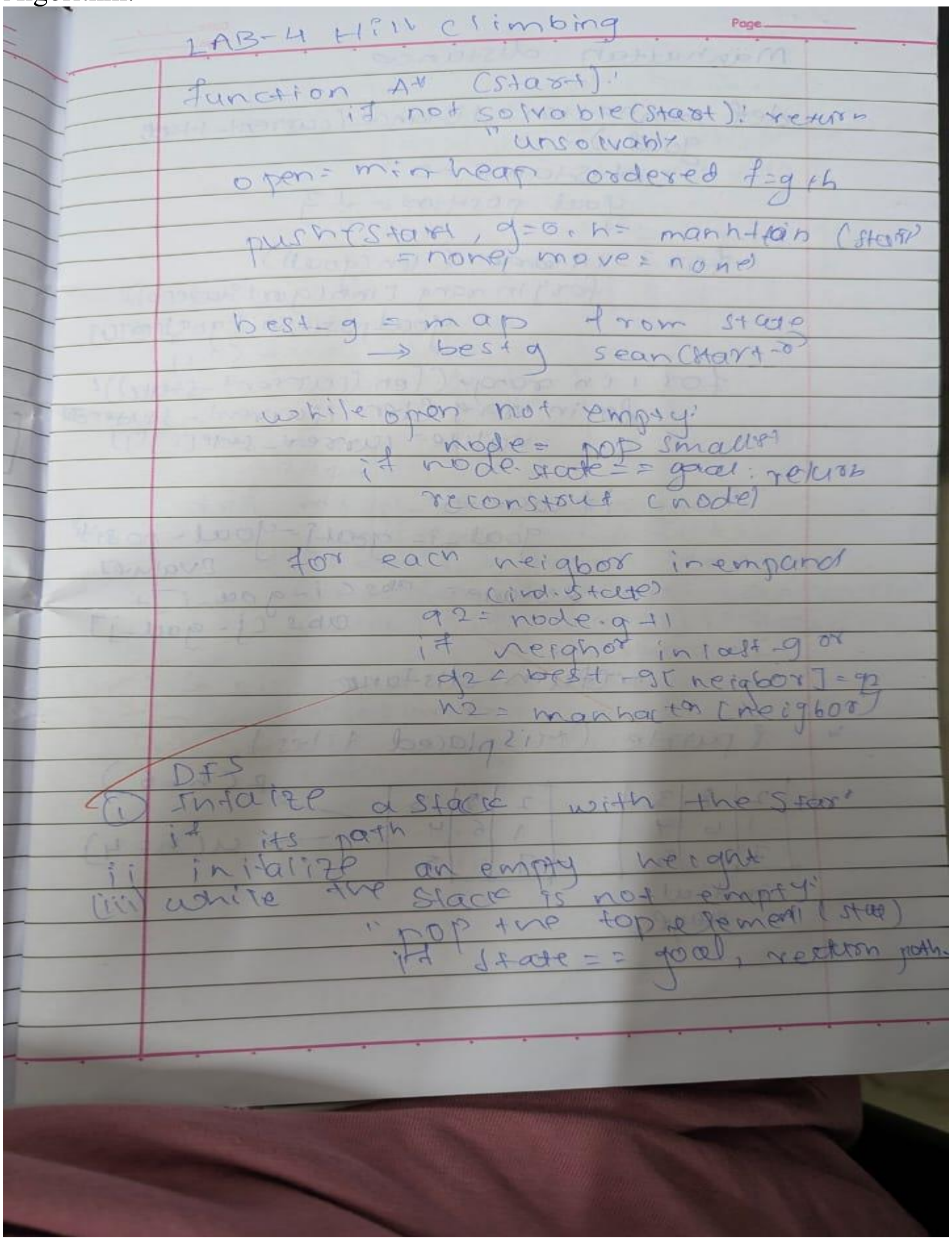
g(n) = 5, h(n) = 0, f(n) = g(n) + h(n) = 5

Output 2:
Start state:
1 2 3
6 7 8
4 5
Total states visited: 21
Solution found!
Moves: L U L D R R U L D R
Number of moves: 10

Move 1: L
1 2 3
6 7 8
4 5
g(n) = 1, h(n) = 9, f(n) = g(n) + h(n) = 10
Move 2: U
1 2 3
6 8
4 7 5
30
g(n) = 2, h(n) = 8, f(n) = g(n) + h(n) = 10
Move 3: L
1 2 3
 6 8
4 7 5
g(n) = 3, h(n) = 7, f(n) = g(n) + h(n) = 10
Move 4: D
1 2 3
4 6 8
 7 5
g(n) = 4, h(n) = 6, f(n) = g(n) + h(n) = 10
Move 5: R
1 2 3
4 6 8
7 5
g(n) = 5, h(n) = 5, f(n) = g(n) + h(n) = 10
Move 6: R
1 2 3
4 6 8
7 5
g(n) = 6, h(n) = 4, f(n) = g(n) + h(n) = 10
Move 7: U
1 2 3
4 6
7 5 8

g(n) = 7, h(n) = 3, f(n) = g(n) + h(n) = 10
Move 8: L
1 2 3
4 6
7 5 8
g(n) = 8, h(n) = 2, f(n) = g(n) + h(n) = 10
31
Move 9: D
1 2 3
4 5 6
7 8
g(n) = 9, h(n) = 1, f(n) = g(n) + h(n) = 10
Move 10: R
1 2 3
4 5 6
7 8
g(n) = 10, h(n) = 0, f(n) = g(n) + h(n) = 10

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


LAB-4 Hill climbing

function A* (start):
if not solvable(start): return "unsolvable"
open = min heap ordered f=g+h
push(start, g=0, h= manhattan (start) = none, move = none)
best_g = map from state → best g seen(start→0)
while open not empty:
node = pop smallest
if node.state == goal: return reconstruct (node)
for each neighbor in expand (ind.state)
g2 = node.g+1
if neighbor in last_g or g2 < best_g[neighbor] = g2
h2 = manhattan [neighbor]

DFS
(i) Initialize a stack with the start if its path
ii initialize an empty height
(iii) while the stack is not empty:
pop the top element (state)
if state == goal, return path.

Code :

```
import random

def calculate_cost(state):
    """Calculates the number of attacking queen pairs."""
    n = len(state)
    cost = 0
```

```
    for i in range(n):
        for j in range(i + 1, n):
            # Check for attacks in the same row or on diagonals
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                cost += 1
    return cost

def generate_random_state(n=4):
    """Generates a random initial state."""
    return [random.randint(0, n - 1) for _ in range(n)]

def generate_neighbours(state):
    """Generates all neighbours by swapping column positions."""
    n = len(state)
    neighbours = []
    for i in range(n):
        for j in range(i + 1, n):
            neighbour = list(state)
            neighbour[i], neighbour[j] = neighbour[j], neighbour[i]
            neighbours.append(neighbour)
    return neighbours

def hill_climbing_search():
    """Performs hill-climbing search for the 4-Queens problem."""
    current_state = generate_random_state()
    current_cost = calculate_cost(current_state)

    print("Initial State:", current_state, "Cost:", current_cost)

    while current_cost > 0:
        neighbours = generate_neighbours(current_state)
        best_neighbour = None
        best_neighbour_cost = current_cost

        for neighbour in neighbours:
            neighbour_cost = calculate_cost(neighbour)
            if neighbour_cost < best_neighbour_cost:
                best_neighbour_cost = neighbour_cost
                best_neighbour = neighbour

        if best_neighbour_cost < current_cost:
            current_state = best_neighbour
            current_cost = best_neighbour_cost
            print("Chosen Neighbour:", current_state, "Cost:", current_cost)
        else:
            print("No neighbour has a lower cost. Stopping.")
            break
```

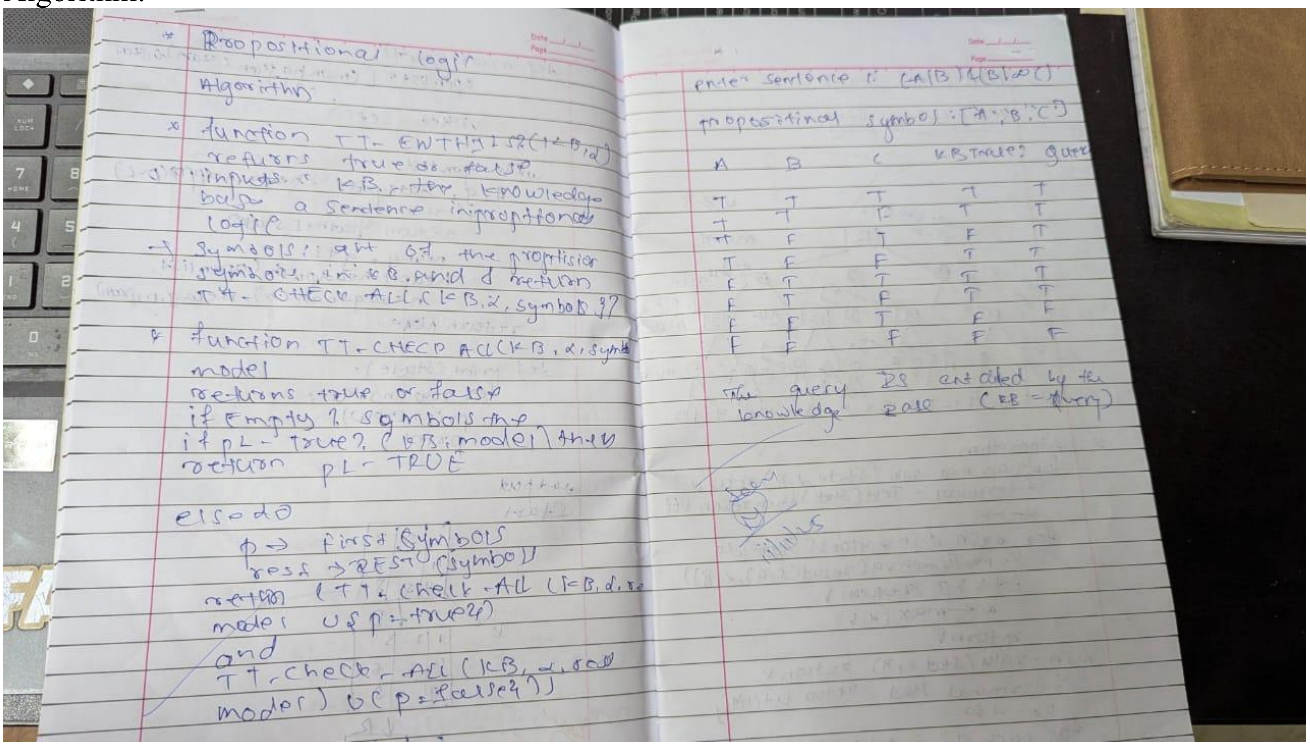
```
    if current_cost == 0:
        print("Goal reached! Solution:", current_state)

# Run the hill-climbing search
hill_climbing_search()
```

Output :

```
Initial State: [0, 1, 2, 3] Cost: 6
Chosen Neighbour: [1, 0, 2, 3] Cost: 2
Chosen Neighbour: [2, 0, 1, 3] Cost: 1
Chosen Neighbour: [2, 0, 3, 1] Cost: 0
Goal reached! Solution: [2, 0, 3, 1]
```

Program 5
Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

Algorithm:


* Propositional logic
Algorithm
* function TT-ENTAILS?(KB, α)
returns true or false.
inputs: KB, the knowledge base a sentence in propositional logic
symbols: a list of the proposition symbols in KB and α
return TT-CHECK-ALL(KB, α, symbols, {})
* function TT-CHECK-ALL(KB, α, symbols, model)
returns true or false
if EMPTY?(symbols) then
if PL-TRUE?(KB, model) then
return PL-TRUE
else do
P ← FIRST(symbols)
rest ← REST(symbols)
return (TT-CHECK-ALL(KB, α, rest, model ∪ {P = true})
and
TT-CHECK-ALL(KB, α, rest, model ∪ {P = false}))

enter sentence: (A|B)&(B|~C)
propositional symbols: ['A', 'B', 'C']

A	B	C	KB True?	Query
T	T	T	T	T
T	T	F	T	T
T	F	T	F	T
T	F	F	T	T
F	T	T	T	T
F	T	F	T	T
F	F	T	F	F
F	F	F	F	F

The query is entailed by the knowledge base (KB ⊨ query)

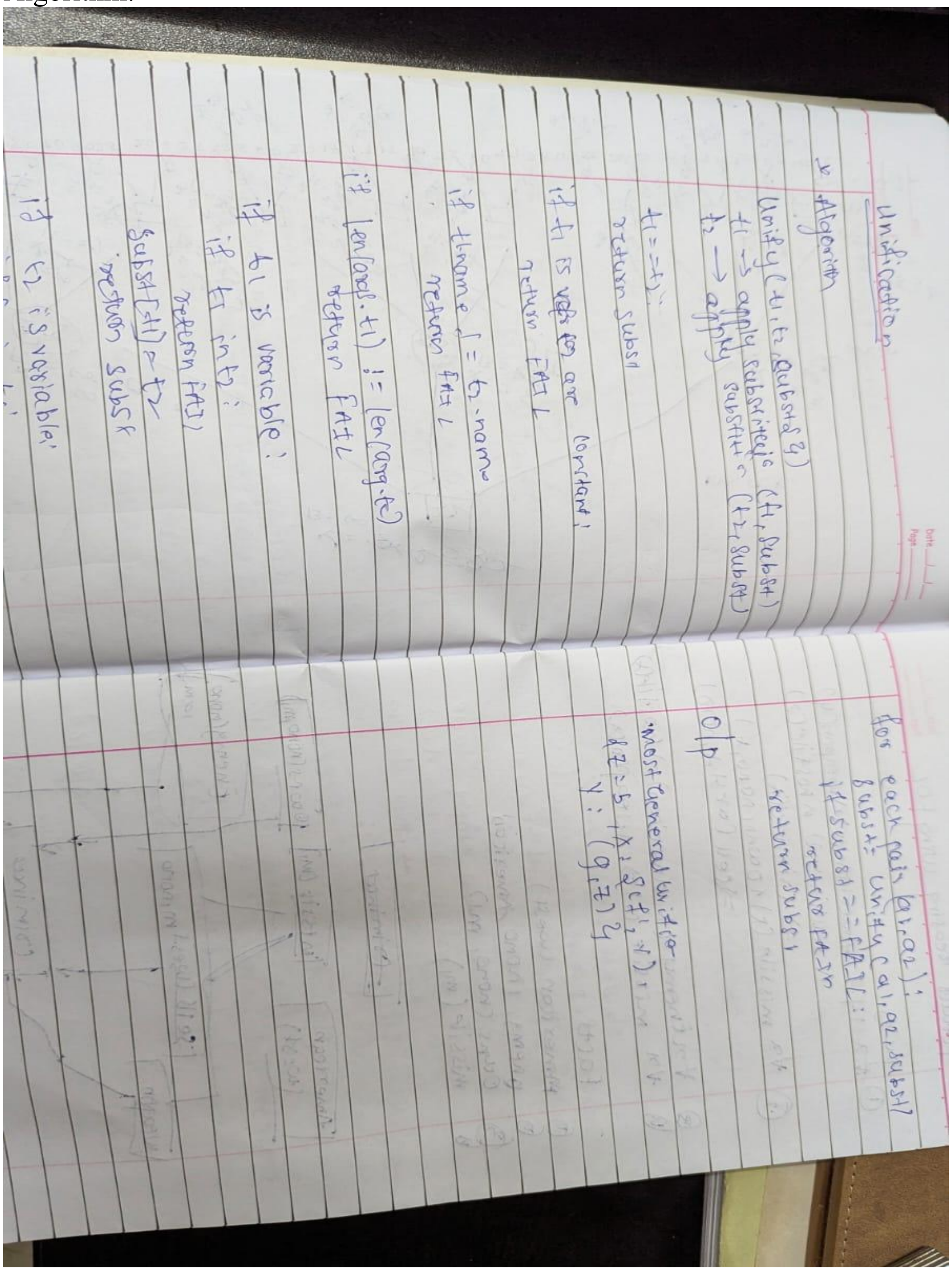
Code:
```
from itertools import product
import pandas as pd

def KB(combo):
  a,b,c = combo
  return ((a or c) and (b or not c))

def alpha(combo):
  a, b, c = combo
  return a or b



n = 3
binary = [list(i) for i in product([False, True], repeat=n)]
base = [KB(x) for x in binary]
alpha_values = [alpha(x) for x in binary]

result = [(base[i] and alpha_values[i]) for i in range(2**n)]
```

```
df = pd.DataFrame(binary, columns=[f"V{i+1}" for i in range(n)])

df['KB'] = base
df['Alpha'] = alpha_values
df['Final'] = result
df
for index, row in df[df['Final'] == True].iterrows():
  print(row['V1'], row['V2'], row['V3'])
```

Output :
False True True
True False False
True True False
True True True

Program 6
Implement unification in first order logic

Algorithm:



Unification

* Algorithm

Unify(t1, t2, subst {})
t1 → apply substitution (t1, subst)
t2 → apply substitution (t2, subst)

t1 == t2:
return subst

if t1 is var or are constant:
return FAIL

if t1.name != t2.name
return FAIL

if len(args.t1) != len(arg.t2)
return FAIL

if t1 is variable:
if t1 in t2:
return FAIL
subst[t1] = t2
return subst

if t2 is variable:

for each pair (a1, a2):
subst = unify(a1, a2, subst)
if subst == FAIL:
return FAIL
return subst

O/p

Most General unifier
{Z = b, X: f('f', Y), Y: (g, Z)}

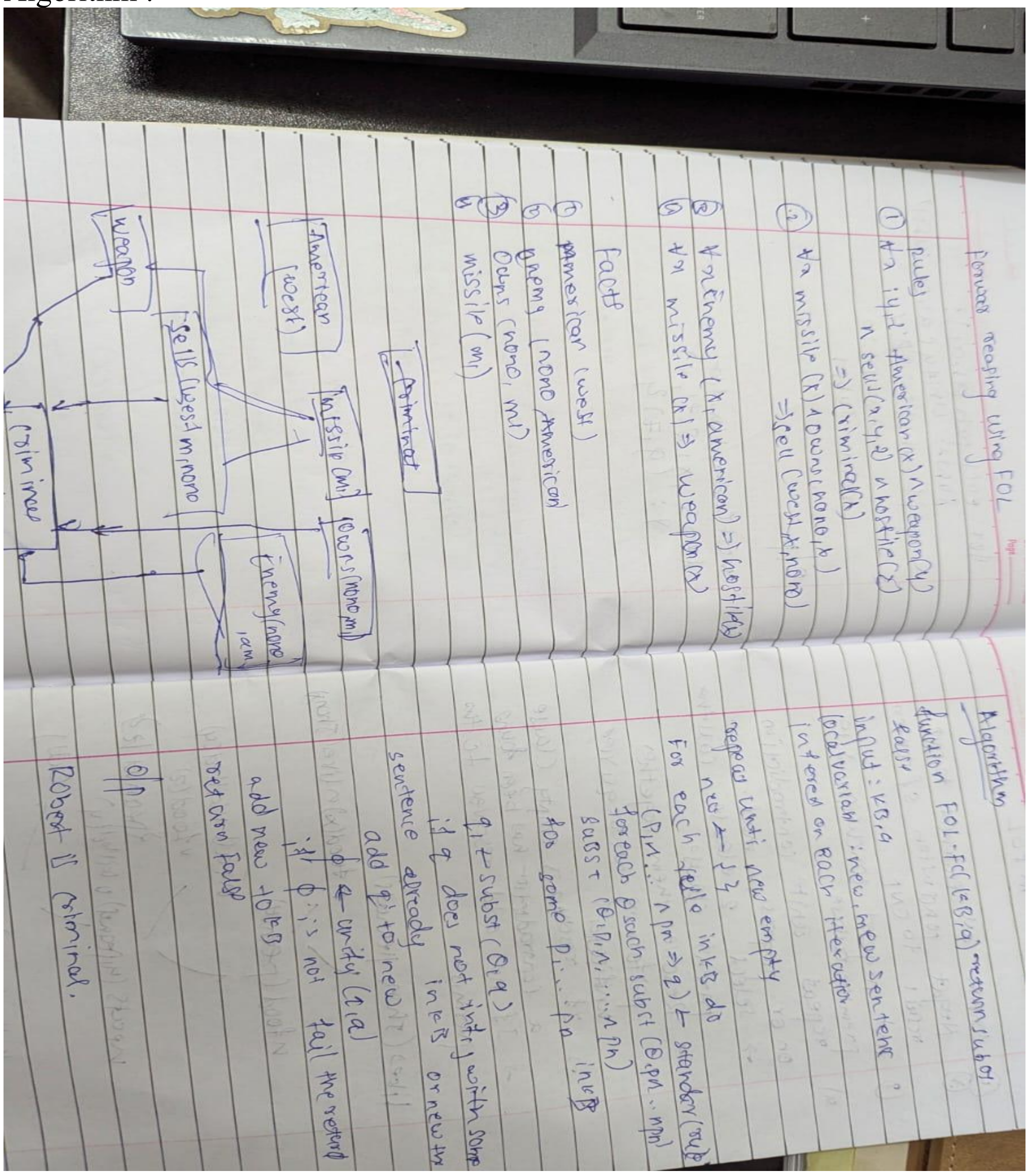
Code :
```
class UnificationError(Exception):
    pass

def occurs_check(var, term):
    """Check if a variable occurs in a term (to prevent infinite recursion)."""
    if var == term:
        return True
    if isinstance(term, tuple):  # Term is a compound (function term)
        return any(occurs_check(var, subterm) for subterm in term)
    return False

def unify(term1, term2, substitutions=None):
    """Try to unify two terms, return the MGU (Most General Unifier)."""
    if substitutions is None:
        substitutions = {}

    # If both terms are equal, no further substitution is needed
    if term1 == term2:
        return substitutions

    # If term1 is a variable, we substitute it with term2
    elif isinstance(term1, str) and term1.isupper():
        # If term1 is already substituted, recurse
        if term1 in substitutions:
            return unify(substitutions[term1], term2, substitutions)
        elif occurs_check(term1, term2):
            raise UnificationError(f"Occurs check fails: {term1} in {term2}")
        else:
            substitutions[term1] = term2
            return substitutions

    # If term2 is a variable, we substitute it with term1
    elif isinstance(term2, str) and term2.isupper():
        # If term2 is already substituted, recurse
        if term2 in substitutions:
            return unify(term1, substitutions[term2], substitutions)
        elif occurs_check(term2, term1):
            raise UnificationError(f"Occurs check fails: {term2} in {term1}")
        else:
            substitutions[term2] = term1
            return substitutions

    # If both terms are compound (i.e., functions), unify their parts recursively
    elif isinstance(term1, tuple) and isinstance(term2, tuple):
        # Ensure that both terms have the same "functor" and number of arguments
        # if len(term1) != len(term2):
```

```
        #     raise UnificationError(f"Function arity mismatch: {term1} vs {term2}")

        for subterm1, subterm2 in zip(term1, term2):
            substitutions = unify(subterm1, subterm2, substitutions)

        return substitutions

    else:
        raise UnificationError(f"Cannot unify: {term1} with {term2}")

# Define the terms as tuples
term1 = ('p', 'b', 'X', ('f', ('g', 'Z')))
term2 = ('p', 'Z', ('f', 'Y'), ('f', 'Y'))

try:
    # Find the MGU
    result = unify(term1, term2)
    print("Most General Unifier (MGU):")
    print(result)
except UnificationError as e:
    print(f"Unification failed: {e}")

    Output :
    Most General Unifier (MGU):
    {'Z': 'b', 'X': ('f', 'Y'), 'Y': ('g', 'Z')}
```

Program 7

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

Algorithm :



Code :

```
def FOL_FC_ASK(KB, alpha):
    # Initialize the new sentences to be inferred in this iteration
    new = set()

    while new:  # Repeat until new is empty
        new = set()  # Clear new sentences on each iteration

        # For each rule in KB
        for rule in KB:
```

```
            # Standardize the rule variables to avoid conflicts
            standardized_rule = standardize_variables(rule)
            p1_to_pn, q = standardized_rule  # Premises (p1, ..., pn) and conclusion (q)

            # For each substitution θ such that Subst(θ, p1, ..., pn) matches the premises
            for theta in get_matching_substitutions(p1_to_pn, KB):
                q_prime = apply_substitution(theta, q)

                # If q_prime does not unify with some sentence already in KB or new
                if not any(unify(q_prime, sentence) != 'FAILURE' for sentence in KB.union(new)):
                    new.add(q_prime)  # Add q_prime to new

                    # Unify q_prime with the query (alpha)
                    phi = unify(q_prime, alpha)
                    if phi != 'FAILURE':
                        return phi  # Return the substitution if it unifies

        # Add newly inferred sentences to the knowledge base
        KB.update(new)

    return False  # Return false if no substitution is found


def standardize_variables(rule):
    """
    Standardize variables in the rule to avoid variable conflicts.
    Rule is assumed to be a tuple (premises, conclusion).
    """
    premises, conclusion = rule
    # Apply standardization logic here (for simplicity, assume no conflict in this case)
    return (premises, conclusion)


def get_matching_substitutions(premises, KB):
    """
    Get matching substitutions for the premises in the KB.
    This is a placeholder to represent how substitutions would be found.
    """
    # Implement substitution matching here
    return []  # This should return a list of valid substitutions


def apply_substitution(theta, expression):
    """
    Apply a substitution θ to an expression.
    This function will replace variables in expression with their corresponding terms from θ.
    """
```

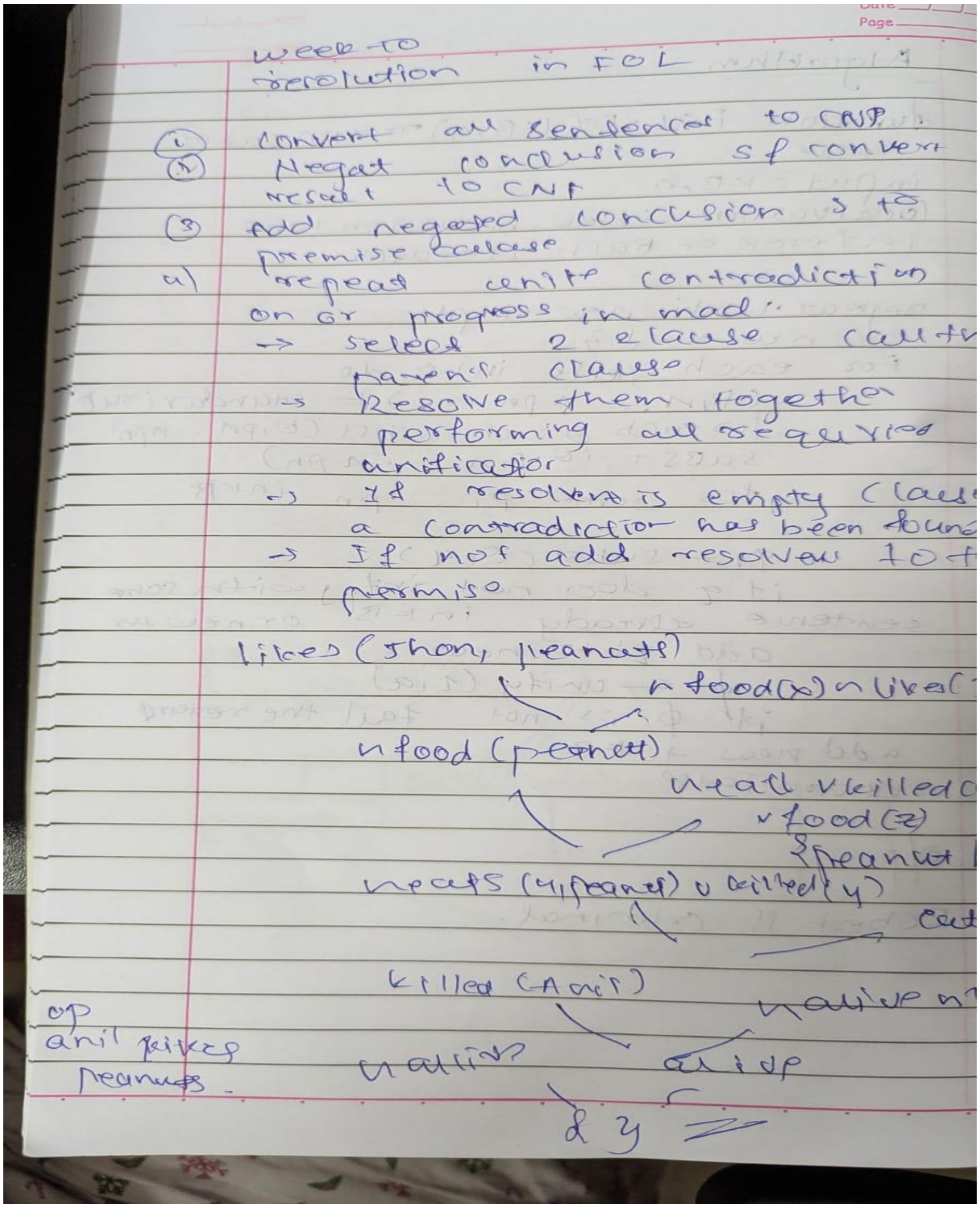
```
    if isinstance(expression, str) and expression.startswith('?'):
        return theta.get(expression, expression)  # Apply substitution to variable
    elif isinstance(expression, tuple):
        return tuple(apply_substitution(theta, arg) for arg in expression)
    return expression


def unify(psi1, psi2, subst=None):
    """Unification algorithm (simplified)"""
    if subst is None:
        subst = {}

    def apply_subst(s_map, expr):
        if isinstance(expr, str) and expr.startswith('?'):
            return s_map.get(expr, expr)
        elif isinstance(expr, tuple):
            return tuple(apply_subst(s_map, arg) for arg in expr)
        return expr

    def is_variable(expr):
        return isinstance(expr, str) and expr.startswith('?')

    _psi1 = apply_subst(subst, psi1)
    _psi2 = apply_subst(subst, psi2)

    if is_variable(_psi1) or is_variable(_psi2) or not isinstance(_psi1, tuple) or not isinstance(_psi2,
tuple):
        if _psi1 == _psi2:
            return subst
        elif is_variable(_psi1):
            if _psi1 in str(_psi2):
                return 'FAILURE'
            return {**subst, _psi1: _psi2}
        elif is_variable(_psi2):
            if _psi2 in str(_psi1):
                return 'FAILURE'
            return {**subst, _psi2: _psi1}
        else:
            return 'FAILURE'

    if _psi1[0] != _psi2[0] or len(_psi1) != len(_psi2):
        return 'FAILURE'

    for arg1, arg2 in zip(_psi1[1:], _psi2[1:]):
        s = unify(arg1, arg2, subst)
        if s == 'FAILURE':
            return 'FAILURE'
```

```
        subst = s

    return subst

# Knowledge Base (KB)
KB = set()

# Adding facts to KB:
KB.add(('american', 'Robert'))  # Robert is an American
KB.add(('hostile_nation', 'Country_A'))  # Country A is a hostile nation
KB.add(('sell_weapons', 'Robert', 'Country_A'))  # Robert sold weapons to Country A

# Adding the rule (the law):
KB.add((('american(x)', 'hostile_nation(y)', 'sell_weapons(x, y)'),
        'criminal(x)'))

# Goal: Prove that Robert is a criminal
goal = 'criminal(Robert)'

# Calling FOL_FC_ASK to prove the goal
result = FOL_FC_ASK(KB, goal)

if result != 'FAILURE':
    print("Robert is a criminal!")
else:
    print("Robert is not a criminal.")
```

Output :
Robert is a criminal!

Program 8

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



week-10
resolution in FOL

(1) convert all sentences to CNF
(2) Negat conclusion s& convert result to CNF
(3) Add negated conclusion s to premise clause
4) repeat until contradiction on or progress in mad:
→ select 2 clause call th parent clause
→ Resolve them together performing all required unification
→ If resolvent is empty clause a contradiction has been found
→ If not add resolven to th premiso

likes (Jhon, peanuts)
¬food(x) ∨ likes(
¬food (peanut)
¬eats ∨ killed
∨ food(z)
{peanut
¬eats (y, peanut) ∨ killed (y)
killed (Anil)
¬alive
¬alive
alive
op
anil likes peanuts

Code :

```
from itertools import combinations

def unify(x, y, theta=None):
    if theta is None:
        theta = {}
    if x == y:
```

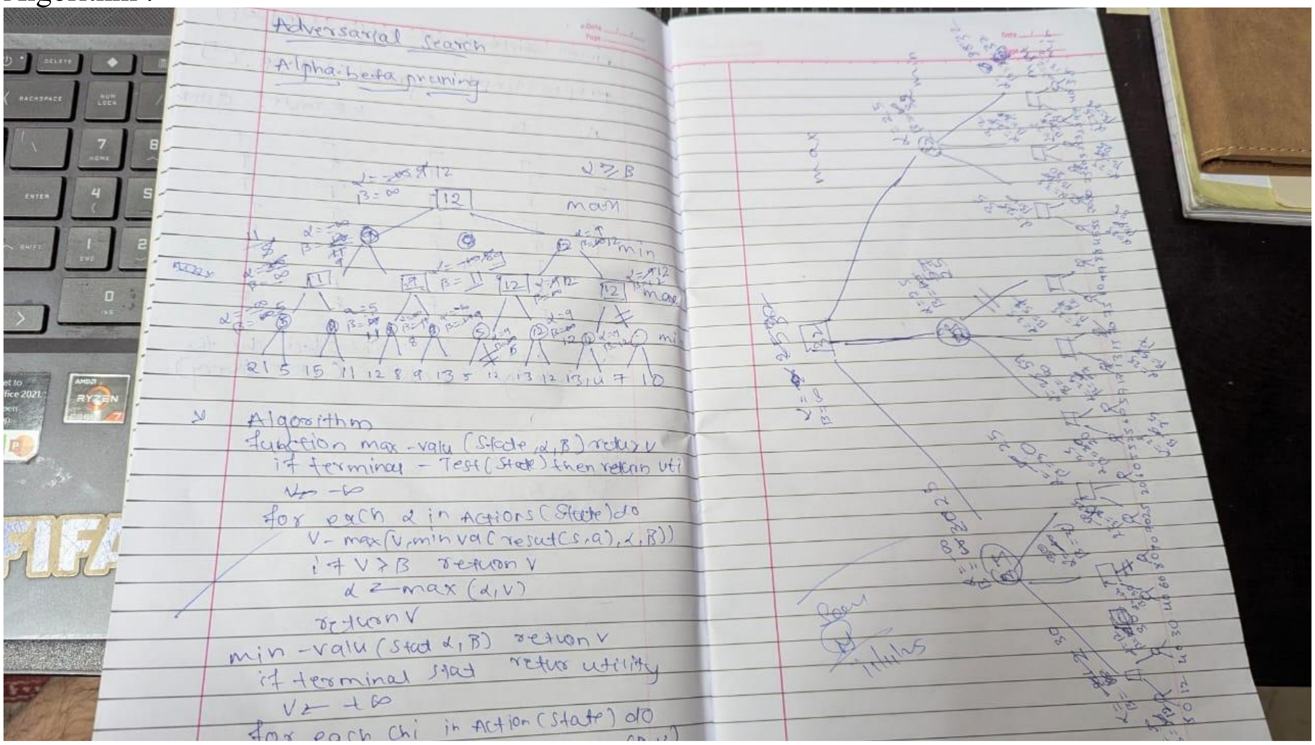
```
        return theta
    elif isinstance(x, str) and x.islower():
        return unify_var(x, y, theta)
    elif isinstance(y, str) and y.islower():
        return unify_var(y, x, theta)
    elif isinstance(x, tuple) and isinstance(y, tuple) and len(x) == len(y):
        return unify(x[1:], y[1:], unify(x[0], y[0], theta))
    else:
        return None

def unify_var(var, x, theta):
    if var in theta:
        return unify(theta[var], x, theta)
    elif x in theta:
        return unify(var, theta[x], theta)
    else:
        theta[var] = x
        return theta

def negate(predicate):
    if isinstance(predicate, tuple) and predicate[0] == 'not':
        return predicate[1]
    else:
        return ('not', predicate)

def substitute_predicate(predicate, theta):
    if isinstance(predicate, str):
        return theta.get(predicate, predicate)
    elif isinstance(predicate, tuple):
        return (predicate[0],) + tuple(theta.get(arg, arg) for arg in predicate[1:])
    return predicate

def substitute(clause, theta):
    return {substitute_predicate(p, theta) for p in clause}

def resolve(clause1, clause2):
    resolvents = []
    for p1 in clause1:
        for p2 in clause2:
            theta = unify(p1, negate(p2))
            if theta is not None:
                new_clause = (substitute(clause1, theta) | substitute(clause2, theta)) - {p1, p2}
                resolvents.append(frozenset(new_clause))
    return resolvents

def resolution(kb, query):
    negated_query = frozenset({negate(query)})
```

```python
    clauses = kb | {negated_query}
    new = set()

    while True:
        pairs = list(combinations(clauses, 2))
        for (ci, cj) in pairs:
            resolvents = resolve(ci, cj)
            if frozenset() in resolvents:
                return True
            new |= set(resolvents)
        if new.issubset(clauses):
            return False
        clauses |= new

# Knowledge Base
kb = {
    frozenset({('not', ('Food', 'x')), ('Likes', 'John', 'x')}),  # a
    frozenset({('Food', 'Apple')}),  # b
    frozenset({('Food', 'Vegetables')}),  # b
    frozenset({('not', ('Eats', 'x', 'y')), ('Killed', 'x'), ('Food', 'y')}),  # c
    frozenset({('Eats', 'Anil', 'Peanuts')}),  # d
    frozenset({('Alive', 'Anil')}),  # d
    frozenset({('not', ('Eats', 'Anil', 'x')), ('Eats', 'Harry', 'x')}),  # e
    frozenset({('not', ('Alive', 'x')), ('not', ('Killed', 'x'))}),  # f
    frozenset({('Killed', 'x'), ('Alive', 'x')}),  # g
}
query = ('Likes', 'John', 'Peanuts')

# Run resolution
result = resolution(kb, query)

if result:
    print("Proved by resolution: John likes peanuts.")
else:
    print("Cannot prove that John likes peanuts.")
```

Output :
Proved by resolution: John likes peanuts.

Program 9
Implement Alpha-Beta Pruning

Algorithm :



Code :

```
import numpy as np

class Node:
    def __init__(self, value=None, children=None):
        self.value = value
        self.children = children if children else []

def build_tree_from_array(arr):
    leaves = [Node(value=v) for v in arr.flatten()]
    num_leaves = len(leaves)


    while num_leaves > 1:
        new_level = []
        for i in range(0, num_leaves, 2):
            if i + 1 < num_leaves:
                new_level.append(Node(None, [leaves[i], leaves[i + 1]]))
            else:
```

```
            new_level.append(leaves[i])
        leaves = new_level
        num_leaves = len(leaves)

    return leaves[0]

def alpha_beta_pruning(node, depth, alpha, beta, maximizing_player):
    if not node.children or depth == 0:
        return node.value

    if maximizing_player:
        max_eval = float('-inf')
        for child in node.children:
            eval = alpha_beta_pruning(child, depth - 1, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                print(f"Pruned at MAX node (α={alpha}, β={beta})")
                break
        node.value = max_eval
        return max_eval
    else:
        min_eval = float('inf')
        for child in node.children:
            eval = alpha_beta_pruning(child, depth - 1, alpha, beta, True)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                print(f"Pruned at MIN node (α={alpha}, β={beta})")
                break
        node.value = min_eval
        return min_eval

def print_tree(node, level=0):
    print("  " * level + f"Node Value: {node.value}")
    for child in node.children:
        print_tree(child, level + 1)

if __name__ == "__main__":
    tree_array = np.array([
        [10, 9],
        [14, 18],
        [5, 4],
        [50, 3]
    ])

    root = build_tree_from_array(tree_array)
```

```
print("Game Tree Before Alpha-Beta Pruning:")
print_tree(root)

depth = int(np.log2(tree_array.size))
final_value = alpha_beta_pruning(root, depth, alpha=float('-inf'), beta=float('inf'),
maximizing_player=True)

print("\nGame Tree After Alpha-Beta Pruning:")
print_tree(root)

print(f"\nFinal Value at MAX node: {final_value}")
```

Output :

```
Game Tree Before Alpha-Beta Pruning:
Node Value: None
  Node Value: None
    Node Value: None
      Node Value: 10
      Node Value: 9
    Node Value: None
      Node Value: 14
      Node Value: 18
  Node Value: None
    Node Value: None
      Node Value: 5
      Node Value: 4
    Node Value: None
      Node Value: 50
      Node Value: 3
Pruned at MAX node (α=14, β=10)
Pruned at MIN node (α=10, β=5)

Game Tree After Alpha-Beta Pruning:
Node Value: 10
  Node Value: 10
    Node Value: 10
      Node Value: 10
      Node Value: 9
    Node Value: 14
      Node Value: 14
      Node Value: 18
  Node Value: 5
    Node Value: 5
      Node Value: 5
      Node Value: 4
    Node Value: None
```

Node Value: 50
Node Value: 3

Final Value at MAX node: 10