VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Satish Girish Kudare (1BM23CS306)

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

<u>CERTIFICATE</u>

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Satish Girish Kudare (1BM23CS306),** 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.

Mrs. Seema Patil Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

INDEX

NAME: Satish STD.: ______ SEC.: ______ ROLL NO.: ______ SUB.: AI (Lab+Theory)

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1a	18/8/25	Implement Tic-Tac-Toe		10
1b.	25/8/25	Implement Vaccum Cleaner Agent.		10
2a.	25/8/25	BFS - without Heuristic approach.		10
2b.	25/8/25	BFS - with Heuristic approach		
2c.	25/8/25	Iterative Deepening DFS		
3.		Apply A* for 8 puzzle		10
3a.	8/9/25	Using Misplaced Tiles		
3b	8/9/25	Using Manhatten Dist.		
4	15/9/25	Hill climbing for 4 queens.	10	
5.	15/9/25	Simulated Annealing for 8 queens.		
6	22/9/25	Propositional logic	10	
7	13/10/25	Unification Algo	8	
8..	13/10/25	First Order logic - Forward Chaining		
9	27/10/25	FOL - Resolution.	10	
10.	27/10/25	Alpha-Beta pruning		

Github Link: https://github.com/Satish1895/1BM23CS306_AI.git

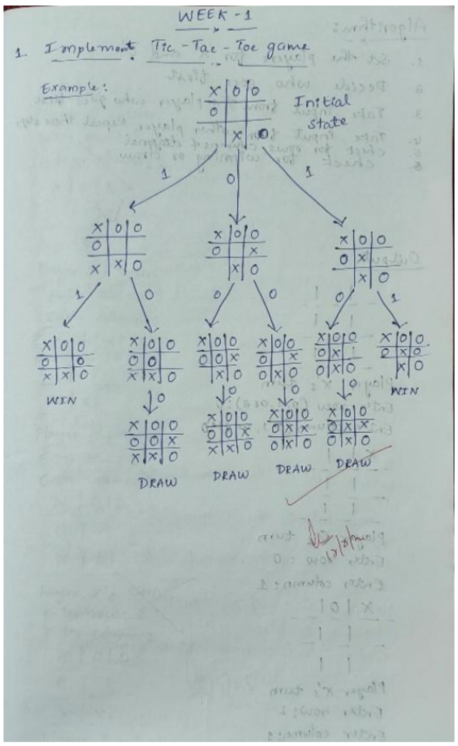
Program 1

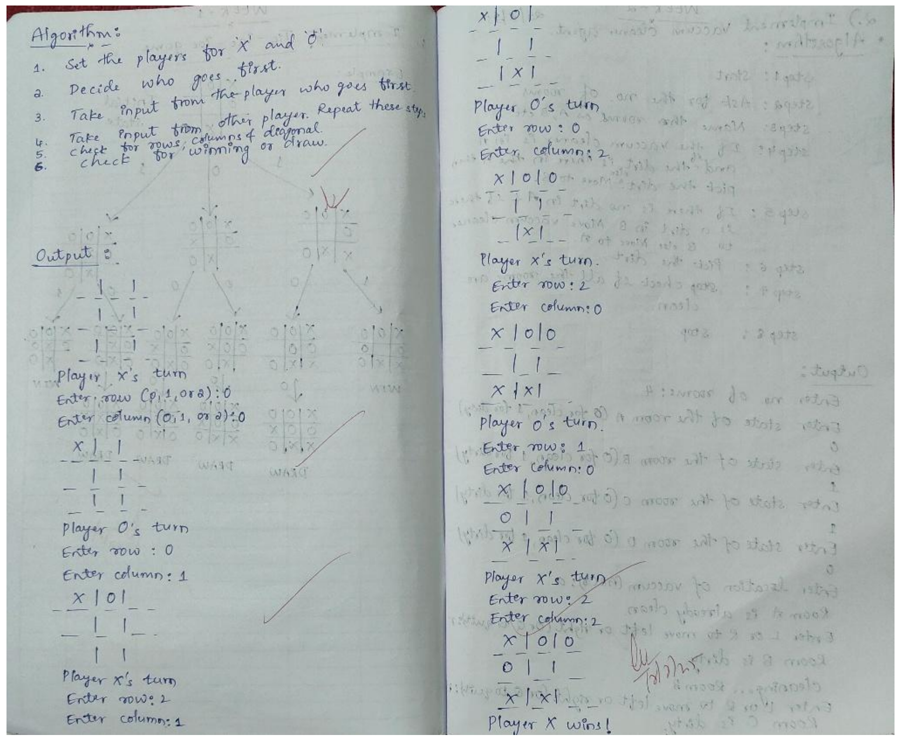
Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Algorithm:

a) Tic Tac Toe




Algorithm :
1. Set the players for 'X' and 'O'.
2. Decide who goes first.
3. Take input from the player who goes first
4. Take input from other player. Repeat these steps
5. Check for rows, columns & diagonal.
6. check for winning or draw.

Output :

Player X's turn
Enter row (0,1,or 2): 0
Enter column (0,1, or 2): 0

Player O's turn
Enter row : 0
Enter column: 1

Player X's turn
Enter row: 2
Enter column: 1

Player O's turn
Enter row : 0
Enter column: 2

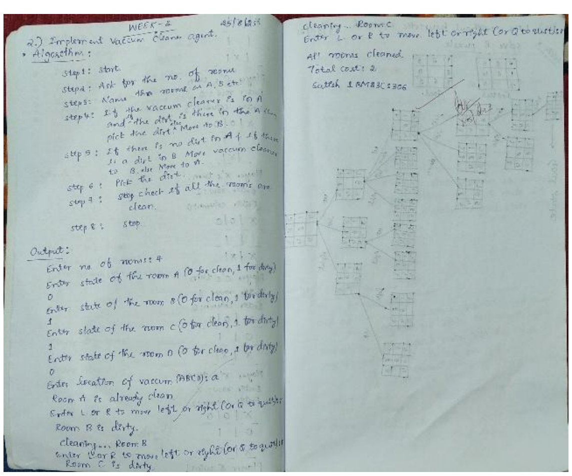
Player X's turn
Enter row : 2
Enter column: 0

Player O's turn
Enter row: 1
Enter column: 0

Player X's turn
Enter row: 2
Enter column: 2

Player X wins!

b) Vacuum Cleaner


WEEK - 2

2.) Implement Vacuum Cleaner agent.

Algorithm :

step 1: start

step 2: Ask for the no. of rooms

step 3: Name the rooms as A, B etc.

step 4: If the vaccum cleaner is in A and the dirt is there in the A then pick the dirt else Move to B.

step 5: If there is no dirt in A & if there is a dirt in B Move vaccum cleaner to B else Move to A.

step 6: Pick the dirt.

step 7: Stop check if all the rooms are clean.

step 8: stop.

Output:

Enter no of rooms: 4

Enter state of the room A (0 for clean, 1 for dirty)

0

Enter state of the room B (0 for clean, 1 for dirty)

1

Enter state of the room C (0 for clean, 1 for dirty)

1

Enter state of the room D (0 for clean, 1 for dirty)

0

Enter location of vaccum (ABCD): a

Room A is already clean

Enter L or R to move left or right (or Q to quit):

Room B is dirty.

cleaning... Room B

Enter L or R to move left or right (or Q to quit):

Room C is dirty.

cleaning... Room C

Enter L or R to move left or right (or Q to quit):

All rooms cleaned.

Total cost: 2

Satish 1BM23CS306

Code:
a)Tic Tac Toe

```
def print_board(board):
    """Prints the Tic Tac Toe board."""
    for row in board:
        print(" | ".join(row))
        print("-" * 9)

def check_win(board, player):
    """Checks if the current player has won."""
    n = len(board)
    # Check rows
    for row in board:
        if all(cell == player for cell in row):
            return True
    # Check columns
    for col in range(n):
        if all(board[row][col] == player for row in range(n)):
            return True
    # Check diagonals
    if all(board[i][i] == player for i in range(n)) or \
       all(board[i][n - 1 - i] == player for i in range(n)):
        return True
    return False

def check_draw(board):
    """Checks if the game is a draw."""
    for row in board:
        if " " in row:
            return False
    return True

def tic_tac_toe():
    """Runs the Tic Tac Toe game."""
    board = [[" " for _ in range(3)] for _ in range(3)]
    players = ["X", "O"]
    current_player = 0

    while True:
        print_board(board)
        player = players[current_player]
```

```
        print(f"Player {player}'s turn.")

        while True:
            try:
                row = int(input("Enter row (0, 1, or 2): "))
                col = int(input("Enter column (0, 1, or 2): "))
                if 0 <= row < 3 and 0 <= col < 3 and board[row][col] == " ":
                    board[row][col] = player
                    break
                else:
                    print("Invalid move. Try again.")
            except ValueError:
                print("Invalid input. Please enter numbers.")

        if check_win(board, player):
            print_board(board)
            print(f"Player {player} wins!")
            break
        elif check_draw(board):
            print_board(board)1

            print("It's a draw!")
            break

        current_player = (current_player + 1) % 2

if __name__ == "__main__":
    tic_tac_toe()

print("Satish 1BM23CS306")
```

Output:

```
  |  |
---------
 |  |
---------
 |  |
---------
Player X's turn.
Enter row (0, 1, or 2): 0
Enter column (0, 1, or 2): 0
```

```
X |   |
---------
  |   |
---------
  |   |
---------
Player O's turn.
Enter row (0, 1, or 2): 0
Enter column (0, 1, or 2): 2
X |   | O
---------
  |   |
---------
  |   |
---------
Player X's turn.
Enter row (0, 1, or 2): 1
Enter column (0, 1, or 2): 1
X |   | O
---------
  | X |
---------
  |   |
---------
Player O's turn.
Enter row (0, 1, or 2): 2
Enter column (0, 1, or 2): 2
X |   | O
---------
  | X |
---------
  |   | O
---------
Player X's turn.
Enter row (0, 1, or 2): 1
Enter column (0, 1, or 2): 2
X |   | O
---------
  | X | X
---------
  |   | O
```

```
---------
Player O's turn.
Enter row (0, 1, or 2): 1
Enter column (0, 1, or 2): 0
X |   | O
---------
O | X | X
---------
  |   | O
---------
Player X's turn.
Enter row (0, 1, or 2): 0
Enter column (0, 1, or 2): 1
X | X | O
---------
O | X | X
---------
  |   | O
---------
Player O's turn.
Enter row (0, 1, or 2): 2
Enter column (0, 1, or 2): 1
X | X | O
---------
O | X | X
---------
  | O | O
---------
Player X's turn.
Enter row (0, 1, or 2): 2
Enter column (0, 1, or 2): 0
X | X | O
---------
O | X | X
---------
X | O | O
---------
It's a draw!
Satish 1BM23CS306
```

b) Vacuum Cleaner

```
rooms = int(input("Enter no. of rooms: "))
Rooms = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
RoomState = {}
cost = 0
for i in range(rooms):
  print(f"Enter state of the room {Rooms[i]} (0 for clean,1 for dirty)")
  state = int(input())
  RoomState[Rooms[i]] = state
loc = input(f"Enter Location of vacuum ({Rooms[:rooms]}): ").upper()
while 1 in RoomState.values():
    if RoomState[loc] == 1:
        print(f"Room {loc} is dirty.")
        print(f"Cleaning...Room {loc}")
        RoomState[loc] = 0
        cost += 1
    else:
        print(f"Room {loc} is already clean.")

    move = input("Enter L or R to move left or right (or Q to quit): ").upper()

    if move == "L":
        if loc != Rooms[0]:
            loc = Rooms[Rooms.index(loc) - 1]
        else:
            print("No room to move left.")
    elif move == "R":
        if loc != Rooms[rooms - 1]:
            loc = Rooms[Rooms.index(loc) + 1]
        else:
            print("No room to move right.")
    elif move == "Q":
        break
    else:
        print("Invalid input. Please enter L, R, or Q.")

print("\nAll Rooms Cleaned." if 1 not in RoomState.values() else "Exited before cleaning all
rooms.")
print(f"Total cost: {cost}")
print("Satish 1BM23CS306")
```

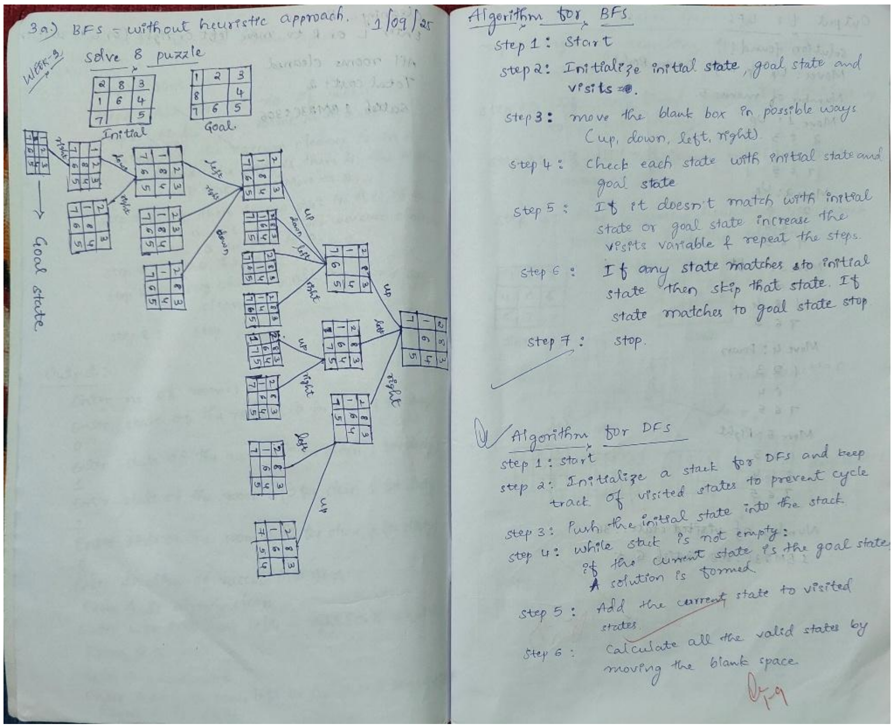
Output:
Enter Number of rooms: 3
Enter Room A state (0 for clean, 1 for dirty): 1
Enter Room B state (0 for clean, 1 for dirty): 0
Enter Room C state (0 for clean, 1 for dirty):
1
Enter Location of vacuum (ABC): B Room
B is already clean.
Enter L or R to move left or right (or Q to quit): L Room
A is dirty. Cleaning...
Enter L or R to move left or right (or Q to quit): R Room
B is already clean.
Enter L or R to move left or right (or Q to quit): R Room
C is dirty. Cleaning...
Enter L or R to move left or right (or Q to quit): R No
room to move right.

All Rooms Cleaned.
Total cost: 2
Satish 1BM23CS306

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)

Implement Iterative deepening search algorithm Algorithm:



3a) BFS - without heuristic approach. 1/09/25

Week-3 solve 8 puzzle

Initial Goal

Goal state

Algorithm for BFS

step 1: Start

step 2: Initialize initial state, goal state and visits = 0.

step 3: move the blank box in possible ways (up, down, left, right)

step 4: Check each state with initial state and goal state

step 5: If it doesn't match with initial state or goal state increase the visits variable & repeat the steps.

step 6: If any state matches to initial state then skip that state. If state matches to goal state stop

step 7: stop.

Algorithm for DFS

step 1: start

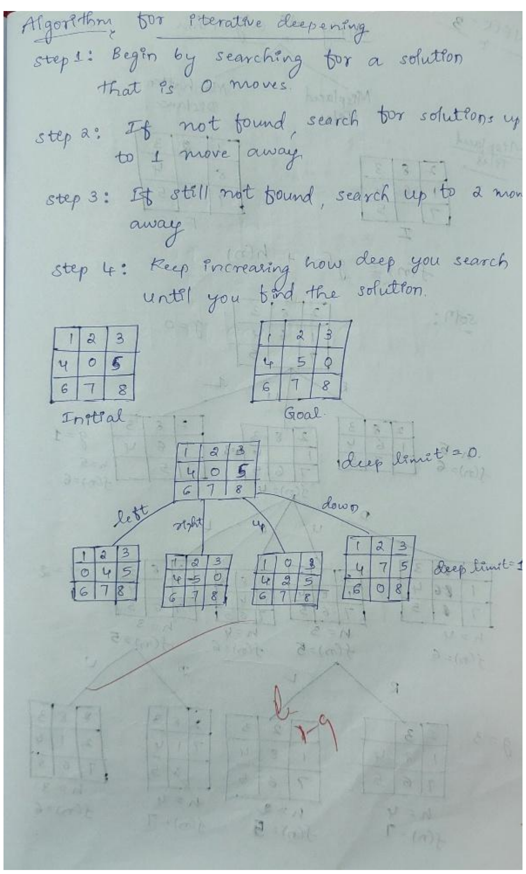
step 2: Initialize a stack for DFS and keep track of visited states to prevent cycle

step 3: Push the initial state into the stack.

step 4: while stack is not empty: if the current state is the goal state A solution is formed

step 5: Add the current state to visited states

step 6: Calculate all the valid states by moving the blank space.



Algorithm for iterative deepening

step 1: Begin by searching for a solution that is 0 moves.

step 2: If not found, search for solutions up to 1 move away.

step 3: If still not found, search up to 2 moves away

step 4: Keep increasing how deep you search until you find the solution.

1	2	3
4	0	5
6	7	8

Initial

1	2	3
4	5	0
6	7	8

Goal

Code:

a) DFS

```
goal_state = '123804765'

moves = {
    'U': -3,
    'D': 3,
    'L': -1,
    'R': 1
}

invalid_moves = {
    0: ['U', 'L'], 1: ['U'], 2: ['U', 'R'],
    3: ['L'],        5: ['R'],
    6: ['D', 'L'], 7: ['D'], 8: ['D', 'R']
}

def move_tile(state, direction):
    index = state.index('0')
    if direction in invalid_moves.get(index, []):
        return None

    new_index = index + moves[direction]
    if new_index < 0 or new_index >= 9:
        return None

    state_list = list(state)
    state_list[index], state_list[new_index] = state_list[new_index], state_list[index]
    return ''.join(state_list)

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

def dfs(start_state, max_depth=50):
    visited = set()
    stack = [(start_state, [])]  # Each element: (state, path)

    while stack:
        current_state, path = stack.pop()
```

```
        if current_state in visited:
            continue

        # Print every visited state
        print("Visited state:")
        print_state(current_state)

        if current_state == goal_state:
            return path

        visited.add(current_state)

        if len(path) >= max_depth:
            continue

        for direction in moves:
            new_state = move_tile(current_state, direction)
            if new_state and new_state not in visited:
                stack.append((new_state, path + [direction]))

    return None


start = input("Enter the INITIAL state (give '0'for empty space): ")

if len(start) == 9 and set(start) == set('012345678'):
    print("INITIAL state:")
    print_state(start)

    result = dfs(start)

    if result is not None:
        print("Solution found!")
        print("Moves:", ' '.join(result))
        print("Number of moves:", len(result))

        current_state = start
        for i, move in enumerate(result, 1):
            current_state = move_tile(current_state, move)
            print(f"Move {i}: {move}")
```

```
            print_state(current_state)
    else:
        print("No solution exists for the given start state or max depth reached.")
else:
    print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")

print("1BM23CS306 Satish")
```

Output:

```
Enter start state (e.g., 724506831): 123456078
Start state:
1 2 3
4 5 6
  7 8

Visited state:
1 2 3
4 5 6
  7 8

Visited state:
1 2 3
4 5 6
7   8

Visited state:
1 2 3
4 5 6
7 8

Solution found!
Moves: R R
Number of moves: 2
Move 1: R
1 2 3
4 5 6
7   8
Move 2: R
1 2 3
4 5 6
7 8
1BM23CS306  Satish
```

b) Iterative Deepening Search

```
goal_state = '123456780'

moves = {
    'U': -3,
    'D': 3,
    'L': -1,
    'R': 1
}

invalid_moves = {
    0: ['U', 'L'], 1: ['U'], 2: ['U', 'R'],
    3: ['L'],        5: ['R'],
    6: ['D', 'L'], 7: ['D'], 8: ['D', 'R']
}

def move_tile(state, direction):
    index = state.index('0')
    if direction in invalid_moves.get(index, []):
        return None

    new_index = index + moves[direction]
    if new_index < 0 or new_index >= 9:
        return None

    state_list = list(state)
    state_list[index], state_list[new_index] = state_list[new_index], state_list[index]
    return ''.join(state_list)

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

def dls(state, depth, path, visited, visited_count):
    visited_count[0] += 1  # Increment visited states count
    if state == goal_state:
        return path
```

```
    if depth == 0:
        return None

    visited.add(state)

    for direction in moves:
        new_state = move_tile(state, direction)
        if new_state and new_state not in visited:
            result = dls(new_state, depth - 1, path + [direction], visited, visited_count)
            if result is not None:
                return result

    visited.remove(state)
    return None

def iddfs(start_state, max_depth=50):
    visited_count = [0]  # Using list to pass by reference
    for depth in range(max_depth + 1):
        visited = set()
        result = dls(start_state, depth, [], visited, visited_count)
        if result is not None:
            return result, visited_count[0]
    return None, visited_count[0]

# Main
start = input("Enter start state (e.g., 724506831): ")

if len(start) == 9 and set(start) == set('012345678'):
    print("Start state:")
    print_state(start)

    result, visited_states = iddfs(start,15)

    print(f"Total states visited: {visited_states}")

    if result is not None:
        print("Solution found!")
        print("Moves:", ' '.join(result))
        print("Number of moves:", len(result))
        print("1BM23CS306 Satish G K\n")
```

```
        current_state = start
        for i, move in enumerate(result, 1):
            current_state = move_tile(current_state, move)
            print(f"Move {i}: {move}")
            print_state(current_state)
    else:
        print("No solution exists for the given start state or max depth reached.")
else:
    print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")
```

Output:

```
Enter start state (e.g., 724506831): 123450678
Start state:
1 2 3
4 5
6 7 8

Total states visited: 9504
Solution found!
Moves: D L L U R D R U L L D R R
Number of moves: 13
1BM23CS306 Satish G K

Move 1: D
1 2 3
4 5 8
6 7

Move 2: L
1 2 3
4 5 8
6   7

Move 3: L
1 2 3
4 5 8
  6 7

Move 4: U
1 2 3
  5 8
4 6 7
```

Move 5: R
1 2 3
5   8
4 6 7

Move 6: D
1 2 3
5 6 8
4   7

Move 7: R
1 2 3
5 6 8
4 7

Move 8: U
1 2 3
5 6
4 7 8

Move 9: L
1 2 3
5   6
4 7 8

Move 10: L
1 2 3
  5 6
4 7 8

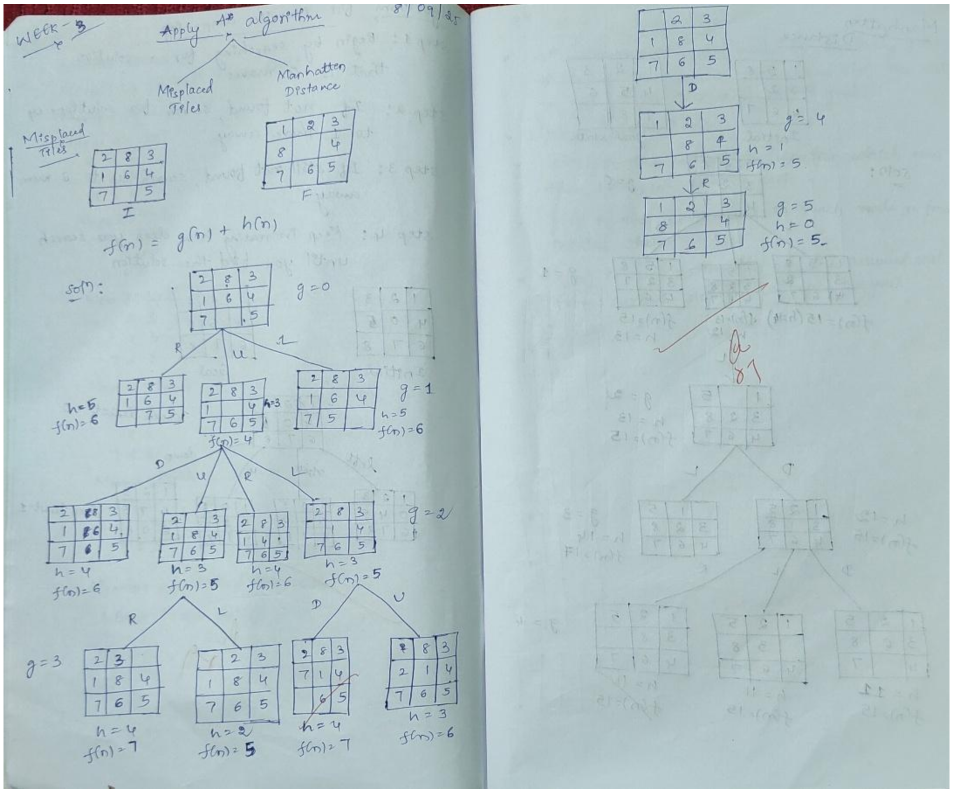
Move 11: D
1 2 3
4 5 6
  7 8

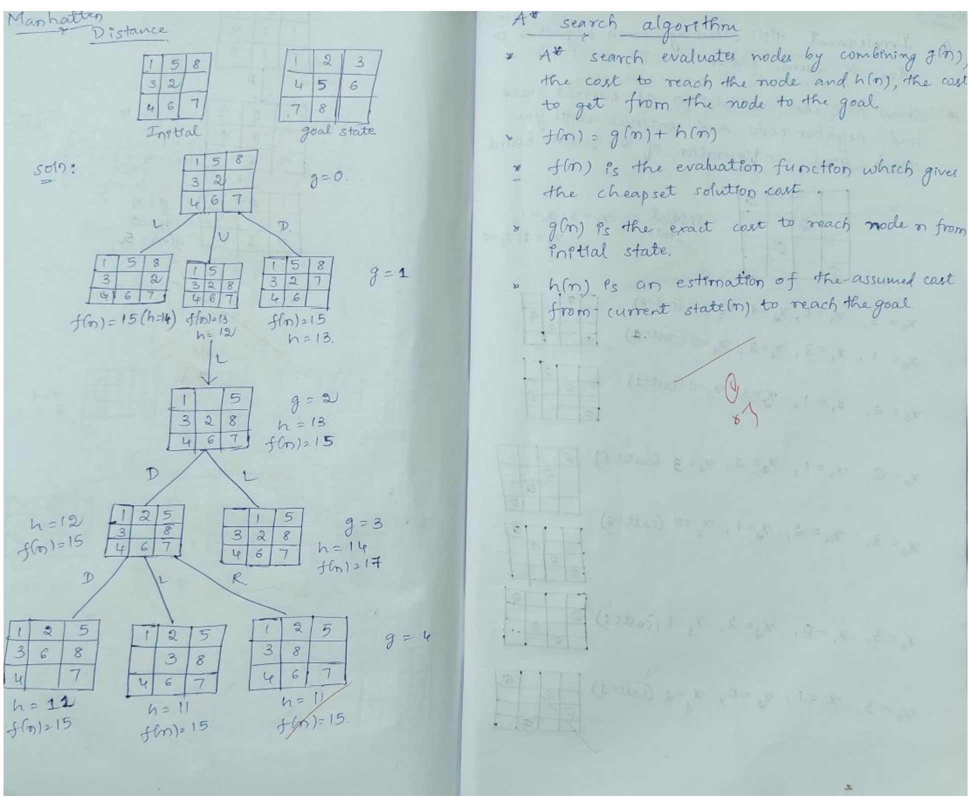
Move 12: R
1 2 3
4 5 6
7   8

Move 13: R
1 2 3
4 5 6
7 8

Program 3

Implement A* search algorithm





Code:

a) Misplaced Tiles

```
import heapq

goal_state = '123456780'

moves = {
    'U': -3,
    'D': 3,
    'L': -1,
    'R': 1
}

invalid_moves = {
    0: ['U', 'L'], 1: ['U'], 2: ['U', 'R'],
    3: ['L'],         5: ['R'],
    6: ['D', 'L'], 7: ['D'], 8: ['D', 'R']
}

def move_tile(state, direction):
    index = state.index('0')
    if direction in invalid_moves.get(index, []):
        return None

    new_index = index + moves[direction]
    if new_index < 0 or new_index >= 9:
        return None

    state_list = list(state)
    state_list[index], state_list[new_index] = state_list[new_index],
state_list[index]
    return ''.join(state_list)

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

def manhattan_distance(state):
    distance = 0
    for i, val in enumerate(state):
        if val == '0':
            continue
        goal_pos = int(val) - 1
        current_row, current_col = divmod(i, 3)
```

```
        goal_row, goal_col = divmod(goal_pos, 3)
        distance += abs(current_row - goal_row) + abs(current_col -
goal_col)
    return distance

def a_star(start_state):
    visited_count = 0
    open_set = []
    heapq.heappush(open_set, (manhattan_distance(start_state), 0,
start_state, []))
    visited = set()

    while open_set:
        f, g, current_state, path = heapq.heappop(open_set)
        visited_count += 1

        if current_state == goal_state:
            return path, visited_count

        if current_state in visited:
            continue
        visited.add(current_state)

        for direction in moves:
            new_state = move_tile(current_state, direction)
            if new_state and new_state not in visited:
                new_g = g + 1
                new_f = new_g + manhattan_distance(new_state)
                heapq.heappush(open_set, (new_f, new_g, new_state, path +
[direction]))

    return None, visited_count

# Main
start = input("Enter start state (e.g., 724506831): ")

if len(start) == 9 and set(start) == set('012345678'):
    print("Start state:")
    print_state(start)

    result, visited_states = a_star(start)

    print(f"Total states visited: {visited_states}")

    if result is not None:
        print("Solution found!")
```

```
        print("Moves:", ' '.join(result))
        print("Number of moves:", len(result))
        print("1BM23CS306 Satish G K\n")

        current_state = start
        g = 0  # initialize cost so far
        for i, move in enumerate(result, 1):
            new_state = move_tile(current_state, move)
            g += 1
            h = manhattan_distance(new_state)
            f = g + h
            print(f"Move {i}: {move}")
            print_state(new_state)
            print(f"g(n) = {g}, h(n) = {h}, f(n) = g(n) + h(n) = {f}\n")
            current_state = new_state
    else:
        print("No solution exists for the given start state.")
else:
    print("Invalid input! Please enter a 9-digit string using digits 0-8 
without repetition.")
```

Output:

Enter start state (e.g., 724506831): 283164705

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

1BM23CS306 Satish G K

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
8   4
7 6 5

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

b) Manhattan Distance

```
import heapq

goal_state = '123456780'

moves = {
    'U': -3,
    'D': 3,
    'L': -1,
    'R': 1
}

invalid_moves = {
    0: ['U', 'L'], 1: ['U'], 2: ['U', 'R'],
    3: ['L'],        5: ['R'],
    6: ['D', 'L'], 7: ['D'], 8: ['D', 'R']
}
```

```
def move_tile(state, direction):
    index = state.index('0')
    if direction in invalid_moves.get(index, []):
        return None

    new_index = index + moves[direction]
    if new_index < 0 or new_index >= 9:
        return None

    state_list = list(state)
    state_list[index], state_list[new_index] = state_list[new_index], state_list[index]
    return ''.join(state_list)

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

def manhattan_distance(state):
    distance = 0
    for i, val in enumerate(state):
        if val == '0':
            continue
        goal_pos = int(val) - 1
        current_row, current_col = divmod(i, 3)
        goal_row, goal_col = divmod(goal_pos, 3)
        distance += abs(current_row - goal_row) + abs(current_col - goal_col)
    return distance

def a_star(start_state):
    visited_count = 0
    open_set = []
    heapq.heappush(open_set, (manhattan_distance(start_state), 0, start_state, []))
    visited = set()

    while open_set:
        f, g, current_state, path = heapq.heappop(open_set)
        visited_count += 1

        if current_state == goal_state:
            return path, visited_count

        if current_state in visited:
            continue
```

```
        visited.add(current_state)

        for direction in moves:
            new_state = move_tile(current_state, direction)
            if new_state and new_state not in visited:
                new_g = g + 1
                new_f = new_g + manhattan_distance(new_state)
                heapq.heappush(open_set, (new_f, new_g, new_state, path + [direction]))

    return None, visited_count

# Main
start = input("Enter start state (e.g., 724506831): ")

if len(start) == 9 and set(start) == set('012345678'):
    print("Start state:")
    print_state(start)

    result, visited_states = a_star(start)

    print(f"Total states visited: {visited_states}")

    if result is not None:
        print("Solution found!")
        print("Moves:", ' '.join(result))
        print("Number of moves:", len(result))
        print("1BM23CS306 Satish G K\n")

        current_state = start
        g = 0  # initialize cost so far
        for i, move in enumerate(result, 1):
            new_state = move_tile(current_state, move)
            g += 1
            h = manhattan_distance(new_state)
            f = g + h
            print(f"Move {i}: {move}")
            print_state(new_state)
            print(f"g(n) = {g}, h(n) = {h}, f(n) = g(n) + h(n) = {f}\n")
            current_state = new_state
    else:
        print("No solution exists for the given start state.")
else:
    print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")
```

Output:

Enter start state (e.g., 724506831): 123678405
Start state:
1 2 3
6 7 8
4   5

Total states visited: 14
Solution found!
Moves: U L D R R U L D R
Number of moves: 9
1BM23CS306 Satish G K

Move 1: U
1 2 3
6   8
4 7 5

g(n) = 1, h(n) = 8, f(n) = g(n) + h(n) = 9

Move 2: L
1 2 3
  6 8
4 7 5

g(n) = 2, h(n) = 7, f(n) = g(n) + h(n) = 9

Move 3: D
1 2 3
4 6 8
  7 5

g(n) = 3, h(n) = 6, f(n) = g(n) + h(n) = 9

Move 4: R
1 2 3
4 6 8
7   5

g(n) = 4, h(n) = 5, f(n) = g(n) + h(n) = 9

Move 5: R
1 2 3
4 6 8
7 5

g(n) = 5, h(n) = 4, f(n) = g(n) + h(n) = 9

Move 6: U
1 2 3
4 6
7 5 8

g(n) = 6, h(n) = 3, f(n) = g(n) + h(n) = 9

Move 7: L
1 2 3
4   6
7 5 8

g(n) = 7, h(n) = 2, f(n) = g(n) + h(n) = 9

Move 8: D
1 2 3
4 5 6
7   8

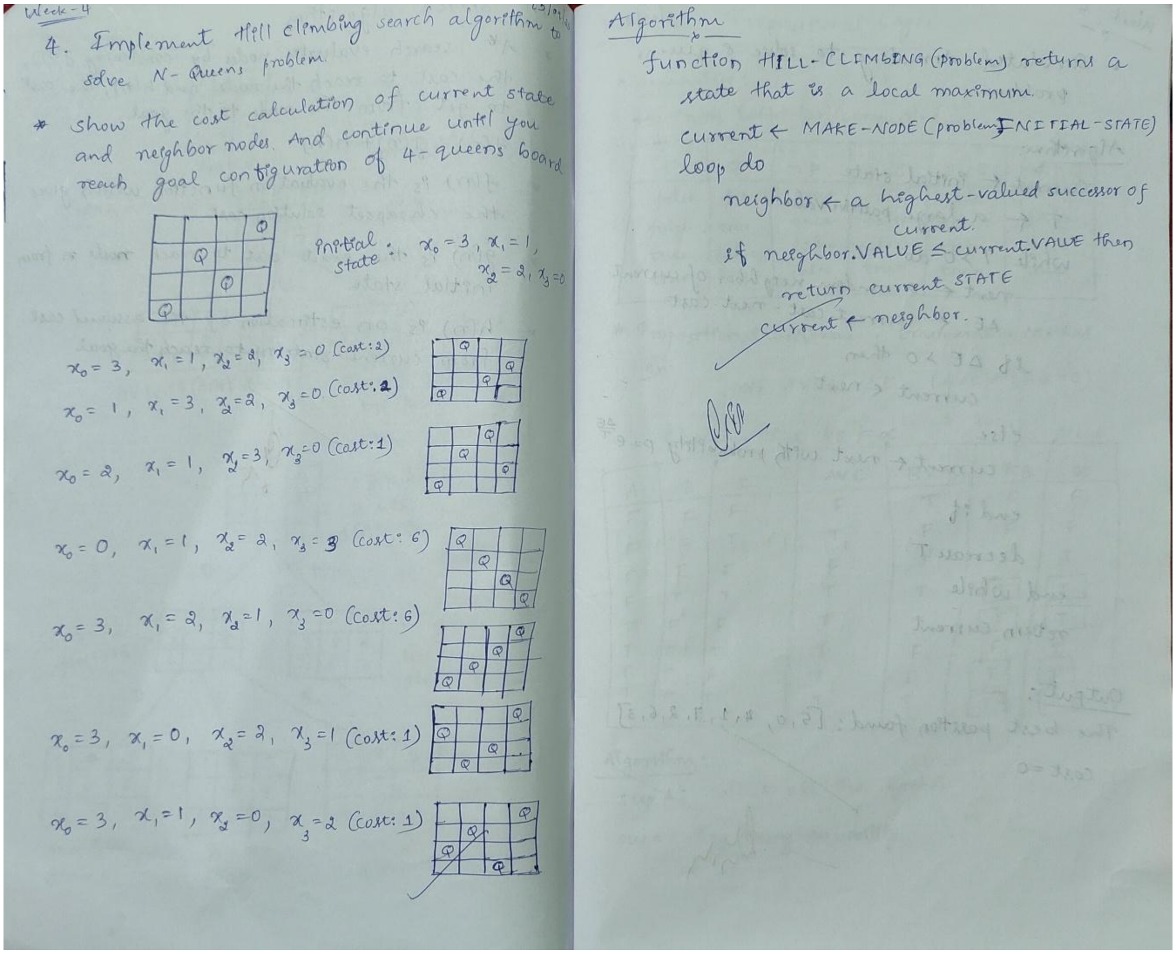
g(n) = 8, h(n) = 1, f(n) = g(n) + h(n) = 9

Move 9: R
1 2 3
4 5 6
7 8

g(n) = 9, h(n) = 0, f(n) = g(n) + h(n) = 9

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

Algorithm:


Week-4

4. Implement Hill climbing search algorithm to solve N-Queens problem.

* Show the cost calculation of current state and neighbor nodes. And continue until you reach goal configuration of 4-queens board

Initial state : $x_0 = 3$, $x_1 = 1$, $x_2 = 2$, $x_3 = 0$

$x_0 = 3$, $x_1 = 1$, $x_2 = 2$, $x_3 = 0$ (cost: 2)

$x_0 = 1$, $x_1 = 3$, $x_2 = 2$, $x_3 = 0$ (cost: 2)

$x_0 = 2$, $x_1 = 1$, $x_2 = 3$, $x_3 = 0$ (cost: 1)

$x_0 = 0$, $x_1 = 1$, $x_2 = 2$, $x_3 = 3$ (cost: 6)

$x_0 = 3$, $x_1 = 2$, $x_2 = 1$, $x_3 = 0$ (cost: 6)

$x_0 = 3$, $x_1 = 0$, $x_2 = 2$, $x_3 = 1$ (cost: 1)

$x_0 = 3$, $x_1 = 1$, $x_2 = 0$, $x_3 = 2$ (cost: 1)

Algorithm

function HILL-CLIMBING(problem) returns a state that is a local maximum

current ← MAKE-NODE(problem.INITIAL-STATE)
loop do
neighbor ← a highest-valued successor of current
if neighbor.VALUE ≤ current.VALUE then
return current.STATE
current ← neighbor.

Code:

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

# Generate neighbors (move one queen in a column to a different row)
def generate_neighbors(state):
    neighbors = []
```

```
    n = len(state)
    for col in range(n):
        for row in range(n):
            if state[col] != row:  # move queen
                new_state = list(state)
                new_state[col] = row
                neighbors.append(new_state)
    return neighbors

def hill_climbing(initial_state):
    current = initial_state
    current_cost = calculate_cost(current)
    step = 0

    print(f"Step {step}: State = {current}, Cost = {current_cost}")

    while True:
        neighbors = generate_neighbors(current)
        neighbor_costs = [(n, calculate_cost(n)) for n in neighbors]

        # Print state space for this step
        print("\nNeighbors and their costs:")
        for n, c in neighbor_costs:
            print(f"    {n} -> Cost = {c}")

        # Pick the best neighbor (lowest cost)
        best_neighbor, best_cost = min(neighbor_costs, key=lambda x: x[1])

        if best_cost >= current_cost:
            print("\nNo better neighbor found. Algorithm stops.")
            break

        # Move to better state
        step += 1
        current, current_cost = best_neighbor, best_cost
        print(f"\nStep {step}: Move to {current}, Cost = {current_cost}")

        if current_cost == 0:
            print("\nGoal reached! Solution found.")
            break

def get_user_initial_state(n):
    """
    Get initial state input from user.
```

```
    """
    while True:
        user_input = input(f"Enter the initial state as {n} integers (0 to {n-1}) separated by spaces:\n")
        parts = user_input.strip().split()
        if len(parts) != n:
            print(f"Error: Please enter exactly {n} integers.")
            continue

        try:
            state = [int(x) for x in parts]
        except ValueError:
            print("Error: Please enter valid integers.")
            continue

        if any(row < 0 or row >= n for row in state):
            print(f"Error: Each integer must be between 0 and {n-1}.")
            continue

        return state

if __name__ == "__main__":
    n = int(input("Enter the number of queens (N): "))
    initial_state = get_user_initial_state(n)
    hill_climbing(initial_state)

    print("\nSatish G K - 1BM23CS306")
```

Output:

```
Enter the number of queens (N): 4
Enter the initial state as 4 integers (0 to 3) separated by spaces:
2 0 3 1
Step 0: State = [2, 0, 3, 1], Cost = 0

Neighbors and their costs:
  [0, 0, 3, 1] -> Cost = 1
  [1, 0, 3, 1] -> Cost = 3
  [3, 0, 3, 1] -> Cost = 1
  [2, 1, 3, 1] -> Cost = 2
  [2, 2, 3, 1] -> Cost = 2
  [2, 3, 3, 1] -> Cost = 3
  [2, 0, 0, 1] -> Cost = 3
  [2, 0, 1, 1] -> Cost = 2
  [2, 0, 2, 1] -> Cost = 2
```

[2, 0, 3, 0] -> Cost = 1
[2, 0, 3, 2] -> Cost = 3
[2, 0, 3, 3] -> Cost = 1

No better neighbor found. Algorithm stops.

Satish G K - 1BM23CS306

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:

Week

- Simulated Annealing to solve 8 queens problem

Algorithm:

```
Current ← initial state
T ← a large positive value
while T > 0 do
    next ← a random neighbor of current
    ΔE ← current.cost - next.cost
    if ΔE > 0 then
        current ← next
    else
        current ← next with probability p = e^(ΔE/T)
    end if
    decrease T
end while
return current
```

Output:

The best position found: [5, 0, 4, 1, 7, 2, 6, 3]

cost = 0.

Code:

```
import random
import math

# Heuristic: number of attacking pairs
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]) == abs(i - j):
                cost += 1
    return cost

# Generate a random neighbor
def get_random_neighbor(state):
    n = len(state)
    new_state = list(state)
    col = random.randint(0, n - 1)   # pick random column
    row = random.randint(0, n - 1)   # new row
    new_state[col] = row
    return new_state

def simulated_annealing(n=8, max_iterations=10000, initial_temp=100.0, cooling_rate=0.99):
    # start with a random state
    current = [random.randint(0, n - 1) for _ in range(n)]
    current_cost = calculate_cost(current)
    best = current
    best_cost = current_cost
    temperature = initial_temp

    for _ in range(max_iterations):
        if current_cost == 0:
            break  # found solution

        neighbor = get_random_neighbor(current)
        neighbor_cost = calculate_cost(neighbor)
        delta = neighbor_cost - current_cost

        # Decide whether to accept the neighbor
        if delta < 0 or random.random() < math.exp(-delta / temperature):
            current, current_cost = neighbor, neighbor_cost

            # Update best if improved
```

```
            if current_cost < best_cost:
                best, best_cost = current, current_cost

        temperature *= cooling_rate
        if temperature < 1e-6:
            break

    return best, best_cost

# Run simulated annealing for 8 queens
best_state, best_cost = simulated_annealing()

print("The best position found:", best_state)
print("cost =", best_cost)

print("Satish G K - 1BM23CS306")
```

Output:
The best position found: [5, 0, 4, 1, 7, 2, 6, 3]
cost = 0
Satish G K - 1BM23CS306

Program 6

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

Algorithm:

Week-6 Propositional Logic 22/9/25

* Semantics:

→ Truth table for connectives.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \leftrightarrow Q$
false	false	true	false	false	true
false	true	true	false	true	false
true	false	false	false	true	false
true	true	false	true	true	true

* Propositional Inference: Enumeration Method

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

Checking that $KB \models \alpha$

A	B	C	$A \vee C$	$B \vee \neg C$	KB	α
F	F	F	F	T	F	F
F	F	T	T	F	F	F
F	T	F	F	T	F	T
F	T	T	T	T	$\boxed{T}$	$\boxed{T}$
T	F	F	T	T	$\boxed{T}$	$\boxed{T}$
T	F	T	T	F	F	T
T	T	F	T	T	$\boxed{T}$	$\boxed{T}$
T	T	T	T	T	$\boxed{T}$	$\boxed{T}$

Algorithm:

step 1: collect all the symbols

step 2: Try every possibility

Algorithm

1. Take KB and query (α) as input.
2. Build the table for all variables
3. Select only the rows where KB = True
4. If in all those rows α = True, then KB entails α, else it does not.

Q: Consider S & T as variables and following relation

a: $\neg(S \vee T)$

b: $(S \wedge T)$

c: $T \vee \neg T$

write truth table and show whether

i) a entails b

ii) a entails c

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

i) a entails b is false

ii) a entails c is true

Code:

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

# Helper: convert booleans to T/F
def tf(val):
    return "T" if val else "F"

# Replace logical symbols (∨, ∧, ¬) with Python equivalents
def translate(expr: str) -> str:
    return (
        expr.replace("∨", " or ")
            .replace("∧", " and ")
            .replace("¬", " not ")
    )

# === Take KB and α (query) from user ===
kb_expr = input("Enter your Knowledge Base (use ∧, ∨, ¬): ")
alpha_expr = input("Enter your α (query) (use ∧, ∨, ¬): ")

# Translate to Python syntax
kb_py = translate(kb_expr)
alpha_py = translate(alpha_expr)

# === Detect variables dynamically (any single letter, uppercase/lowercase)
===
vars_in_expr = sorted(set(re.findall(r"\b[a-zA-Z]\b", kb_expr +
alpha_expr)))

if not vars_in_expr:
    raise ValueError("No variables detected. Please use single letters like
A, b, C...")

# Generate truth table
rows = []
for values in itertools.product([False, True], repeat=len(vars_in_expr)):
    local_vars = dict(zip(vars_in_expr, values))
    kb_val = eval(kb_py, {}, local_vars)
    alpha_val = eval(alpha_py, {}, local_vars)

    row = {var: tf(val) for var, val in local_vars.items()}
    row["KB"] = tf(kb_val)
    row[f"α = {alpha_expr}"] = tf(alpha_val)
    rows.append(row)
```

```python
# Convert to DataFrame
df = pd.DataFrame(rows)

# Full truth table
print("\n=== Full Truth Table ===")
print(tabulate(df, headers="keys", tablefmt="fancy_grid", showindex=False))

# Filtered truth table (only KB = T)
filtered_df = df[df["KB"] == "T"]

print("\n=== Rows where KB is True ===")
if not filtered_df.empty:
    print(tabulate(filtered_df, headers="keys", tablefmt="fancy_grid",
showindex=False))
else:
    print("No rows where KB is True (KB is unsatisfiable).")

# Entailment check
query_col = f"α = {alpha_expr}"
entails = all(filtered_df[query_col] == "T") if not filtered_df.empty else
True
print(f"\nDoes KB entail α ({alpha_expr})? ->", "Yes" if entails else "No")
print("Satish G K - 1BM23CS306\n")
```

Output:

```
Enter your Knowledge Base (use ∧, ∨, ¬): (a or c) and (b or not c)
Enter your α (query) (use ∧, ∨, ¬): a or b
```

=== Full Truth Table ===

a	b	c	KB	α = a or b
F	F	F	F	F
F	F	T	F	F
F	T	F	F	T
F	T	T	T	T
T	F	F	T	T
T	F	T	F	T
T	T	F	T	T

T	T	T	T	T

=== Rows where KB is True ===

a	b	c	KB	α = a or b
F	T	T	T	T
T	F	F	T	T
T	T	F	T	T
T	T	T	T	T

Does KB entail α (a or b)? -> Yes
Satish G K - 1BM23CS306

Program 7
Implement unification in first order logic

Algorithm:

Week-7 Unification Algorithm 13/10/25

Unification is a process to find substitution that make different FOL (First Order Logic) identity

① Unify (Knows (John, x), knows (John, Jane))

$\theta = x / \text{Jane}$

Unify (Knows (John, Jane), knows (John, Jane))

True

Q: Find MGU of

$\{p(b, x, f(g(z)))\}$ $b \rightarrow z$

$x \rightarrow f(y)$

$\{p(z, f(y), f(y))\}$ $y \rightarrow g(z)$

$\{p(z, f(y), f(y))\}$ True

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

$\theta = b/z$, $\theta = x/f(y)$, $\theta = y/(g(z))$

$\{p(z, f(y), f(y))\}$

$\{p(z, f(y), f(y))\}$ True

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

~~$\theta = x/f(b)$~~, $\theta = y/b$

$\{Q(a, g(f(b), a), f(b)\}$

$\{Q(a, g(f(b), a), f(b)\}$ True

Q. Find the MGU of {p(f(a), g(y)), p(x, y)}

θ = x/f(a)   θ = f(a)/g(y) ×

Unification fails.

Q. Unify {prime(11) and prime(y)}

θ = y/11

{prime(11)}

{prime(y)} True

Q. Unify {knows(John, x), knows(y, mother(y))}

θ = y/John   θ = x/mother(y)

{knows(John, mother(y))}

{knows(John, mother(y))} True

Q. Unify {knows(John, x), knows(y, B, 11)}

θ = y/John,   θ = x/B, y

{knows(John, B, 11)}

{knows(John, B, 11)}

Algorithm: Unify(Ψ_1, Ψ_2)

Step 1: If Ψ_1 or Ψ_2 is a variable or constant, then:

a) If Ψ_1 or Ψ_2 are identical, then return NIL

b) Else if Ψ_1 is a variable,
   a. Then if Ψ_1 occurs in Ψ_2, then return Failure
   b. Else return {(Ψ_2/Ψ_1)}

c) Else if Ψ_2 is a variable,
   a. If Ψ_2 occurs in Ψ_1 then return Failure
   b. Else return {(Ψ_1/Ψ_2)}

d) Else return Failure

Step 2: If the initial predicate symbol in Ψ_1 and Ψ_2 are not same, then return Failure.

Step 3: If Ψ_1 and Ψ_2 have a different ~~number~~ number of arguments, then return Failure

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

Step 5: For i=1 to number of elements in Ψ_1.

a) Call Unify function with the ith element of Ψ_1 and ith element of Ψ_2, and put the result into S.

b) If S = failure then return Failure.

c) S ≠ NIL, then do,
   a. Apply S to the remainder of both L1 & L2
   b. SUBST = APPEND (S, SUBST)

Step 6: Return SUBST.

Output:

{p(b, x, R(y(z))), and p(z, R(y), R(y))}

MGU:

z/b

y/('R','y')

y/('y','z')

27.10

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
```

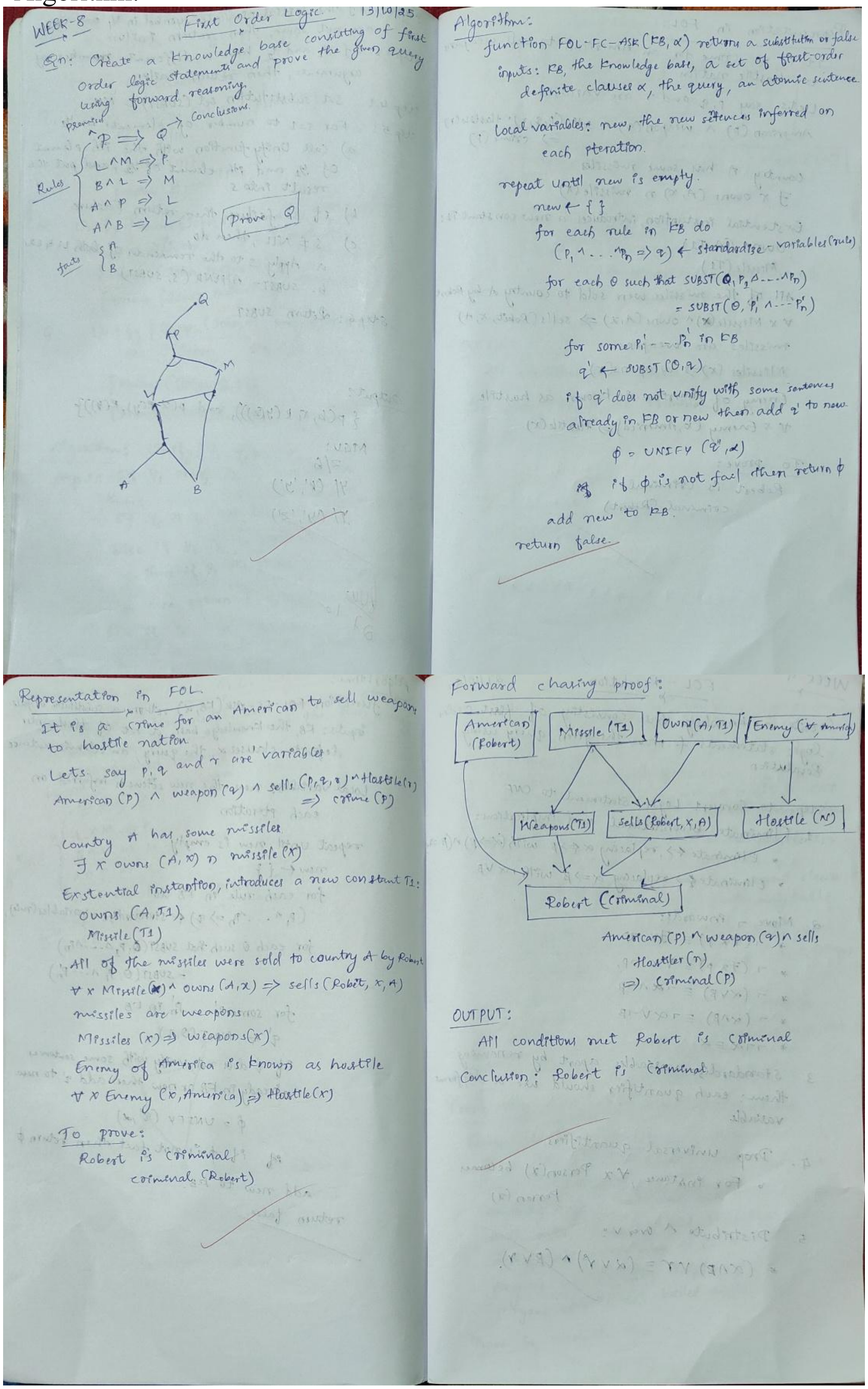
```python
    else:
        raise UnificationError(f"Cannot unify: {term1} with {term2}")

# 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}")
finally:
    print("1BM23CS306 SATISH G K")
```

Output:
Most General Unifier (MGU):
{'Z': 'b', 'X': ('f', 'Y'), 'Y': ('g', 'Z')}
1BM23CS306 SATISH G K

Program 8

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

Algorithm:



WEEK-8 First Order Logic 13/10/25

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

Premises $P \Rightarrow Q$ → Conclusions

Rules
$L \wedge M \Rightarrow P$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$

$\boxed{\text{Prove } Q}$

facts
A
B

Algorithm:

function FOL-FC-ASK(KB, α) returns a substitution or false
inputs: KB, the knowledge base, a set of first-order definite clauses α, the query, an atomic sentence

Local variables: new, the new sentences inferred on each iteration.

repeat until new is empty.
new ← { }
for each rule in KB do
$(p_1 \wedge \ldots \wedge p_n \Rightarrow q)$ ← standardize-variables(rule)
for each θ such that SUBST(θ, $p_1 \wedge \ldots \wedge p_n$) = SUBST($\theta$, $p_1' \wedge \ldots \wedge p_n'$)
for some $p_1' \ldots p_n'$ in KB
q' ← SUBST(θ, q)
if q' does not unify with some sentences already in KB or new then add q' to new
ϕ = UNIFY(q', α)
if ϕ is not fail then return ϕ
add new to KB.
return false.

Representation in FOL

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

Lets say p, q and r are variables
American(p) $\wedge$ weapon(q) $\wedge$ sells(p, q, r) $\wedge$ Hostile(r) $\Rightarrow$ crime(p)

country A has some missiles
$\exists x$ owns(A, x) $\wedge$ missile(x)

Existential instantion, introduces a new constant T1:
owns(A, T1)
Missile(T1)

All of the missiles were sold to country A by Robert
$\forall x$ Missile(x) $\wedge$ owns(A, x) $\Rightarrow$ sells(Robert, x, A)

missiles are weapons
Missiles(x) $\Rightarrow$ weapons(x)

Enemy of America is known as hostile
$\forall x$ Enemy(x, America) $\Rightarrow$ Hostile(x)

To prove:
Robert is criminal
criminal(Robert)

Forward chasing proof:

American(P) $\wedge$ weapon(q) $\wedge$ sells $\wedge$ Hostiler(r) $\Rightarrow$ Criminal(P)

OUTPUT:
All conditions met Robert is criminal

Conclusion: Robert is criminal

Code:

Code :

```
# Define the knowledge base
facts = {
    'American(Robert)': True,  # Robert is an American
    'Hostile(A)': True,        # Country A is hostile to America
    'Sells_Weapons(Robert, A)': True  # Robert sold weapons to Country A
}

# Define the law/rule: If American(X) and Hostile(Y) and Sells_Weapons(X, Y), then Crime(X)
def forward_reasoning(facts):
    # Apply the rule: If American(X) and Hostile(Y) and Sells_Weapons(X, Y), then Crime(X)
    if facts.get('American(Robert)', False) and facts.get('Hostile(A)', False) and
facts.get('Sells_Weapons(Robert, A)', False):
        facts['Crime(Robert)'] = True  # Robert is a criminal

# Perform forward reasoning to see if we can deduce that Robert is a criminal
forward_reasoning(facts)

# Output the result based on the fact derived
if facts.get('Crime(Robert)', False):
    print("Robert is a criminal.")
    print("Satish G K 1BM23CS306")
else:
    print("Robert is not a criminal.")
```

Output:

```
Robert is a criminal.
Satish G K 1BM23CS306
```

Program 9

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

Algorithm:

WEEK-9 FOL - Resolution 21/10/25

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

Steps to Convert Logic Statement to CNF

1. Eliminate biconditionals and Implications:
   - Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
   - Eliminate $\Rightarrow$ replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$.
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 variables apart by renaming them: each quantifier should use a different variable.
4. Drop universal quantifiers.
   - For instance, $\forall x$ Person(x) becomes Person(x)
5. Distribute $\wedge$ over $\vee$:
   - $(\alpha \wedge \beta) \vee \gamma \equiv (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$.

Resolution in FOL.

Basic steps for proving a conclusion S given premises Premise$_1$, ..., Premise$_n$ (all expressed in FOL):

1. Convert all sentences to CNF
2. Negate conclusion S & convert result to CNF
3. Add negated conclusion S to the premise clauses
4. Repeat until contradiction or no progress is made:
   a. Select 2 clauses (call them parent clauses)
   b. Resolve them together, performing all required unifications
   c. if resolvent is the empty clause, a contradiction has been found (i.e, S follows from the premises)
   d. If not, add resolvent to the premises.

If we succeed in step 4, we have proved the conclusion.

Proof by Resolution:

Given the KB or Premises:
John likes all kind of food
Apple and vegetables are food
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

Prove by resolution that:

Code:

Representation in FOL.

a. $\forall x$: food(x) $\rightarrow$ likes(John, x)
b. food(Apple) $\wedge$ food(vegetables)
c. $\forall x \forall y$: eats(x, y) $\wedge \neg$killed(x) $\rightarrow$ food(y)
d. eats(Anil, Peanuts) $\wedge$ alive(Anil)
e. $\forall x$: eats(Anil, x) $\rightarrow$ eats(Harry, x)
f. $\forall x$: $\neg$killed(x) $\rightarrow$ alive(x)
g. $\forall x$: alive(x) $\rightarrow \neg$killed(x)
h. likes(John, Peanuts)

Proof by Resolution:

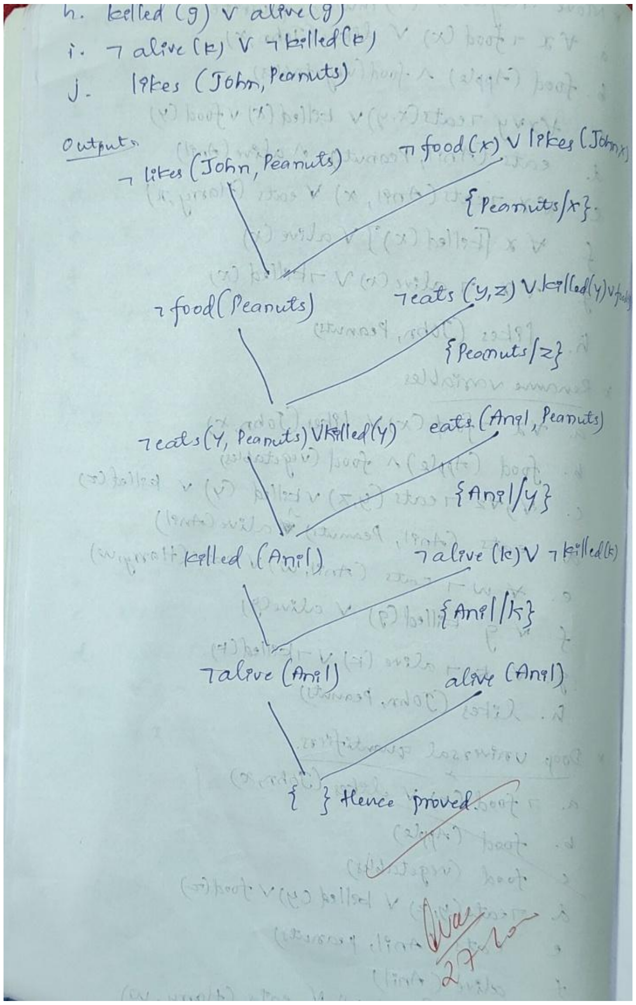
Eliminate implication:-
$\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$

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

* Move negation ($\neg$) inwards & rewrite
a. $\forall x\ \neg$food(x) $\vee$ likes(John, x)
b. food(Apple) $\wedge$ food(vegetables)
c. $\forall x \forall y\ \neg$eats(x, y) $\vee$ killed(x) $\vee$ food(y)
d. eats(Anil, Peanuts) $\wedge$ alive(Anil)
e. $\forall x\ \neg$eats(Anil, x) $\vee$ eats(Harry, x)
f. $\forall x$ [killed(x)] $\vee$ alive(x)
g. $\forall x\ \neg$alive(x) $\vee \neg$killed(x)
h. likes(John, Peanuts)

* Rename variables.
a. $\forall x\ \neg$food(x) $\vee$ likes(John, x)
b. food(Apple) $\wedge$ food(vegetables)
c. $\forall y \forall z\ \neg$eats(y, z) $\vee$ killed(y) $\vee$ killed(z)
d. eats(Anil, Peanuts) $\wedge$ alive(Anil)
e. $\forall w\ \neg$eats(Anil, w) $\vee$ eats(Harry, w)
f. $\forall g$ killed(g) $\vee$ alive(g)
g. $\forall k\ \neg$alive(k) $\vee \neg$killed(k)
h. likes(John, Peanuts)

* Drop universal quantifiers.
a. $\neg$food(x) $\vee$ likes(John, x)
b. food(Apple)
c. food(vegetables)
d. $\neg$eats(y, z) $\vee$ killed(y) $\vee$ food(z)
e. eats(Anil, peanuts)
f. alive(Anil)
g. $\neg$eats(Anil, w) $\vee$ eats(Harry, w)



Code:

```
import re
import itertools


def remove_implications(expr):
    expr = re.sub(r'\(([^()]*)->([^()]*)\)', r'(¬\1V\2)', expr)
    return expr.replace('->', 'V')

def move_negations(expr):
    expr = expr.replace('¬(¬', '(')
    expr = expr.replace('¬(∀', '∃¬')
    expr = expr.replace('¬(∃', '∀¬')
    expr = expr.replace('¬(A∧B)', '(¬AV¬B)')
    expr = expr.replace('¬(AVB)', '(¬A∧¬B)')
    return expr
```

```python
def drop_quantifiers(expr):
    return re.sub(r'[∀∃][a-z]\.', '', expr)

def distribute(expr):
    changed = True
    while changed:
        new_expr = re.sub(r'\(([^()]*)∨\(([^()]*)∧([^()]*)\)\)',
                          r'((\1∨\2)∧(\1∨\3))', expr)
        new_expr = re.sub(r'\(\(([^()]*)∧([^()]*)\)∨([^()]*)\)',
                          r'((\1∨\3)∧(\2∨\3))', new_expr)
        changed = new_expr != expr
        expr = new_expr
    return expr

def to_cnf(expr):
    expr = remove_implications(expr)
    expr = move_negations(expr)
    expr = drop_quantifiers(expr)
    expr = distribute(expr)
    return expr


KB = [
    "∀x.(Food(x) -> Likes(John,x))",
    "Food(Apple)",
    "Food(Vegetable)",
    "∀x∀y.((Eats(x,y) ∧ ¬Killed(x)) -> Food(y))",
    "(Eats(Anil,Peanuts) ∧ Alive(Anil))",
    "∀x∀y.(Eats(Anil,y) -> Eats(Harry,y))",
    "∀x.(Alive(x) -> ¬Killed(x))",
    "∀x.(¬Killed(x) -> Alive(x))"
]


goal = "Likes(John,Peanuts)"


print("=== Knowledge Base in CNF ===")
CNF_KB = [to_cnf(s) for s in KB]
for clause in CNF_KB:
    print(clause)


neg_goal = f"¬{goal}"
print("\nNegated Goal (for resolution):", neg_goal)
```

```
clauses = set(CNF_KB + [neg_goal])

def resolution(clauses):
    new = set()
    while True:
        pairs = [(c1, c2) for i, c1 in enumerate(clauses)
                 for c2 in list(clauses)[i + 1:]]
        for (ci, cj) in pairs:
            resolvents = resolve(ci, cj)
            if "" in resolvents:
                return True
            new |= set(resolvents)
        if new.issubset(clauses):
            return False
        clauses |= new

def resolve(ci, cj):
    resolvents = set()
    ci_literals = set(ci.replace("(", "").replace(")", "").split("∧"))
    cj_literals = set(cj.replace("(", "").replace(")", "").split("∧"))
    for di in ci_literals:
        for dj in cj_literals:
            if di.strip() == ("¬" + dj.strip()) or dj.strip() == ("¬" +
di.strip()):
                new_clause = (ci_literals | cj_literals) - {di, dj}
                resolvents.add("∧".join(new_clause))
    return resolvents

proved = resolution(clauses)
print("\nCan we prove that John likes peanuts?")
print("Result:", "YES (derived contradiction ⇒ proved)" if proved else "NO
(cannot prove)")
print("Satish G K 1BM23CS306")
```

Output:
=== Knowledge Base in CNF ===
(Food(x) ∨ Likes(John,x))
Food(Apple)
Food(Vegetable)
∀x((Eats(x,y) ∧ ¬Killed(x)) ∨ Food(y))
(Eats(Anil,Peanuts) ∧ Alive(Anil))
∀x(Eats(Anil,y) ∨ Eats(Harry,y))
(Alive(x) ∨ ¬Killed(x))

(¬Killed(x) ∨ Alive(x))

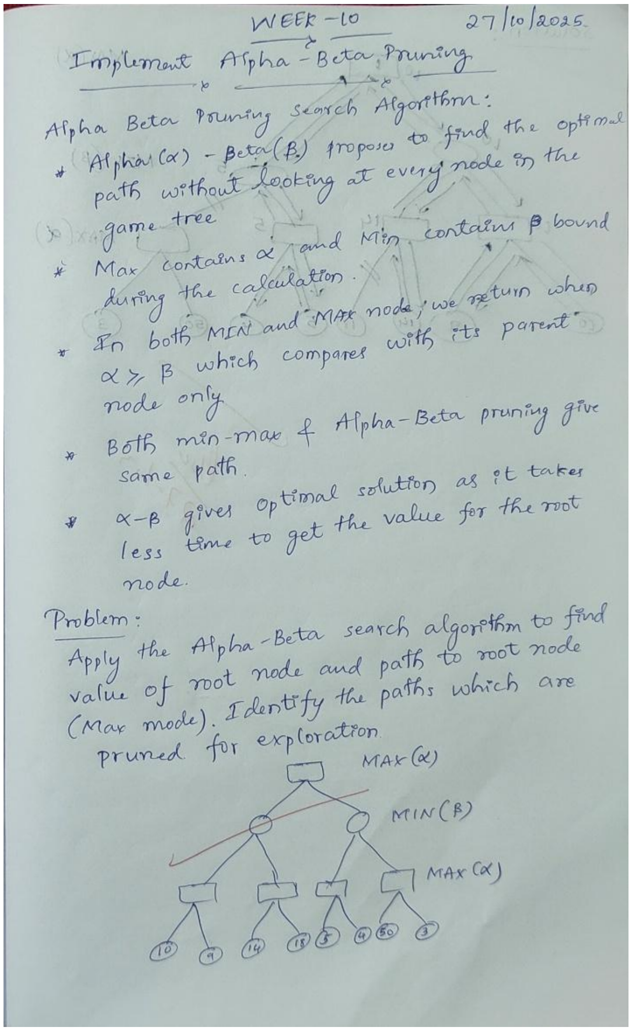
Negated Goal (for resolution): ¬Likes(John,Peanuts)

Can we prove that John likes peanuts?
Result: NO (cannot prove)
Satish G K 1BM23CS306

Program 10

Implement Alpha-Beta Pruning.

Algorithm:



WEEK-10 27/10/2025

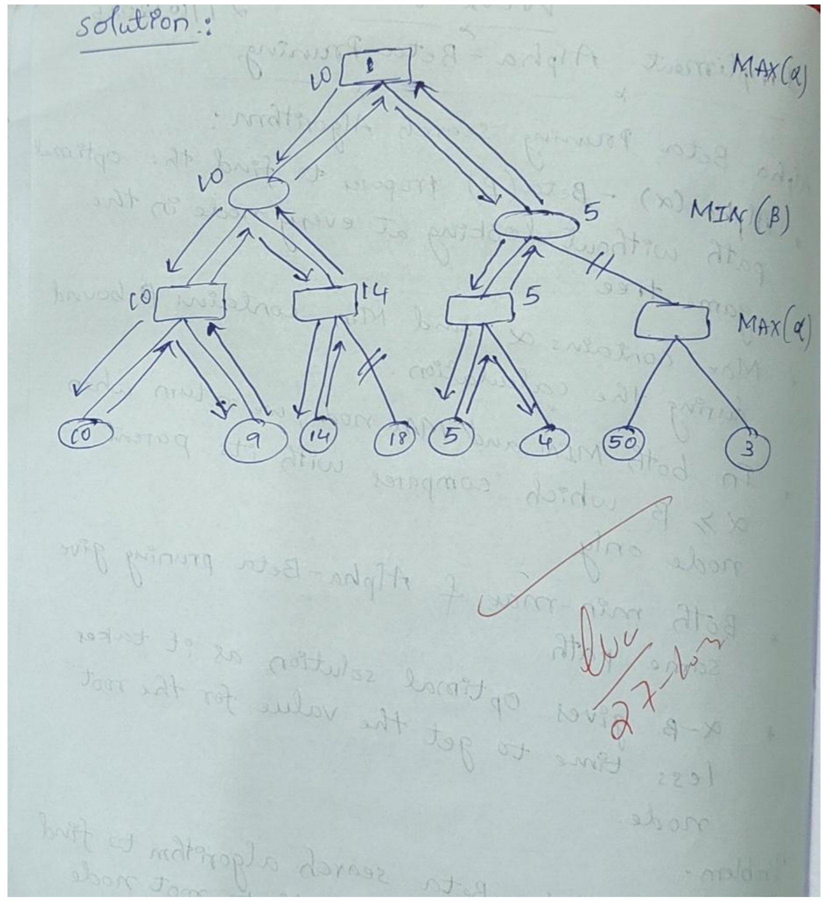
Implement Alpha-Beta Pruning

Alpha Beta Pruning search Algorithm:

* Alpha(α) - Beta(β) propose to find the optimal path without looking at every node in the game tree
* Max contains α and Min contains β bound during the calculation.
* In both MIN and MAX node, we return when α ≥ β which compares with its parent node only
* Both min-max & Alpha-Beta pruning give same path.
* α-β gives optimal solution as it takes less time to get the value for the root node.

Problem:

Apply the Alpha-Beta search algorithm to find value of root node and path to root node (Max mode). Identify the paths which are pruned for exploration.



Code :

```
import math

tree = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': ['H', 'I'],
    'E': ['J', 'K'],
    'F': ['L', 'M'],
    'G': ['N', 'O'],
```

```
    'H': [], 'I': [], 'J': [], 'K': [],
    'L': [], 'M': [], 'N': [], 'O': []
}

# Leaf node values
values = {
    'H': 10, 'I': 9,
    'J': 14, 'K': 18,
    'L': 5, 'M': 4,
    'N': 50, 'O': 3
}

# to store final display values
node_values = {}

def get_children(node):
    return tree.get(node, [])

def is_terminal(node):
    return len(get_children(node)) == 0

def evaluate(node):
    return values[node]

def alpha_beta(node, depth, alpha, beta, maximizing):
    if is_terminal(node) or depth == 0:
        val = evaluate(node)
        node_values[node] = val
        return val

    if maximizing:
        value = -math.inf
        for child in get_children(node):
            val = alpha_beta(child, depth - 1, alpha, beta, False)
            value = max(value, val)
            alpha = max(alpha, val)
            if beta <= alpha:
                # mark remaining children as pruned
                for rem in get_children(node)[get_children(node).index(child)+1:]:
                    node_values[rem] = "X"
                break
```

```
        node_values[node] = value
        return value
    else:
        value = math.inf
        for child in get_children(node):
            val = alpha_beta(child, depth - 1, alpha, beta, True)
            value = min(value, val)
            beta = min(beta, val)
            if beta <= alpha:
                for rem in get_children(node)[get_children(node).index(child)+1:]:
                    node_values[rem] = "X"
                break
        node_values[node] = value
        return value

# Run pruning
alpha_beta('A', depth=4, alpha=-math.inf, beta=math.inf, maximizing=True)

def print_tree(node, prefix="", is_last=True):
    connector = " └── " if is_last else " ├── "
    value = node_values.get(node, "")
    print(prefix + connector + f"{node} ({value})")
    children = get_children(node)
    for i, child in enumerate(children):
        new_prefix = prefix + ("    " if is_last else " │   ")
        print_tree(child, new_prefix, i == len(children)-1)

# Display the final tree
print("\nFINAL TREE\n" )
print_tree('A')

print("\n Satish G K - 1BM23CS306")
```

Output:
FINAL TREE

```
└── A (10)
    ├── B (10)
    │   ├── D (10)
    │   │   ├── H (10)
    │   │   └── I (9)
    │   └── E (14)
    │       ├── J (14)
    │       └── K (X)
    └── C (5)
        ├── F (5)
        │   ├── L (5)
        │   └── M (4)
        └── G (X)
            ├── N ()
            └── O ()
```

Satish G K - 1BM23CS306