

**VISVESVARAYA TECHNOLOGICAL  
UNIVERSITY**

“JnanaSangama”, Belgaum -590014, Karnataka.



**LAB REPORT**

**on**

**Artificial Intelligence (23CS5PCAIN)**

*Submitted by*

**Rachana H D (1BM22CS212)**

*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**

**Sep-2024 to Jan-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 **Rachana H D (1BM22CS212)**, who is bona fide 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.

Radhika A D Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE
----------------------------------------------------------------	------------------------------------------------------------------

## Index

<b>Sl. No.</b>	<b>Date</b>	<b>Experiment Title</b>	<b>Page No.</b>
1	30-9-2024	Implement Tic –Tac –Toe Game Implement vacuum cleaner agent	1-8
2	7-10-2024	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	8-15
3	14-10-2024	Implement 8 puzzle using A* search algorithm	16-20
4	21-10-2024	Implement Hill Climbing search algorithm to solve N-Queens problem	21-23
5	28-10-2024	Simulated Annealing to Solve 8-Queens problem	23-25
6	11-11-2024	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	25-27
7	2-12-2024	Implement unification in first order logic	28-31
8	2-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	32-33
9	16-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	34-35
10	16-12-2024	Implement Alpha-Beta Pruning.	36-38

## Program 1

Implement Tic - Tac - Toe Game  
Implement vacuum cleaner agent

Tic Tac Toe:  
Algorithm:

```

PAGE NO: _____
DATE: _____

if move < 0 or move >
    raise ValueError
row, col = divmod(move, 3)
if board[row][col] == " ":
    return row, col
else:
    print("Invalid input, please enter a
        number 1 and 9 .")

def computer_move(board):
    for move in get_available_moves(board):
        board[move[0]][move[1]] = "O"
        if check_winner(board) == "O":
            return move
    return random.choice(get_available_moves(board))

def main():
    board = [[None for i in range(3)] for j in range(3)]
    while True:
        print_board(board)
        row, col = human_move(board)
        board[row][col] = "X"
        if check_winner(board) == "X":
            print_board(board)
            print("congratulations! you win!")
            break

```

```

PAGE NO: _____
DATE: _____

if is_board_full(board):
    print_board(board)
    print("It's a tie!")
    break

print("computer's turn : ")
row, col = computer_move(board)
board[row][col] = "O"

if check_winner(board) == "O":
    print_board(board)
    if name == "main":
        main()

output
      O | X   O | X   O | X   O | X
      | |   | |   | |   | |
      | |   | |   | |   | |
      win
from
main

```

Code:

```

board=[[['1','1','1'],['1','1','1'],['1','1','1']]]

def check(board,user):
    for i in range(3):
        if(board[0][i]==user and board[1][i]==user and board[2][i]==user):
            return True
        if(board[i][0]==user and board[i][1]==user and board[i][2]==user):
            return True
        if(board[0][0]==user and board[1][1]==user and board[2][2]==user):
            return True
        if(board[0][2]==user and board[1][1]==user and board[2][0]==user):
            return True
    return False

```

```

def show(board):
    for b in board:

```

```

print(b)

def full(board):
    for i in range(3):
        for j in range(3):
            if(board[i][j] == '1'):
                return False
    return True

user=0

user1=input("Enter user name:")
user2=input("Enter user name:")

while True :

    if (full(board)) :
        print("Draw")
        break

    if(user==0):

        show(board)
        print(user1 + " play")
        row=int(input("Enter row:"))
        col=int(input("Enter col:"))

        if(board[row][col]=='1'):
            board[row][col]='X'
        else:
            print("Wrong!")
            continue

        if(check(board,'X')):
            print(user1 + " won!")
            break
        else:
            user=1

    if(full(board)):
        print("Draw")
        break

    if(user==1):

        show(board)

        print(user2 + " play")
        row=int(input("Enter row:"))
        col=int(input("Enter col:"))

        if(board[row][col]=='1'):
            board[row][col]='O'
        else:

```

```
print("Wrong!")
continue

if(check(board, '0')):
    print(user2 + " won!")
    break
else:
    user=0

if full(board):
    print("Draw")
    break
```

```

Enter user name:shreya
Enter user name:sakshi
['1', '1', '1']
['1', '1', '1']
['1', '1', '1']
shreya play
Enter row:0
Enter col:0
['X', '1', '1']
['1', '1', '1']
['1', '1', '1']
sakshi play
Enter row:0
Enter col:1
['X', '0', '1']
['1', '1', '1']
['1', '1', '1']
shreya play
Enter row:1
Enter col:1
['X', '0', '1']
['1', 'X', '1']
['1', '1', '1']
sakshi play
Enter row:2
Enter col:2
['X', '0', '1']
['1', 'X', '1']
['1', '1', '0']
shreya play
Enter row:1
Enter col:2
['X', '0', '1']
['1', 'X', 'X']
['1', '1', '0']
sakshi play
Enter row:1
Enter col:0
['X', '0', '1']
['0', 'X', 'X']
['1', '1', '0']
shreya play
Enter row:0
Enter col:2
['X', '0', 'X']
['0', 'X', 'X']
['1', '1', '0']
sakshi play
Enter row:2
Enter col:0
['X', '0', 'X']
['0', 'X', 'X']
['0', '1', '0']
shreya play
Enter row:2
Enter col:1
Draw

```

```

Enter user name:shreya
Enter user name:santosh
['1', '1', '1']
['1', '1', '1']
['1', '1', '1']
shreya play
Enter row:1
Enter col:1
['1', '1', '1']
['1', 'X', '1']
['1', '1', '1']
santosh play
Enter row:0
Enter col:0
['0', '1', '1']
['1', 'X', '1']
['1', '1', '1']
shreya play
Enter row:0
Enter col:1
['0', 'X', '1']
['1', 'X', '1']
['1', '1', '1']
santosh play
Enter row:1
Enter col:0
['0', 'X', '1']
['0', 'X', '1']
['1', '1', '1']
shreya play
Enter row:2
Enter col:1
Enter row:2
Enter col:1
shreya won!

```

## Vacuum World:

Algorithm:

Percept sequence	action
A [dirty] B [dirty]	Clean A, move to B
A [clean] B [dirty]	Clean B, move to A
A [dirty] B [clean]	Clean A, move to B
A [clean] B [clean]	No successful cleaning.

Code:

```

cost=0

def vacuum(state1, state2, loc):
    global cost
    if state1 == "0" and state2 == "0":
        print("All done")
        return

    if loc == "A":
        if state1 == "1" and state2=="1":
            print("Cleaned A")
            cost= cost+1
            state1 = "0"
            state1 = input("Is A dirty again?: ")
            vacuum(state1,state2,"A")
        elif state1=="1" and state2=="0":
            print("Cleaned A")
            cost= cost+1
            state1 = "0"
            state1 = input("Is A dirty again?: ")
            state2 = input("Is B dirty again?: ")
            vacuum(state1,state2,"A")
        elif state1=="0" and state2=="1":
            print("Moving to B")
            loc="B"
            vacuum(state1,state2,loc)

    elif loc == "B":
```

Percept sequence	action
A [dirty] B [dirty]	Clean A, move to B
A [clean] B [dirty]	Clean B, move to A
A [dirty] B [clean]	Clean A, move to B
A [clean] B [clean]	No successful cleaning.

Step 1: Start.

Step 2: Select two rooms A and B.

Step 3: Take input from the user about clean/dirty status of the room and location.

Step 4: If A is clean move to Room B  
If A is dirty suck the dirt  
  
If B is clean move to Room A  
If B is dirty suck the dirt.  
  
Do it until both Room A and Room B becomes clean.

Step 5: End

Initial position A.

Room (location)	Status	Action
A	dirty	Clean (suck)
B	clean	Move to B
B	dirty	Suck
A	clean	Move to A

```

if state1 == "1" and state2=="1":
    print("Cleaned B")
    cost= cost+1
    state2 = "0"
    state2 = input("Is A dirty again?: ")
    vacuum(state1,state2,"B")
elif state1=="0" and state2=="1":
    print("Cleaned B")
    cost= cost+1
    state1 = "0"
    state1 = input("Is A dirty again?: ")
    state2 = input("Is B dirty again?: ")
    vacuum(state1,state2,"A")
elif state1=="1" and state2=="0":
    print("Moving to B")
    loc="B"
    vacuum(state1,state2,loc)

```

```

print("Enter both states and location of vacuum")
state1 = input("Enter state 1 (0 or 1): ")
state2 = input("Enter state 2 (0 or 1): ")
loc = input("Enter loc (A or B): ")
vacuum(state1, state2, loc)
print("Total cost " + str(cost))

```

→ Enter both states and location of vacuum  
 Enter state 1 (0 or 1): 1  
 Enter state 2 (0 or 1): 1  
 Enter loc (A or B): A  
 Cleaned A  
 Is A dirty again?: 1  
 Cleaned A  
 Is A dirty again?: 0  
 Moving to B  
 Cleaned B  
 Is A dirty again?: 0  
 Is B dirty again?: 1  
 Moving to B  
 Cleaned B  
 Is A dirty again?: 0  
 Is B dirty again?: 0  
 All done  
 Total cost 4

## Program 2

Implement 8 puzzle problems using Depth First Search (DFS)

Implement Iterative deepening search algorithm

8 puzzle using DFS:

Algorithm:

1	2	3
4	5	6
7	8	-

Algorithm:

Step 1: Choose 1 move (left, right, up or down)  
if it is legal and can be moved

Step 2: Compare the move with the goal state  
after every move, if it matches the goal  
state is reached if not, make the next  
move.

Step 3: If the state is reached previously  
then mark it visited.

Step 4: After every move if the state is not  
visited and the move is legal the  
state is reached.

classmate  
Date \_\_\_\_\_  
Page \_\_\_\_\_

	2	8	3	$g=0$	1	2	3
	1	6	4	$h=4$	8	4	
	7	5			6	5	

$L \quad U \quad R$

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

$g(n)=1 \quad g(n)=1 \quad g(n)=1$

$h(n)=6 \quad h(n)=3 \quad h(n)=6$

$f(n)=6 \quad f(n)=4 \quad f(n)=6$

$L \quad R \quad U \quad P$

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

$g(n)=2 \quad g(n)=2 \quad g(n)=2 \quad g(n)=2$

$h(n)=3 \quad h(n)=4 \quad h(n)=3 \quad h(n)=4$

$f(n)=5 \quad f(n)=6 \quad f(n)=5 \quad f(n)=6$

$(\text{visited})$

$U \quad D \quad R \quad g(n)=2 \quad P \quad L \quad R$

8	3	2	8	3	2	8	3	2	3	-
2	1	4	7	1	4	1	4	1	8	4
7	6	5	6	5v	7	6	5	7	6	5v

$h(n)=3 \quad (4) \quad (3) \quad (3) \quad (2) \quad (4)$

$f(n)=5$

$g(n)=5$

1	2	3	1	2	3	1	2	3
8	4		8	v	1	8	4	

$\leftarrow R$

```

Code:
count=0;
def print_state(in_array):
    global count
    count+=1
    for row in in_array:
        print(' '.join(str(num) for num in row))
    print()

def helper(goal, in_array, row, col, vis):
    # Marking current position as visited
    vis[row][col] = 1
    drow = [-1, 0, 1, 0] # Dir for row : up, right, down, left
    dcol = [0, 1, 0, -1] # Dir for column
    dchange = ['Up', 'Right', 'Down', 'Left']

    # Print current state
    print("Current state:")
    print_state(in_array)

    # Check if the current state is the goal state
    if in_array == goal:
        print_state(in_array)
        print(f"Number of states:{cnt}")
        return True

    # Explore all possible directions
    for i in range(4):
        nrow = row + drow[i]
        ncol = col + dcol[i]

        # Check if the new position is within bounds and not visited
        if 0 <= nrow < len(in_array) and 0 <= ncol < len(in_array[0]) and not
vis[nrow][ncol]:
            # Make the move (swap the empty space with the adjacent tile)
            print(f"Took a {dchange[i]} move")
            in_array[row][col], in_array[nrow][ncol] = in_array[nrow][ncol],
in_array[row][col]

            # Recursive call
            if helper(goal, in_array, nrow, ncol, vis):
                return True

            # Backtrack (undo the move)
            in_array[row][col], in_array[nrow][ncol] = in_array[nrow][ncol],
in_array[row][col]

    # Mark the position as unvisited before returning
    vis[row][col] = 0
    return False

# Example usage

```

```
initial_state = [[1, 2, 3], [0, 4, 6], [7, 5, 8]] # 0 represents the empty space
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
visited = [[0] * 3 for _ in range(3)] # 3x3 visited matrix
empty_row, empty_col = 1, 0 # Initial position of the empty space

found_solution = helper(goal_state, initial_state, empty_row, empty_col, visited)
print("Solution found:", found_solution)
```

✓ Current state:  
0s 1 2 3  
 4 6 8  
7 5 0

Took a Left move  
Current state:  
1 2 3  
4 6 8  
7 0 5

Took a Left move  
Current state:  
1 2 3  
4 6 8  
0 7 5

Took a Down move  
Current state:  
1 2 3  
4 5 6  
7 0 8

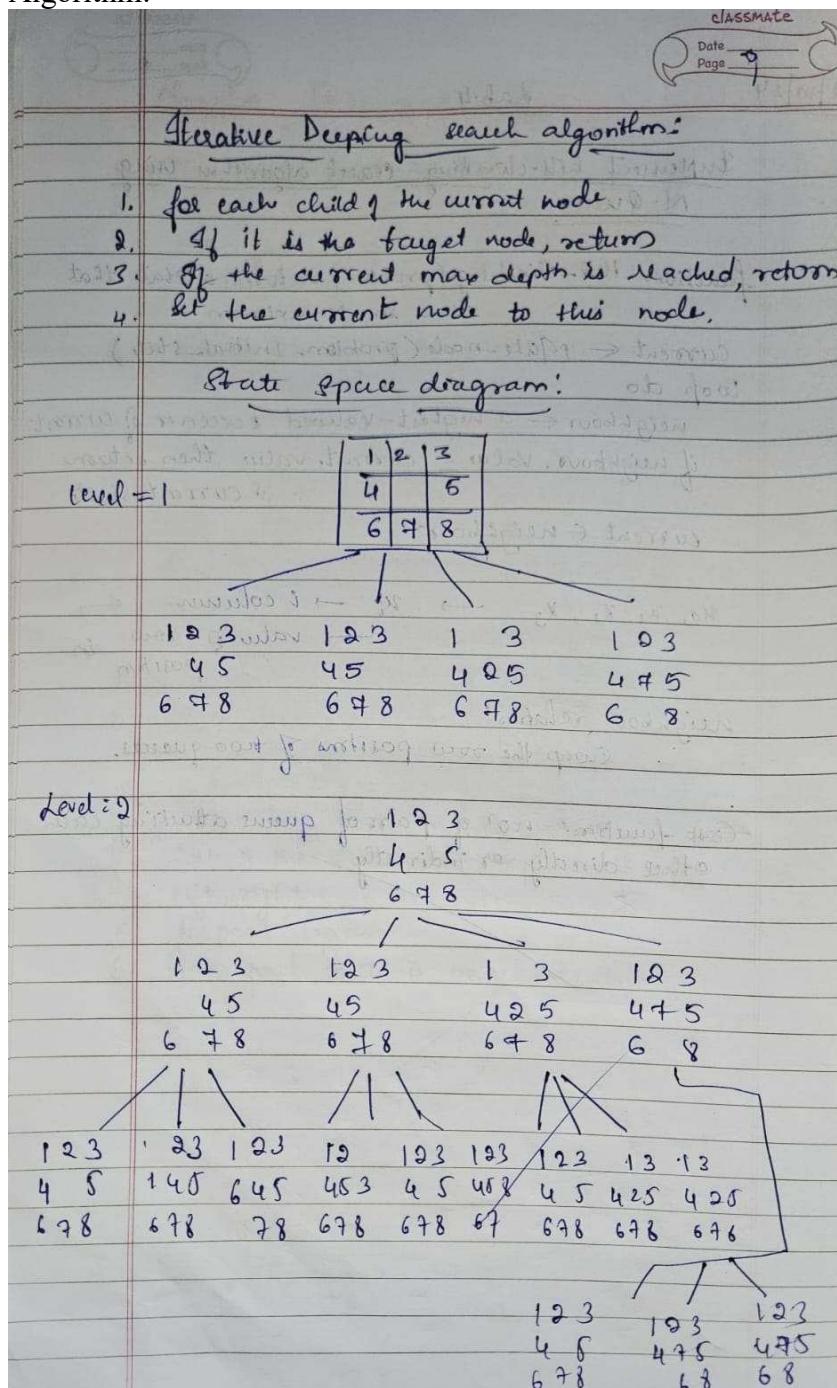
Took a Right move  
Current state:  
1 2 3  
4 5 6  
7 8 0

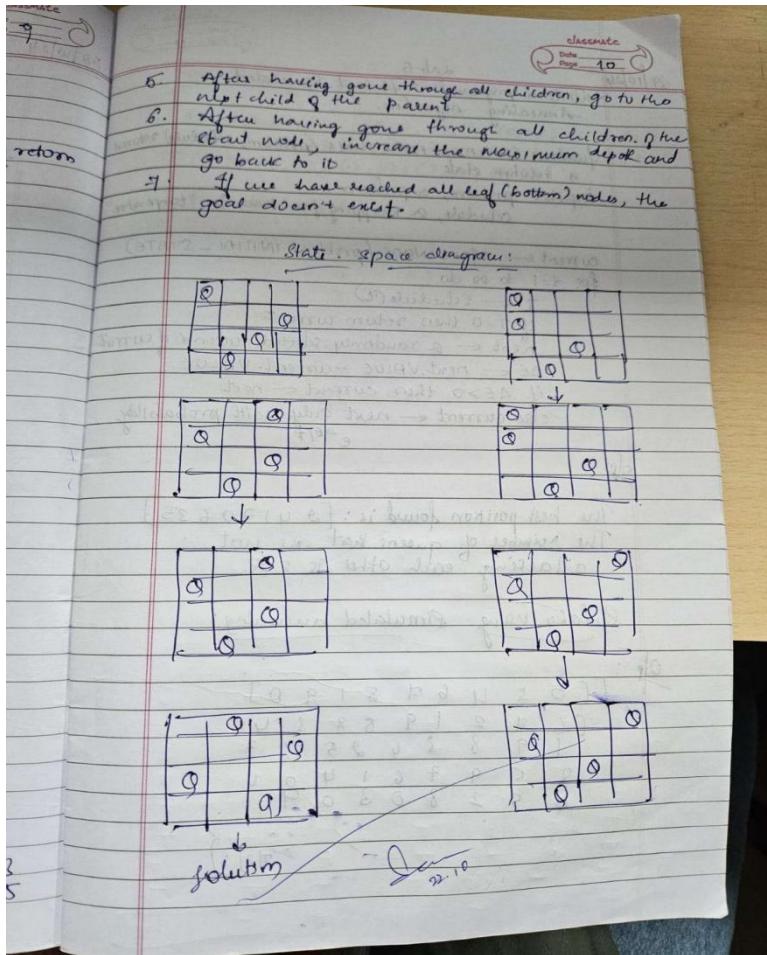
1 2 3  
4 5 6  
7 8 0

Number of states:42  
Solution found: True

Iterative deepening search algorithm:

### Algorithm:





Code:

```
#iterative-deepening
from collections import deque

class PuzzleState:
    def __init__(self, board, zero_pos, moves=0, previous=None):
        self.board = board
        self.zero_pos = zero_pos # Position of the zero tile
        self.moves = moves # Number of moves taken to reach this state
        self.previous = previous # For tracking the path

    def is_goal(self, goal_state):
        return self.board == goal_state

    def get_possible_moves(self):
        moves = []
        x, y = self.zero_pos
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)] # Up, Down, Left, Right
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = [row[:] for row in self.board]
                # Swap the zero tile with the adjacent tile
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y], new_board[x][y]
                moves.append((new_board, (new_x, new_y)))

```

```

    return moves

def ids(initial_state, goal_state, max_depth):
    for depth in range(max_depth):
        visited = set()
        result = dls(initial_state, goal_state, depth, visited)
        if result:
            return result
    return None

def dls(state, goal_state, depth, visited):
    if state.is_goal(goal_state):
        return state
    if depth == 0:
        return None

    visited.add(tuple(map(tuple, state.board))) # Mark this state as visited
    for new_board, new_zero_pos in state.get_possible_moves():
        new_state = PuzzleState(new_board, new_zero_pos, state.moves + 1, state)
        if tuple(map(tuple, new_board)) not in visited:
            result = dls(new_state, goal_state, depth - 1, visited)
            if result:
                return result
    visited.remove(tuple(map(tuple, state.board))) # Unmark this state
    return None

def print_solution(solution):
    path = []
    while solution:
        path.append(solution.board)
        solution = solution.previous
    for board in reversed(path):
        for row in board:
            print(row)
        print()

# Define the initial state and goal state
initial_state = PuzzleState(
    board=[[1, 2, 3],
           [4, 0, 5],
           [7, 8, 6]],
    zero_pos=(1, 1)
)

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

# Perform Iterative Deepening Search
max_depth = 20 # You can adjust this value
solution = ids(initial_state, goal_state, max_depth)

if solution:

```

```
    print("Solution found:")
    print_solution(solution)
else:
    print("No solution found.")
```

→ Solution found:

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

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

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

### **Program 3**

Implement A\* search algorithm

Algorithm:

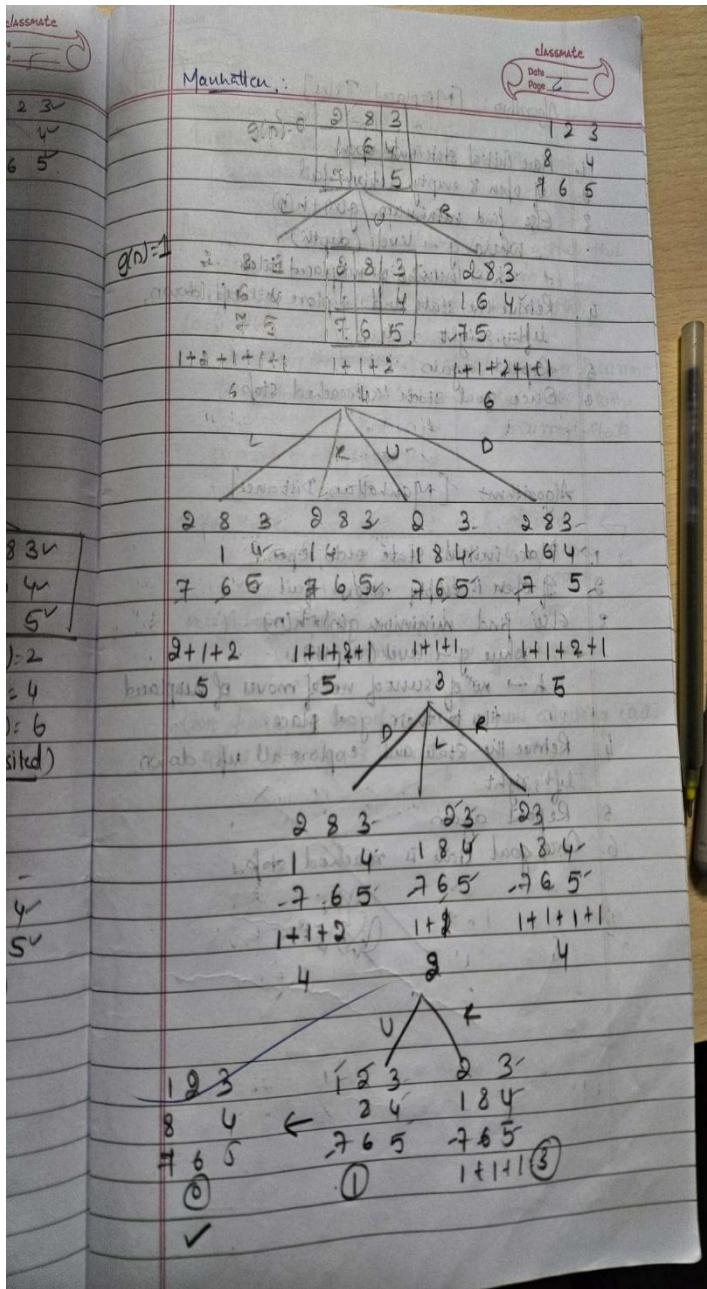
CLASSMATE  
Date \_\_\_\_\_  
Page 7

Algorithm: [Misplaced Tiles]

1. Place initial state onto open.
2. If open is empty return fail
3. Else find minimum  $g(n) + h(n)$ 
  - where  $g$  - level (depth)
  - $h$  - number of misplaced tiles
4. Retrive the state and explore all up, down, left, right.
5. Repeat again
6. Once goal state is reached stop.

Algorithm: [Manhattan Distance]

1. Place initial state onto Open.
2. If open is empty return fail



Code:

Misplaced Tiles

def mistil(state, goal):

    count = 0

    for i in range(3):

        for j in range(3):

            if state[i][j] != goal[i][j]:

                count += 1

    return count

def findmin(open\_list, goal):

    minv = float('inf')

    best\_state = None

```

for state in open_list:
    h = mistil(state['state'], goal)
    f = state['g'] + h
    if f < minv:
        minv = f
        best_state = state
open_list.remove(best_state)
return best_state

def operation(state):
    next_states = []
    blank_pos = find_blank_position(state['state'])
    for move in ['up', 'down', 'left', 'right']:
        new_state = apply_move(state['state'], blank_pos, move)
        if new_state:
            next_states.append({
                'state': new_state,
                'parent': state,
                'move': move,
                'g': state['g'] + 1
            })
    return next_states

def find_blank_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return None

def apply_move(state, blank_pos, move):
    i, j = blank_pos
    new_state = [row[:] for row in state]
    if move == 'up' and i > 0:
        new_state[i][j], new_state[i - 1][j] = new_state[i - 1][j], new_state[i][j]
    elif move == 'down' and i < 2:
        new_state[i][j], new_state[i + 1][j] = new_state[i + 1][j], new_state[i][j]
    elif move == 'left' and j > 0:
        new_state[i][j], new_state[i][j - 1] = new_state[i][j - 1], new_state[i][j]
    elif move == 'right' and j < 2:
        new_state[i][j], new_state[i][j + 1] = new_state[i][j + 1], new_state[i][j]
    else:
        return None
    return new_state

def print_state(state):
    for row in state:

```

```

print(''.join(map(str, row)))

initial_state = [[2,8,3], [1,6,4], [7,0,5]]
goal_state = [[1,2,3], [8,0,4], [7,6,5]]
open_list = [{ 'state': initial_state, 'parent': None, 'move': None, 'g': 0}]
visited_states = []

while open_list:
    best_state = findmin(open_list, goal_state)
    print("Current state:")
    print_state(best_state['state'])
    h = mistil(best_state['state'], goal_state)
    f = best_state['g'] + h
    print(f'g(n): {best_state["g"]}, h(n): {h}, f(n): {f}')
    if best_state['move'] is not None:
        print(f'Move: {best_state["move"]}')
    print()
    if mistil(best_state['state'], goal_state) == 0:
        goal_state_reached = best_state
        break
    visited_states.append(best_state['state'])
    next_states = operation(best_state)
    for state in next_states:
        if state['state'] not in visited_states:
            open_list.append(state)

moves = []
while goal_state_reached['move'] is not None:
    moves.append(goal_state_reached['move'])
    goal_state_reached = goal_state_reached['parent']
moves.reverse()

print("\nMoves to reach the goal state:", moves)
print("\nGoal state reached:")
print_state(goal_state)

```

```

Current state:
2 8 3
1 6 4
7 0 5
g(n): 0, h(n): 5, f(n): 5

Current state:
2 8 3
1 0 4
7 6 5
g(n): 1, h(n): 3, f(n): 4
Move: up

Current state:
2 0 3
1 8 4
7 6 5
g(n): 2, h(n): 4, f(n): 6
Move: up

Current state:
2 8 3
0 1 4
7 6 5
g(n): 2, h(n): 4, f(n): 6
Move: left

Current state:
0 2 3
1 8 4
7 6 5
g(n): 3, h(n): 3, f(n): 6
Move: left

Current state:
1 2 3
0 8 4
7 6 5
g(n): 4, h(n): 2, f(n): 6
Move: down

Current state:
1 2 3
8 0 4
7 6 5
g(n): 5, h(n): 0, f(n): 5
Move: right

Moves to reach the goal state: ['up', 'up', 'left', 'down', 'right']

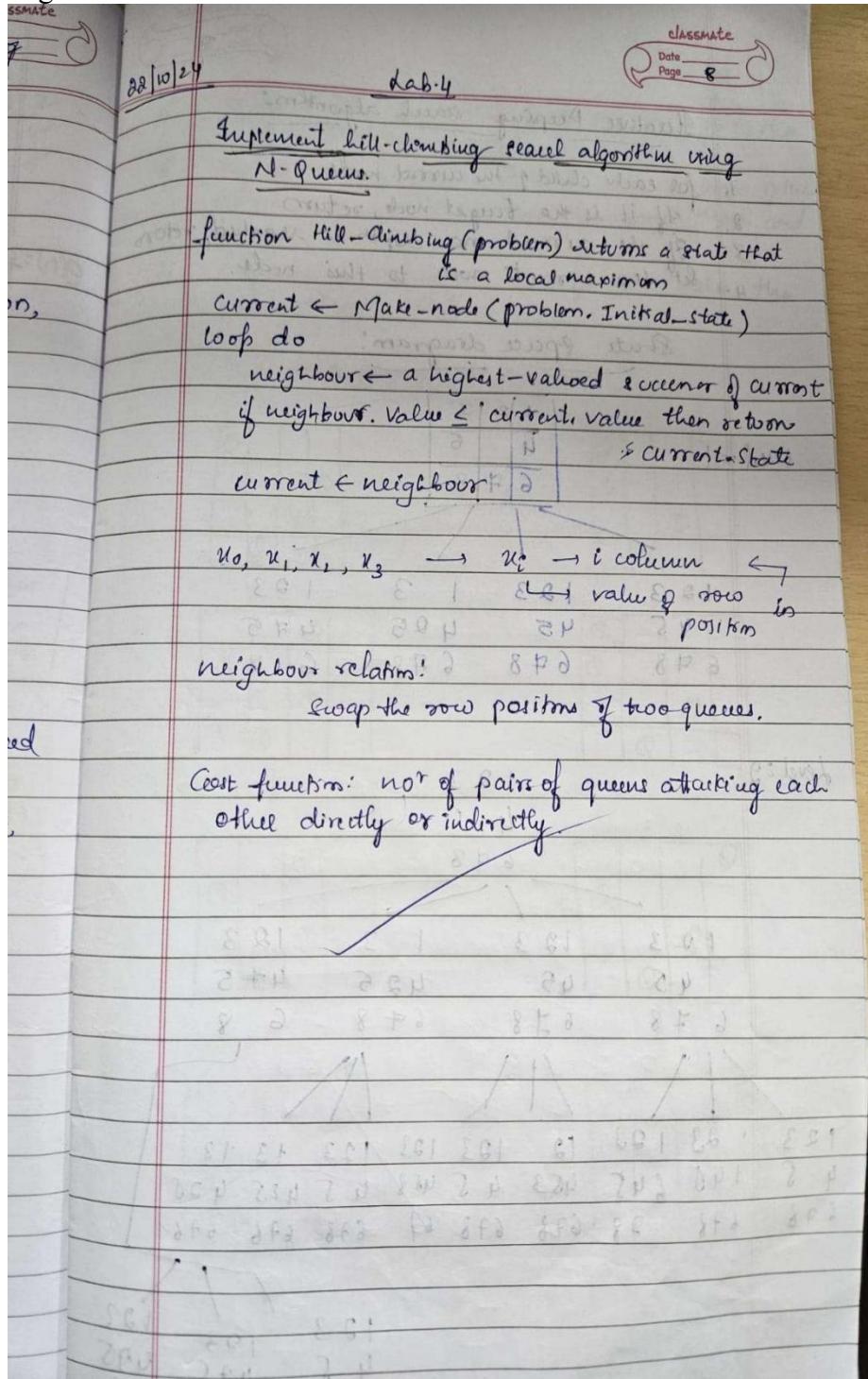
Goal state reached:
1 2 3
8 0 4
7 6 5

```

## Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

### Algorithm:



## Code:

```
import random
```

```
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]) == abs(i - j):
            conflicts += 1
return conflicts

```

```

def hill_climbing(n):
    cost=0
    while True:
        # Initialize a random board
        current_board = list(range(n))
        random.shuffle(current_board)
        current_conflicts = calculate_conflicts(current_board)

```

```

        while True:
            # Generate neighbors by moving each queen to a different position
            found_better = False
            for i in range(n):
                for j in range(n):
                    if j != current_board[i]: # Only consider different positions
                        neighbor_board = list(current_board)
                        neighbor_board[i] = j
                        neighbor_conflicts = calculate_conflicts(neighbor_board)

```

```

                        if neighbor_conflicts < current_conflicts:
                            current_board = neighbor_board
                            current_conflicts = neighbor_conflicts
                            cost+=1
                            found_better = True
                            break
            if found_better:
                break

```

```

        # If no better neighbor found, stop searching
        if not found_better:
            break

```

```

    # If a solution is found (zero conflicts), return the board
    if current_conflicts == 0:
        return current_board, current_conflicts, cost

```

```

def print_board(board):
    n = len(board)
    for i in range(n):
        row = ['.']*n
        row[board[i]] = 'Q' # Place a queen
        print(' '.join(row))
    print()

```

```

# Example Usage
n = 4
solution, conflicts, cost = hill_climbing(n)
print("Final Board Configuration:")
print_board(solution)

```

```
print("Number of Cost:", cost)
```

→ Final Board Configuration:

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

Number of Cost: 32

## Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:

29/10/24 Date: 29/10/24 Page: 44

dab-5

Write a program to implement Simulated Annealing algorithm.

function SIMULATED\_ANNEALING (problem, schedule) returns a solution state

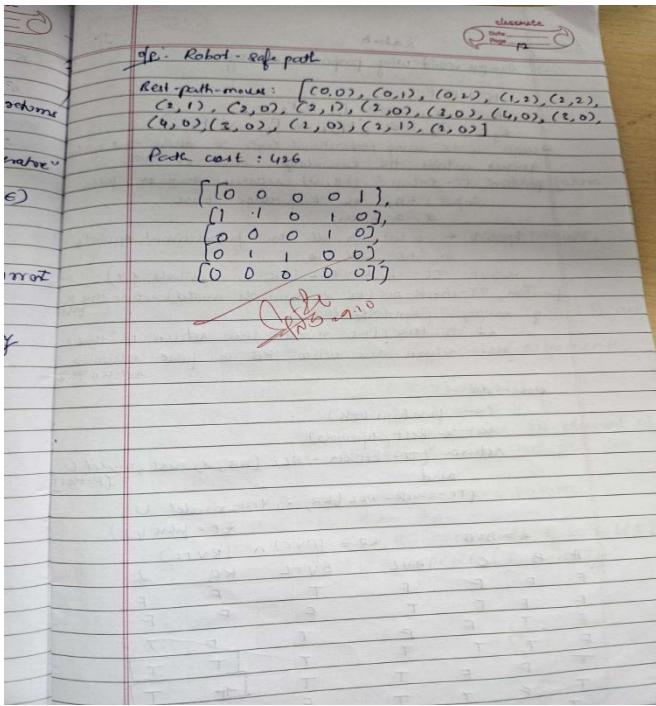
inputs: problem, a problem  
schedule, a mapping from time to "temperature"

current ← MAKE-NODE (problem, INITIAL-STATE)  
for t=1 to do do  
    T ← schedule(t)  
    if T=0 then return current  
    next ← a randomly selected successor of current  
    ΔE ← next.VALUE - current.VALUE  
    if ΔE > 0 then current ← next  
    else current ← next only with probability  $e^{\Delta E/T}$

The best position found is: [9 4 1 7 0 6 3 5]  
The number of queens that are not attacking each other is: 8.

Sudoku using simulated annealing:

of :  
[[5, 3, 4, 6, 7, 8, 1, 9, 0],  
[6, 7, 2, 1, 9, 5, 8, 3, 4],  
[1, 9, 8, 3, 4, 2, 5, 6, 7],  
[8, 6, 9, 7, 1, 4, 2, 0, 3],  
[4, 5, 2, 8, 0, 3, 1, 9, 6],  
[7, 1, 3, 9, 5, 8, 4, 2, 0],  
[2, 8, 1, 6, 9, 7, 3, 4, 5],  
[3, 4, 5, 2, 8, 1, 9, 6, 7],  
[9, 6, 8, 4, 1, 3, 7, 5, 2]]



Code:

```

import numpy as np
from scipy.optimize import dual_annealing

def queens_max(position):
    # This function calculates the number of pairs of queens that are not attacking each other
    position = np.round(position).astype(int) # Round and convert to integers for queen positions
    n = len(position)
    queen_not_attacking = 0

    for i in range(n - 1):
        no_attack_on_j = 0
        for j in range(i + 1, n):
            # Check if queens are on the same row or on the same diagonal
            if position[i] != position[j] and abs(position[i] - position[j]) != (j - i):
                no_attack_on_j += 1
            if no_attack_on_j == n - 1 - i:
                queen_not_attacking += 1
        if queen_not_attacking == n - 1:
            queen_not_attacking += 1
    return -queen_not_attacking # Negative because we want to maximize this value

# Bounds for each queen's position (0 to 7 for an 8x8 chessboard)
bounds = [(0, 7) for _ in range(8)]

# Use dual_annealing for simulated annealing optimization
result = dual_annealing(queens_max, bounds)

# Display the results
best_position = np.round(result.x).astype(int)
best_objective = -result.fun # Flip sign to get the number of non-attacking queens

print('The best position found is:', best_position)

```

```
print('The number of queens that are not attacking each other is:', best_objective)
```

→ The best position found is: [2 4 1 7 0 6 3 5]  
 The number of queens that are not attacking each other is: 8

## Program 6

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

Algorithm:

Lab-6

19/11/24

→ Output world using propositional logic

$P_1 : \neg P_{1,1}$   
 $P_2 : P_{1,1} \Leftrightarrow P_{2,1} \vee P_{2,2}$   
 $P_3 : P_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{2,3})$

→ Create a KB using propositional logic & show that given queries entails the knowledge base or not.

function  $\text{TT\_entails?}(KB, \psi)$  returns true or false

Input: KB, the knowledge base  
 $\alpha$ , a query

symbols ← a list of all the propositional symbols in KB and  $\alpha$

return  $\text{TT\_check\_ALL}(KB, \alpha, \text{symbols}, \text{model})$

function  $\text{TT\_check\_ALL}(KB, \alpha, \text{symbols}, \text{model})$  return true or false

if empty? ( $\text{symbols}$ ) then  
 if  $\alpha \models \text{true?}(KB, \text{model})$  then return  $\text{true}$   
 else return  $\text{true}$  when KB is false always return  $\text{true}$

else do -  
 $\beta \leftarrow \text{first}(\text{symbols})$   
 rest ←  $\text{rest}(\text{symbols})$   
 return  $(\text{TT\_CHECK\_ALL}(KB, \alpha, \text{rest}, \text{model}) \wedge$   
 and  
 $\{\beta \models \text{true}\})$

$\text{TT\_check\_ALL}(KB, \alpha, \text{rest}, \text{model}) \vee$   
 $\{\beta \models \text{false}\})$

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

A	B	C	AVL	BVTL	KB	$\alpha$
F	F	F	F	T	F	F
F	F	T	T	F	F	F
F	T	F	F	T	F	T
F	T	T	T	T	T	T
T	F	F	T	T	F	T
T	F	T	T	F	F	T
T	T	F	T	T	T	T

Code:

```
import itertools

# Function to evaluate an expression
def evaluate_expression(a, b, c, expression):
    # Use eval() to evaluate the logical expression
```

```

    return eval(expression)

# Function to generate the truth table and evaluate a logical expression
def truth_table_and_evaluation(kb, query):
    # All possible combinations of truth values for a, b, and c
    truth_values = [True, False]
    combinations = list(itertools.product(truth_values, repeat=3))

    # Reverse the combinations to start from the bottom (False -> True)
    combinations.reverse()

    # Header for the full truth table
    print(f"{'a':<5} {'b':<5} {'c':<5} {'KB':<20}{Query':<20}")

    # Evaluate the expressions for each combination
    for combination in combinations:
        a, b, c = combination

        # Evaluate the knowledge base (KB) and query expressions
        kb_result = evaluate_expression(a, b, c, kb)
        query_result = evaluate_expression(a, b, c, query)

        # Replace True/False with string "True"/"False"
        kb_result_str = "True" if kb_result else "False"
        query_result_str = "True" if query_result else "False"

        # Convert boolean values of a, b, c to "True"/"False"
        a_str = "True" if a else "False"
        b_str = "True" if b else "False"
        c_str = "True" if c else "False"

        # Print the results for the knowledge base and the query
        print(f"{a_str:<5} {b_str:<5} {c_str:<5} {kb_result_str:<20} {query_result_str:<20}")

    # Additional output for combinations where both KB and query are true
    print("\nCombinations where both KB and Query are True:")
    print(f"{'a':<5} {'b':<5} {'c':<5} {'KB':<20}{Query':<20}")

    # Print only the rows where both KB and Query are True
    for combination in combinations:
        a, b, c = combination

        # Evaluate the knowledge base (KB) and query expressions
        kb_result = evaluate_expression(a, b, c, kb)
        query_result = evaluate_expression(a, b, c, query)

        # If both KB and query are True, print the combination
        if kb_result and query_result:
            a_str = "True" if a else "False"
            b_str = "True" if b else "False"
            c_str = "True" if c else "False"
            kb_result_str = "True" if kb_result else "False"
            query_result_str = "True" if query_result else "False"
            print(f"{a_str:<5} {b_str:<5} {c_str:<5} {kb_result_str:<20} {query_result_str:<20}")

```

```
# Define the logical expressions as strings
kb = "(a or c) and (b or not c)" # Knowledge Base
query = "a or b" # Query to evaluate

# Generate the truth table and evaluate the knowledge base and query
truth_table_and_evaluation(kb, query)
```

a	b	c	KB	Query
False	False	False	False	False
False	False	True	False	False
False	True	False	False	True
False	True	True	True	True
True	False	False	True	True
True	False	True	False	True
True	True	False	True	True
True	True	True	True	True

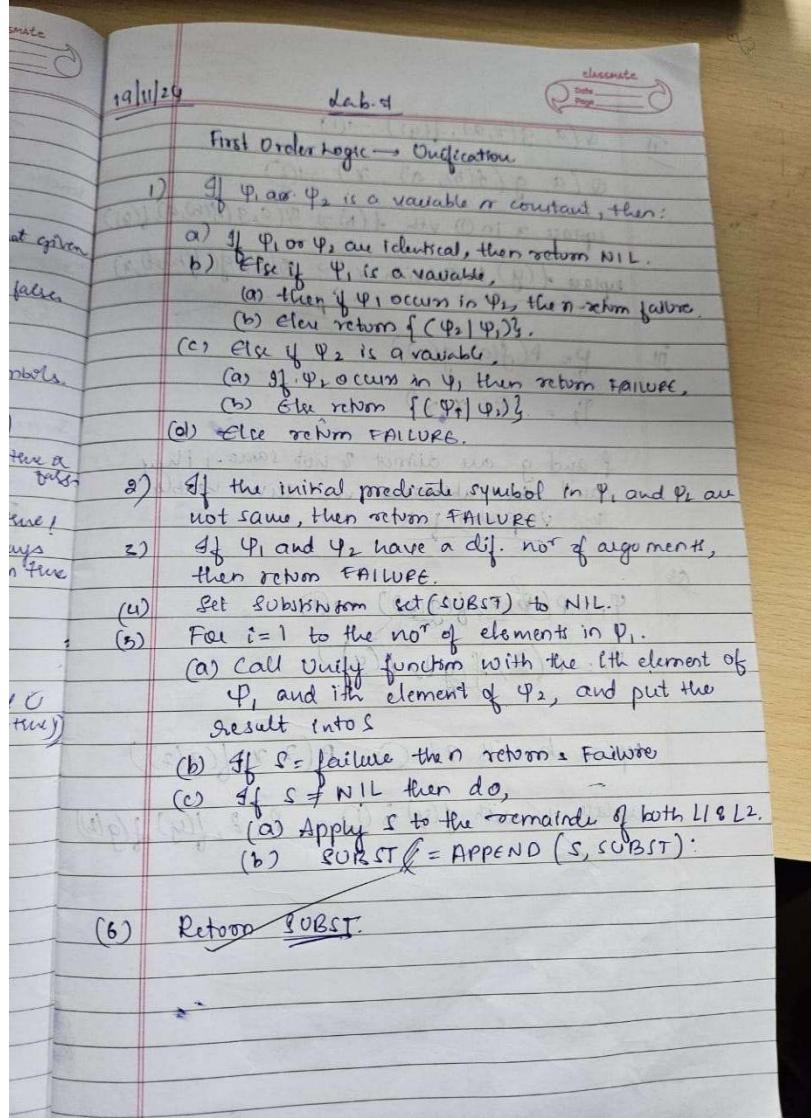
Combinations where both KB and Query are True:

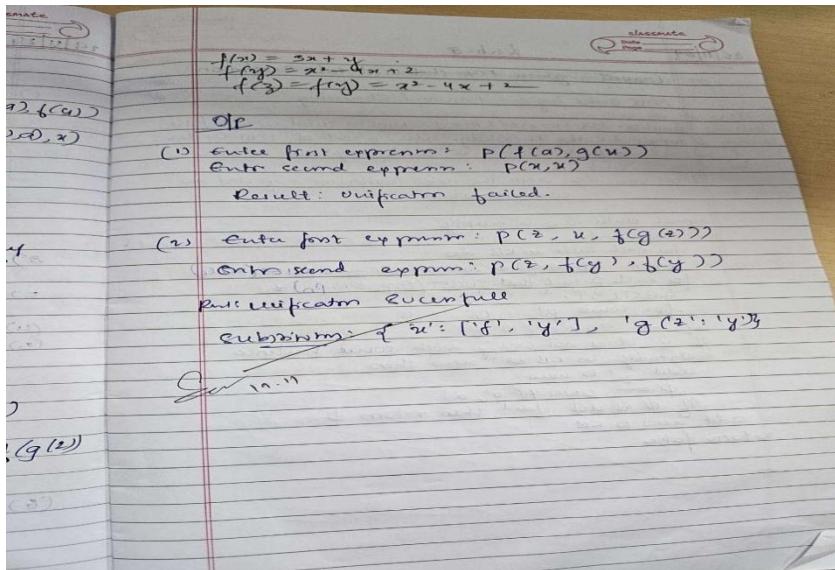
a	b	c	KB	Query
False	True	True	True	True
True	False	False	True	True
True	True	False	True	True
True	True	True	True	True

## Program 7

Implement unification in first order logic

Algorithm:





Code:

```
import re

def occurs_check(var, x):
    """Checks if var occurs in x (to prevent circular substitutions)."""
    if var == x:
        return True
    elif isinstance(x, list): # If x is a compound expression (like a function or predicate)
        return any(occurs_check(var, xi) for xi in x)
    return False
```

```
def unify_var(var, x, subst):
    """Handles unification of a variable with another term."""
    if var in subst: # If var is already substituted
        return unify(subst[var], x, subst)
    elif isinstance(x, (list, tuple)) and tuple(x) in subst: # Handle compound expressions
        return unify(var, subst[tuple(x)], subst)
    elif occurs_check(var, x): # Check for circular references
        return "FAILURE"
    else:
        # Add the substitution to the set (convert list to tuple for hashability)
        subst[var] = tuple(x) if isinstance(x, list) else x
    return subst
```

```
def unify(x, y, subst=None):
    """
    Unifies two expressions x and y and returns the substitution set if they can be unified.
    Returns 'FAILURE' if unification is not possible.
    """
    if subst is None:
        subst = {} # Initialize an empty substitution set

    # Step 1: Handle cases where x or y is a variable or constant
    if x == y: # If x and y are identical
        return subst
    elif isinstance(x, str) and x.islower(): # If x is a variable
```

```

        return unify_var(x, y, subst)
    elif isinstance(y, str) and y.islower(): # If y is a variable
        return unify_var(y, x, subst)
    elif isinstance(x, list) and isinstance(y, list): # If x and y are compound expressions (lists)
        if len(x) != len(y): # Step 3: Different number of arguments
            return "FAILURE"

    # Step 2: Check if the predicate symbols (the first element) match
    if x[0] != y[0]: # If the predicates/functions are different
        return "FAILURE"

    # Step 5: Recursively unify each argument
    for xi, yi in zip(x[1:], y[1:]): # Skip the predicate (first element)
        subst = unify(xi, yi, subst)
        if subst == "FAILURE":
            return "FAILURE"
    return subst
else: # If x and y are different constants or non-unifiable structures
    return "FAILURE"

```

```

def unify_and_check(expr1, expr2):
"""
Attempts to unify two expressions and returns a tuple:
(is_unified: bool, substitutions: dict or None)
"""
result = unify(expr1, expr2)
if result == "FAILURE":
    return False, None
return True, result

```

```

def display_result(expr1, expr2, is_unified, subst):
    print("Expression 1:", expr1)
    print("Expression 2:", expr2)
    if not is_unified:
        print("Result: Unification Failed")
    else:
        print("Result: Unification Successful")
        print("Substitutions:", {k: list(v) if isinstance(v, tuple) else v for k, v in subst.items()})

```

```

def parse_input(input_str):
    """Parses a string input into a structure that can be processed by the unification algorithm."""
    # Remove spaces and handle parentheses
    input_str = input_str.replace(" ", "")

    # Handle compound terms (like p(x, f(y)) -> ['p', 'x', ['f', 'y']])
    def parse_term(term):
        # Handle the compound term
        if '(' in term:
            match = re.match(r'([a-zA-Z0-9_]+)\((.*)\)', term)
            if match:
                predicate = match.group(1)
                arguments_str = match.group(2)
                arguments = [parse_term(arg.strip()) for arg in arguments_str.split(',')]

```

```

        return [predicate] + arguments
    return term

return parse_term(input_str)

# Main function to interact with the user
def main():
    while True:
        # Get the first and second terms from the user
        expr1_input = input("Enter the first expression (e.g., p(x, f(y))): ")
        expr2_input = input("Enter the second expression (e.g., p(a, f(z))): ")

        # Parse the input strings into the appropriate structures
        expr1 = parse_input(expr1_input)
        expr2 = parse_input(expr2_input)

        # Perform unification
        is_unified, result = unify_and_check(expr1, expr2)

        # Display the results
        display_result(expr1, expr2, is_unified, result)

        # Ask the user if they want to run another test
        another_test = input("Do you want to test another pair of expressions? (yes/no): ").strip().lower()
        if another_test != 'yes':
            break

if __name__ == "__main__":
    main()

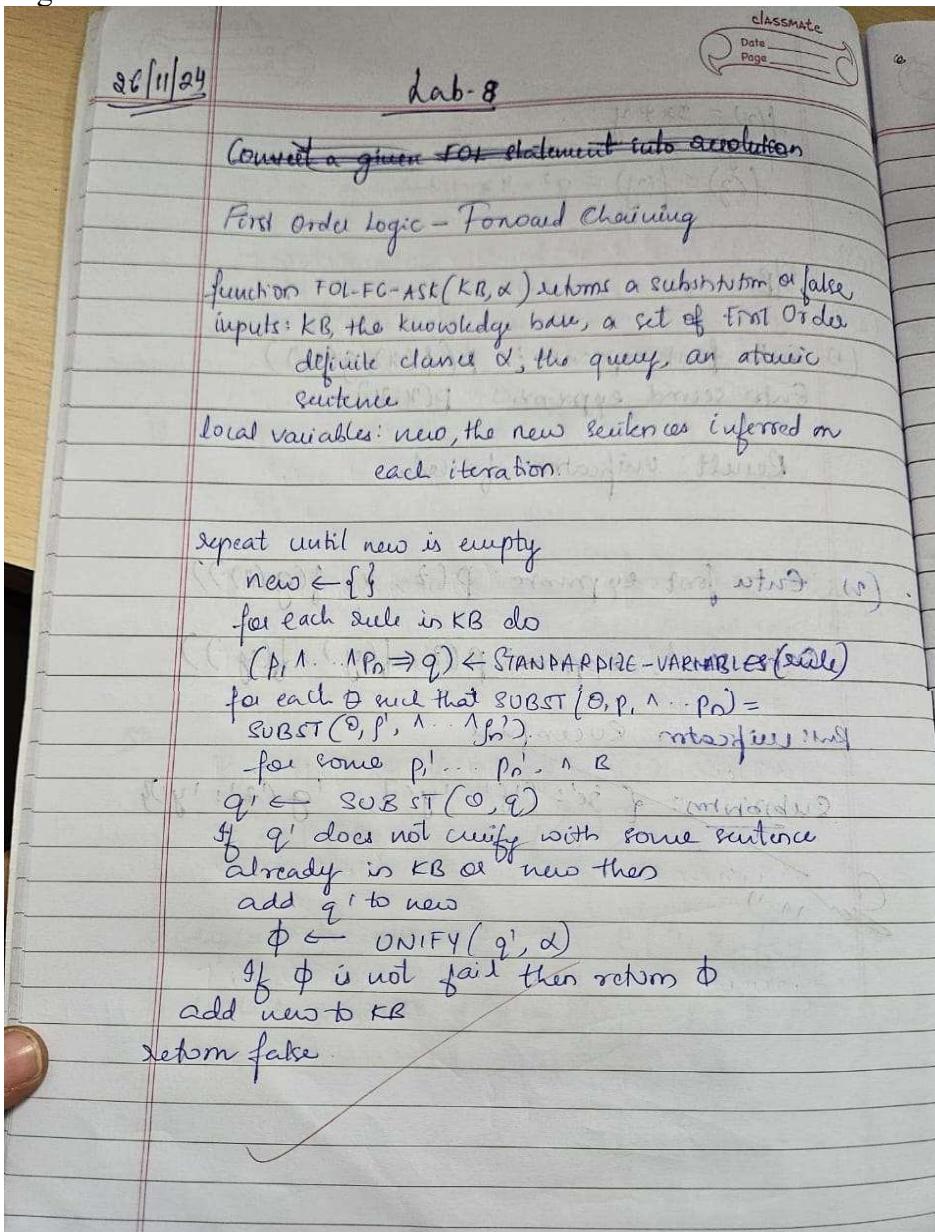
→ Enter the first expression (e.g., p(x, f(y))): q(a,g(x,a),f(y))
Enter the second expression (e.g., p(a, f(z))): q(a,g(f(b),a),x)
Expression 1: ['q', 'a', 'g(x', 'a)', ['f', 'y']]
Expression 2: ['q', 'a', ['g', 'f(b)', 'a'], 'x']
Result: Unification Successful
Substitutions: {'g(x)': ['g', 'f(b)', 'x': ['f', 'y']}]
Do you want to test another pair of expressions? (yes/no): yes
Enter the first expression (e.g., p(x, f(y))): p(z,x,f(g(z))
Enter the second expression (e.g., p(a, f(z))): p(z,f(y),f(y))
Expression 1: ['p', 'z', 'x', ['f', 'g(z']]]
Expression 2: ['p', 'z', ['f', 'y'], ['f', 'y']]
Result: Unification Successful
Substitutions: {'x': ['f', 'y'], 'g(z)': 'y'}
Do you want to test another pair of expressions? (yes/no): yes
Enter the first expression (e.g., p(x, f(y))): p(f(a),g(x))
Enter the second expression (e.g., p(a, f(z))): p(x,x)
Expression 1: ['p', ['f', 'a'], ['g', 'x']]
Expression 2: ['p', 'x', 'x']
Result: Unification Failed
Do you want to test another pair of expressions? (yes/no): no

```

## Program 8

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

Algorithm:



Code:

```
# Define the knowledge base (KB) as a set of facts
KB = set()
```

```
# Premises based on the provided FOL problem
KB.add('American(Robert)')
KB.add('Enemy(America, A)')
KB.add('Missile(T1)')
KB.add('Owns(A, T1)')
```

```
# Define inference rules
```

```

def modus_ponens(fact1, fact2, conclusion):
    """ Apply modus ponens inference rule: if fact1 and fact2 are true, then conclude conclusion """
    if fact1 in KB and fact2 in KB:
        KB.add(conclusion)
        print(f"Inferred: {conclusion}")

def forward_chaining():
    """ Perform forward chaining to infer new facts until no more inferences can be made """
    # 1. Apply: Missile(x) → Weapon(x)
    if 'Missile(T1)' in KB:
        KB.add('Weapon(T1)')
        print(f"Inferred: Weapon(T1)")

    # 2. Apply: Sells(Robert, T1, A) from Owns(A, T1) and Weapon(T1)
    if 'Owns(A, T1)' in KB and 'Weapon(T1)' in KB:
        KB.add('Sells(Robert, T1, A)')
        print(f"Inferred: Sells(Robert, T1, A)")

    # 3. Apply: Hostile(A) from Enemy(A, America)
    if 'Enemy(America, A)' in KB:
        KB.add('Hostile(A)')
        print(f"Inferred: Hostile(A)")

    # 4. Now, check if the goal is reached (i.e., if 'Criminal(Robert)' can be inferred)
    if 'American(Robert)' in KB and 'Weapon(T1)' in KB and 'Sells(Robert, T1, A)' in KB and 'Hostile(A)' in KB:
        KB.add('Criminal(Robert)')
        print("Inferred: Criminal(Robert)")

    # Check if we've reached our goal
    if 'Criminal(Robert)' in KB:
        print("Robert is a criminal!")
    else:
        print("No more inferences can be made.")

# Run forward chaining to attempt to derive the conclusion
forward_chaining()

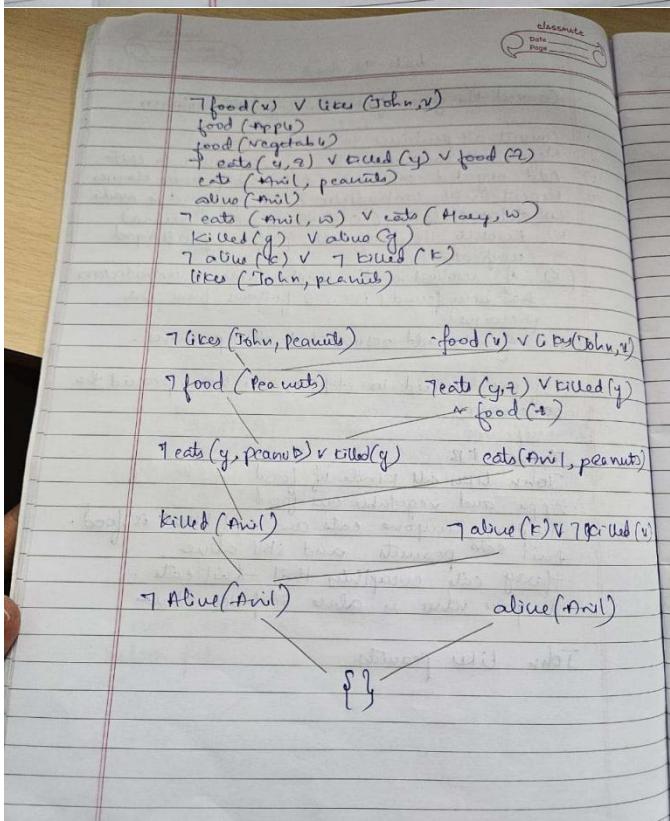
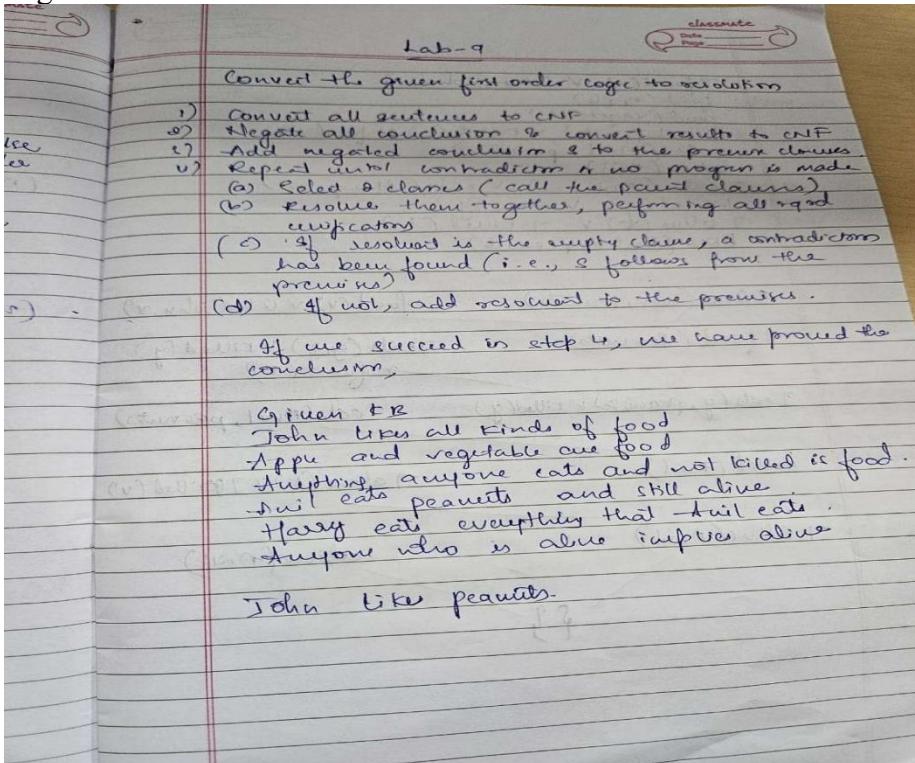
```

→ Inferred: Weapon(T1)  
 Inferred: Sells(Robert, T1, A)  
 Inferred: Hostile(A)  
 Inferred: Criminal(Robert)  
 Robert is a criminal!

## Program 9

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

## Algorithm:



Code:

```
# Define the knowledge base (KB)
KB = {
    "food(Apple)": True,
    "food(vegetables)": True,
    "eats(Anil, Peanuts)": True,
    "alive(Anil)": True,
    "likes(John, X)": "food(X)", # Rule: John likes all food
    "food(X)": "eats(Y, X) and not killed(Y)", # Rule: Anything eaten and not killed is food
    "eats(Harry, X)": "eats(Anil, X)", # Rule: Harry eats what Anil eats
    "alive(X)": "not killed(X)", # Rule: Alive implies not killed
    "not killed(X)": "alive(X)", # Rule: Not killed implies alive
}

# Function to evaluate if a predicate is true based on the KB
def resolve(predicate):
    # If it's a direct fact in KB
    if predicate in KB and isinstance(KB[predicate], bool):
        return KB[predicate]

    # If it's a derived rule
    if predicate in KB:
        rule = KB[predicate]
        if " and " in rule: # Handle conjunction
            sub_preds = rule.split(" and ")
            return all(resolve(sub.strip()) for sub in sub_preds)
        elif " or " in rule: # Handle disjunction
            sub_preds = rule.split(" or ")
            return any(resolve(sub.strip()) for sub in sub_preds)
        elif "not " in rule: # Handle negation
            sub_pred = rule[4:] # Remove "not "
            return not resolve(sub_pred.strip())
        else: # Handle single predicate
            return resolve(rule.strip())

    # If the predicate is a specific query (e.g., likes(John, Peanuts))
    if "(" in predicate:
        func, args = predicate.split("(")
        args = args.strip(")").split(", ")
        if func == "food" and args[0] == "Peanuts":
            return resolve("eats(Anil, Peanuts)") and not resolve("killed(Anil)")
        if func == "likes" and args[0] == "John" and args[1] == "Peanuts":
            return resolve("food(Peanuts)")

    # Default to False if no rule or fact applies
    return False

# Query to prove: John likes Peanuts
query = "likes(John, Peanuts)"
result = resolve(query)

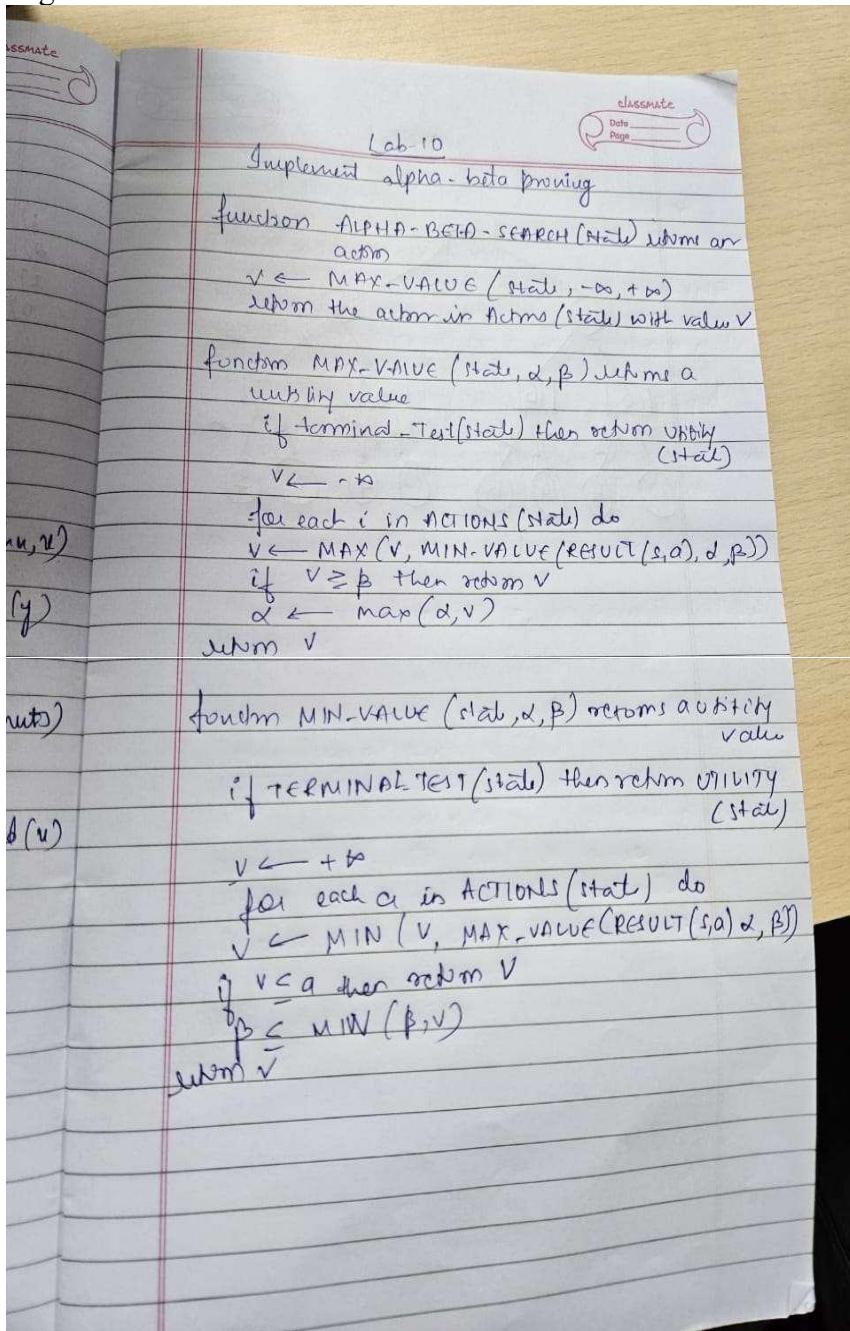
# Print the result
print(f"Does John like peanuts? {'Yes' if result else 'No'}")
```

→ Does John like peanuts? Yes

## Program 10

Implement Alpha-Beta Pruning

Algorithm:



Code:

```
# Alpha-Beta Pruning Implementation
def alpha_beta_pruning(node, alpha, beta, maximizing_player):
    # Base case: If it's a leaf node, return its value (simulating evaluation of the node)
    if type(node) is int:
        return node
```

```

# If not a leaf node, explore the children
if maximizing_player:
    max_eval = -float('inf')
    for child in node: # Iterate over children of the maximizer node
        eval = alpha_beta_pruning(child, alpha, beta, False)
        max_eval = max(max_eval, eval)
        alpha = max(alpha, eval) # Maximize alpha
        if beta <= alpha: # Prune the branch
            break
    return max_eval
else:
    min_eval = float('inf')
    for child in node: # Iterate over children of the minimizer node
        eval = alpha_beta_pruning(child, alpha, beta, True)
        min_eval = min(min_eval, eval)
        beta = min(beta, eval) # Minimize beta
        if beta <= alpha: # Prune the branch
            break
    return min_eval

# Function to build the tree from a list of numbers
def build_tree(numbers):
    # We need to build a tree with alternating levels of maximizers and minimizers
    # Start from the leaf nodes and work up
    current_level = [[n] for n in numbers]

    while len(current_level) > 1:
        next_level = []
        for i in range(0, len(current_level), 2):
            if i + 1 < len(current_level):
                next_level.append(current_level[i] + current_level[i + 1]) # Combine two nodes
            else:
                next_level.append(current_level[i]) # Odd number of elements, just carry forward
        current_level = next_level

    return current_level[0] # Return the root node, which is a maximizer

# Main function to run alpha-beta pruning
def main():
    # Input: User provides a list of numbers
    numbers = list(map(int, input("Enter numbers for the game tree (space-separated): ").split()))

    # Build the tree with the given numbers
    tree = build_tree(numbers)

    # Parameters: Tree, initial alpha, beta, and the root node is a maximizing player

```

```
alpha = -float('inf')
beta = float('inf')
maximizing_player = True # The root node is a maximizing player

# Perform alpha-beta pruning and get the final result
result = alpha_beta_pruning(tree, alpha, beta, maximizing_player)

print("Final Result of Alpha-Beta Pruning:", result)

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

```
Enter numbers for the game tree (space-separated): 10 9 14 18 5 4 50 3
Final Result of Alpha-Beta Pruning: 50
```