VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Sharada Koundinya (1BM23CS310)

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 **Sharada Koundinya (1BM23CS310),** 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.

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

Index

INDEX

NAME: Sharada SEM: V SEC.: F ROLL NO.: 310 SUB.: AI Lab

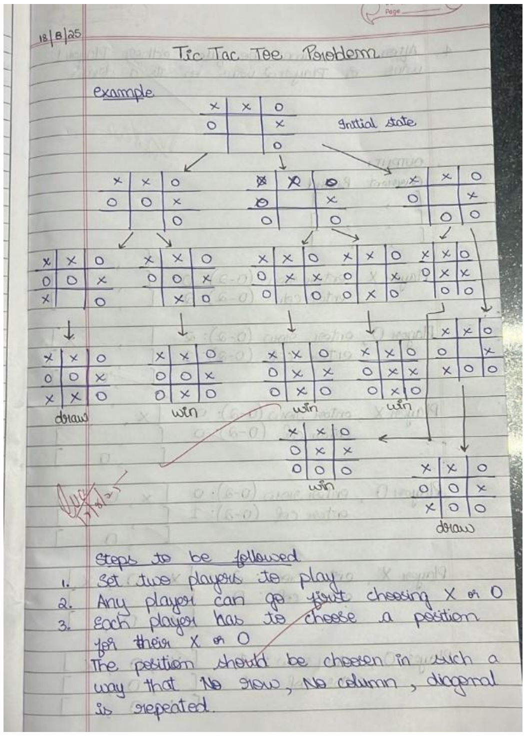
S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1.	18/8/25	Tic Tac Toe	1	10 18/8/25
2.	25/8/25	Vaccum Cleaner	4	10 25/8/25
3a.	25/8/25	BFS 8 puzzle w/out Heuristic approach	5	10 1/9/25
3b.	1/9/25	DFS w/out Heuristic approach	6	
3c.	1/9/25	iterative DFS	9	
4a.	8/9/25	A* using misplaced tile	10	10 8/9/25
4b.	8/9/25	A* using manhatten distance	11	
5.	15/9/25	Hill climbing algo	14	10 15/9
6.	15/9/25	Stimulated Annealing	16	
7.	22/9/25	Propositional Logic	17	10 22/9
8.	13/10/25	Unification in FOL	19	10 13/10
9.	13/10/25	First Order Logic	21	
10.	27/10/25	KB of First Order logic	23	10 27/10
11.	27/10/25	Alpha Beta Purning	26	

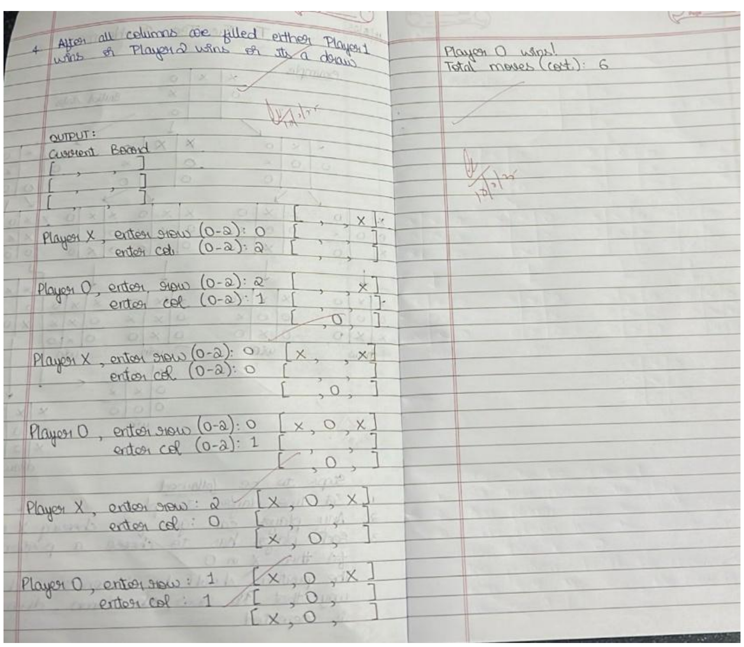
Completed.

Github Link:
https://github.com/sharadakoundinya/ailab.git

Program 1
Implement Tic –Tac –Toe Game

Algorithm:





Code:

```
def print_board(board):
    print("\nCurrent Board:")
    for row in board:
        print(row)
    print()


def check_winner(board, player):
    for i in range(3):
        if all(cell == player for cell in board[i]):
            return True
        if all(board[j][i] == player for j in range(3)):
            return True


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

    return False


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


def tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = "X"
    move_count = 0

    print("Tic-Tac-Toe Game (3x3 Matrix Format)\n")
    print_board(board)

    while True:
        try:
            row = int(input(f"Player {current_player}, enter row (0-2): "))
            col = int(input(f"Player {current_player}, enter col (0-2): "))
        except ValueError:
            print("Please enter integers between 0 and 2.")
            continue

        if not (0 <= row <= 2 and 0 <= col <= 2):
```

```
            print("Invalid position. Try again.")
            continue
        if board[row][col] != " ":
            print("Cell already filled. Choose another.")
            continue

        board[row][col] = current_player
        move_count += 1
        print_board(board)

        if check_winner(board, current_player):
            print(f"Player {current_player} wins!")
            break

        if is_full(board):
            print("Game is a draw.")
            break

        current_player = "O" if current_player == "X" else "X"

    print(f"Total moves (cost): {move_count}")

tic_tac_toe()
```

Output case1:

```
Tic-Tac-Toe Game (3x3 Matrix Format)


Current Board:
[' ', ' ', ' ']
[' ', ' ', ' ']
[' ', ' ', ' ']

Player X, enter row (0-2): 1
Player X, enter col (0-2): 1

Current Board:
[' ', ' ', ' ']
[' ', 'X', ' ']
[' ', ' ', ' ']

Player O, enter row (0-2): 0
Player O, enter col (0-2): 2

Current Board:
[' ', ' ', 'O']
[' ', 'X', ' ']
[' ', ' ', ' ']

Player X, enter row (0-2): 1
Player X, enter col (0-2): 0
```

```
Current Board:
[' ', ' ', 'O']
['X', 'X', ' ']
[' ', ' ', ' ']

Player O, enter row (0-2): 2
Player O, enter col (0-2): 1

Current Board:
[' ', ' ', 'O']
['X', 'X', ' ']
[' ', 'O', ' ']

Player X, enter row (0-2): 2
Player X, enter col (0-2): 2

Current Board:
[' ', ' ', 'O']
['X', 'X', ' ']
[' ', 'O', 'X']

Player O, enter row (0-2): 2
Player O, enter col (0-2): 0

Current Board:
[' ', ' ', 'O']
['X', 'X', ' ']
['O', 'O', 'X']

Player X, enter row (0-2): 0
Player X, enter col (0-2): 1

Current Board:
[' ', 'X', 'O']
['X', 'X', ' ']
['O', 'O', 'X']

Player O, enter row (0-2): 1
Player O, enter col (0-2): 2

Current Board:
[' ', 'X', 'O']
['X', 'X', 'O']
['O', 'O', 'X']

Player X, enter row (0-2): 0
Player X, enter col (0-2): 0

Current Board:
['X', 'X', 'O']
['X', 'X', 'O']
['O', 'O', 'X']

Player X wins!
Total moves (cost): 9
```

Output case2:

```
Tic-Tac-Toe Game (3x3 Matrix Format)


Current Board:
[' ', ' ', ' ']
[' ', ' ', ' ']
[' ', ' ', ' ']

Player X, enter row (0-2): 0
Player X, enter col (0-2): 2

Current Board:
[' ', ' ', 'X']
[' ', ' ', ' ']
[' ', ' ', ' ']

Player O, enter row (0-2): 2
Player O, enter col (0-2): 1

Current Board:
[' ', ' ', 'X']
[' ', ' ', ' ']
[' ', 'O', ' ']

Player X, enter row (0-2): 0
Player X, enter col (0-2): 0

Current Board:
['X', ' ', 'X']
[' ', ' ', ' ']
[' ', 'O', ' ']

Player O, enter row (0-2): 0
Player O, enter col (0-2): 1

Current Board:
['X', 'O', 'X']
[' ', ' ', ' ']
[' ', 'O', ' ']

Player X, enter row (0-2): 2
Player X, enter col (0-2): 0

Current Board:
['X', 'O', 'X']
[' ', ' ', ' ']
['X', 'O', ' ']

Player O, enter row (0-2): 1
Player O, enter col (0-2): 1

Current Board:
['X', 'O', 'X']
[' ', 'O', ' ']
```

```
['X', 'O', ' ']

Player O wins!
Total moves (cost): 6
```

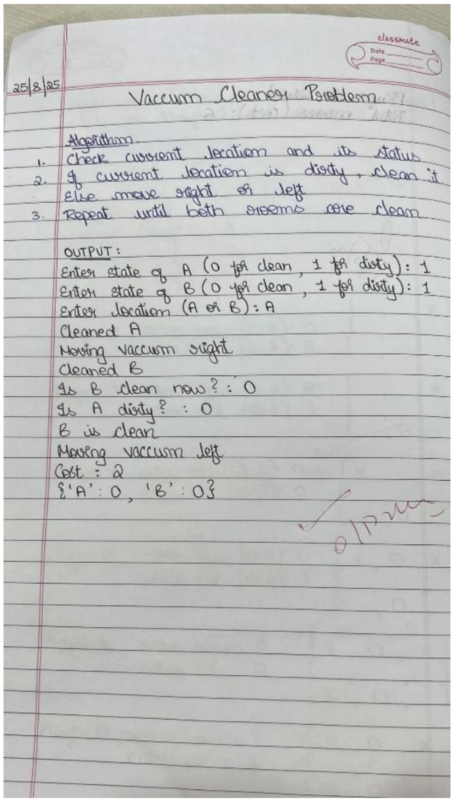
Output case3:

```
Tic-Tac-Toe Game (3x3 Matrix Format):

Current Board:
[' ', ' ', ' ']
[' ', ' ', ' ']
[' ', ' ', ' ']


Player X, enter row (0-2): 1
Player X, enter col (0-2): 0
Current Board:
[' ', ' ', ' ']
['X', ' ', ' ']
[' ', ' ', ' ']

Player O, enter row (0-2): 0
Player O, enter col (0-2): 2

Current Board:
[' ', ' ', 'O']
['X', ' ', ' ']
[' ', ' ', ' ']

Player X, enter row (0-2): 2
Player X, enter col (0-2): 0

Current Board:
[' ', ' ', 'O']
['X', ' ', ' ']
['X', ' ', ' ']

Player O, enter row (0-2): 0
Player O, enter col (0-2): 0

Current Board:
['O', ' ', 'O']
['X', ' ', ' ']
['X', ' ', ' ']

Player X, enter row (0-2): 0
Player X, enter col (0-2): 1

Current Board:
['O', 'X', 'O']
['X', ' ', ' ']
['X', ' ', ' ']

Player O, enter row (0-2): 2
Player O, enter col (0-2): 1
```

```
Current Board:
['O', 'X', 'O']
['X', ' ', ' ']
['X', 'O', ' ']

Player X, enter row (0-2): 2
Player X, enter col (0-2): 2

Current Board:
['O', 'X', 'O']
['X', ' ', ' ']
['X', 'O', 'X']

Player O, enter row (0-2): 1
Player O, enter col (0-2): 1

Current Board:
['O', 'X', 'O']
['X', 'O', ' ']
['X', 'O', 'X']

Player X, enter row (0-2): 1
Player X, enter col (0-2): 2

Current Board:
['O', 'X', 'O']
['X', 'O', 'X']
['X', 'O', 'X']

Game is a draw.
Total moves (cost): 9
```

Implement vacuum cleaner agent

Algorithm:


25/8/25 Vaccum Cleaner Problem

Algorithm

1. Check current location and its status
2. If current location is dirty, clean it else move right or left
3. Repeat until both rooms are clean

OUTPUT:

Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): A
Cleaned A
Moving vaccum right
Cleaned B
Is B clean now? : 0
Is A dirty? : 0
B is clean
Moving vaccum left
Cost : 2
{'A' : 0, 'B' : 0}

Code:

```
def vacuum_cleaner()
    A = int(input("Enter state of A (0 for clean, 1 for dirty): "))
    B = int(input("Enter state of B (0 for clean, 1 for dirty): "))
    location = input("Enter location (A or B): ").upper()

    cost = 0
    state = {'A': A, 'B': B}

    if location == 'A':
        if state['A'] == 1:  # If A is dirty
            print("Cleaned A.")
            state['A'] = 0
            cost += 1
        else:
            print("A is clean")

        if state['B'] == 1:  # If B is dirty
            print("Moving vacuum right")
            print("Cleaned B.")
```

```
            state['B'] = 0
            cost += 1
            print("Is B clean now? (0 if clean, 1 if dirty):", state['B'])
            print("Is A dirty? (0 if clean, 1 if dirty):", state['A'])
            print("B is clean")
            print("Moving vacuum left")
        else:
            print("Turning vacuum off")

    elif location == 'B':
        if state['B'] == 1:  # If B is dirty
            print("Cleaned B.")
            state['B'] = 0
            cost += 1
        else:
            print("B is clean")

        if state['A'] == 1:  # If A is dirty
            print("Moving vacuum left")
            print("Cleaned A.")
            state['A'] = 0
            cost += 1
            print("Is A clean now? (0 if clean, 1 if dirty):", state['A'])
            print("Is B dirty? (0 if clean, 1 if dirty):", state['B'])
            print("A is clean")
            print("Moving vacuum right")
        else:
            print("Turning vacuum off")

    print("Cost:", cost)
    print(state)
    print("Sharada Koundinya, 1BM23CS310")

vacuum_cleaner()
```

OUTPUT Case1:

```
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): A
Cleaned A.
Moving vacuum right
Cleaned B.
Is B clean now? (0 if clean, 1 if dirty): 0
Is A dirty? (0 if clean, 1 if dirty): 0
B is clean
Moving vacuum left
Cost: 2
{'A': 0, 'B': 0}
Sharada Koundinya, 1BM23CS310
```

OUTPUT Case2:

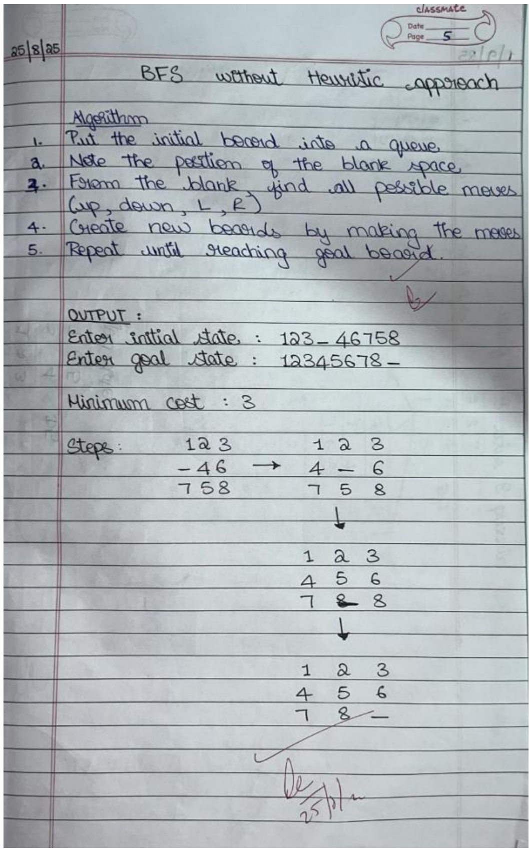
```
Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): A
A is clean
Moving vacuum right
Cleaned B.
Is B clean now? (0 if clean, 1 if dirty): 0
Is A dirty? (0 if clean, 1 if dirty): 0
B is clean
Moving vacuum left
Cost: 1
{'A': 0, 'B': 0}
Sharada Koundinya, 1BM23CS310
```

OUTPUT Case3:

```
Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A
A is clean
Turning vacuum off
Cost: 0
{'A': 0, 'B': 0}
Sharada Koundinya, 1BM23CS310
```

Program2
Implement 8 puzzle problems using Breath First Search (BFS)

Algorithm:


25/8/25

BFS without Heuristic approach

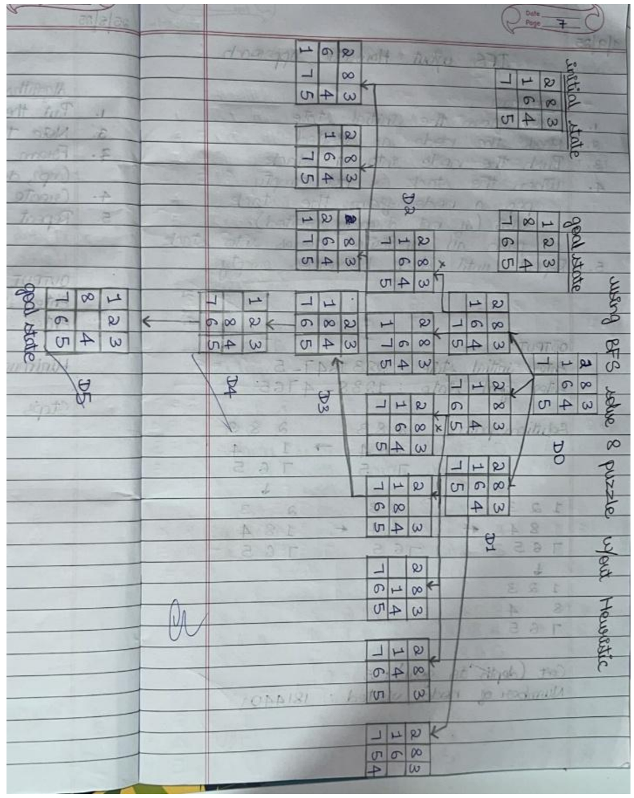
Algorithm
1. Put the initial board into a queue
2. Note the position of the blank space
3. From the blank, find all possible moves (up, down, L, R)
4. Create new boards by making the moves
5. Repeat until reaching goal board.

OUTPUT :
Enter initial state : 123_46758
Enter goal state : 12345678_

Minimum cost : 3

Steps:

```
123      1 2 3
_46  →   4 _ 6
758      7 5 8
           ↓
         1 2 3
         4 5 6
         7 _ 8
           ↓
         1 2 3
         4 5 6
         7 8 _
```



Code:

```
#bfs without heuristic approach showing visited node
from collections import deque

def get_moves(state):
    idx = state.index("_")
    x, y = divmod(idx, 3)
    moves = []
    for dx, dy in [(-1,0),(1,0),(0,-1),(0,1)]:
        nx, ny = x+dx, y+dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            nidx = nx*3 + ny
            lst = list(state)
            lst[idx], lst[nidx] = lst[nidx], lst[idx]
            moves.append("".join(lst))
    return moves

def bfs(start, goal):
    q = deque([(start, 0)])
    parent = {start: None}
```

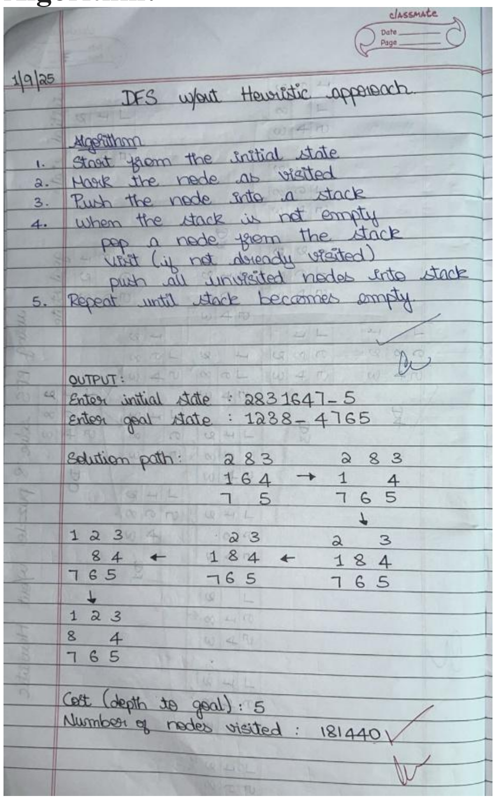
```
    visited = {start}
    order = []
    while q:
        state, cost = q.popleft()
        order.append(state)
        if state == goal:   # stop immediately
            path = []
            while state:
                path.append(state)
                state = parent[state]
            path.reverse()
            return path, cost, order
        for move in get_moves(state):
            if move not in visited:
                visited.add(move)
                parent[move] = state
                q.append((move, cost+1))
    return None, -1, order   # if no solution

start = input("Enter initial state (e.g., 54_618732): ")
goal = input("Enter goal state (e.g., 12345678_): ")
path, cost, visited = bfs(start, goal)

print("Minimum cost:", cost)
print("\nSteps:")
for p in path:
    for i in range(0, 9, 3):
        print(p[i:i+3])
    print()

print("Visited states:")
for v in visited:
    for i in range(0, 9, 3):
        print(v[i:i+3])
    print()

print("Sharada Koundinya,1BM23CS310")
```

Output:

```
Enter initial state (e.g., 54_618732): 2831647_5
Enter goal state (e.g., 12345678_): 1238_4765
Minimum cost: 5

Steps:
283
164
7_5

283
1_4
765

2_3
184
765

_23
184
765

123
_84
765

123
8_4
765
```

Implement 8 puzzle problems using Depth First Search (DFS)

Algorithm:



1/9/25

DFS w/out Heuristic approach.

Algorithm

1. Start from the initial state
2. Mark the node as visited
3. Push the node into a stack
4. When the stack is not empty
    pop a node from the stack
    visit (if not already visited)
    push all unvisited nodes into stack
5. Repeat until stack becomes empty.

OUTPUT:

Enter initial state : 2831647_5

Enter goal state : 1238_4765

Solution path:

```
2 8 3        2 8 3
1 6 4   →    1   4
7   5        7 6 5
                ↓
1 2 3         2 3        2   3
  8 4   ←   1 8 4   ←    1 8 4
7 6 5       7 6 5        7 6 5
  ↓
1 2 3
8   4
7 6 5
```

Cost (depth to goal) : 5

Number of nodes visited : 181440

Code:

```
def get_moves(state):
    idx = state.index("_")
    x, y = divmod(idx, 3)
    moves = []
    for dx, dy in [(-1,0),(1,0),(0,-1),(0,1)]:
        nx, ny = x+dx, y+dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            nidx = nx*3 + ny
            lst = list(state)
            lst[idx], lst[nidx] = lst[nidx], lst[idx]
            moves.append("".join(lst))
    return moves

def dfs(start, goal):
    stack = [(start, 0)]
    parent = {start: None}
    visited = {start}
    order = []

    while stack:
        state, cost = stack.pop()
        order.append(state)
```

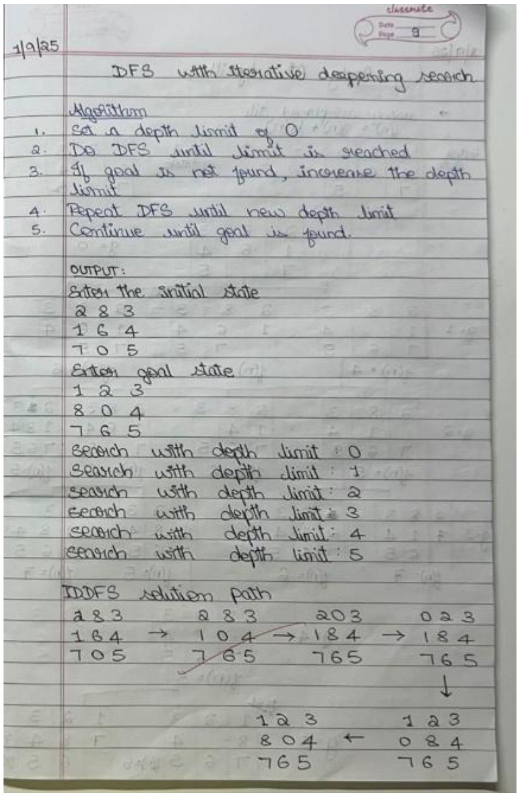
```
        if state == goal:
            path = []
            while state:
                path.append(state)
                state = parent[state]
            path.reverse()
            return path, cost, order, visited
        for move in reversed(get_moves(state)):
            if move not in visited:
                visited.add(move)
                parent[move] = state
                stack.append((move, cost+1))
    return None, -1, order, visited

start = input("Enter initial state (e.g., 54_618732): ")
goal = input("Enter goal state (e.g., 12345678_): ")
path, cost, visited_order, visited_set = dfs(start, goal)

print("Visited nodes (till goal found):")
for v in visited_order:
    for i in range(0, 9, 3):
        print(v[i:i+3])
    print()
    if v == goal:
        break

print("Steps (solution path):")
for p in path:
    for i in range(0, 9, 3):
        print(p[i:i+3])
    print()

print("Cost (depth to goal):", cost)
print("Number of nodes visited:", len(visited_set))

print("Sharada Koundinya,1BM23CS310")
```

Output:

```
Steps (solution path):
283
164
7_5

283
1_4
765

2_3
184
765

_23
184
765

123
_84
765

123
8_4
765

Cost (depth to goal): 5
Number of nodes visited: 181440
Sharada Koundinya,1BM23CS310
```

Implement Iterative deepening search algorithm

Algorithm:



1/9/25

DFS with Iterative deepening search

Algorithm

1. Set a depth limit of 0
2. Do DFS until limit is reached
3. If goal is not found, increase the depth limit
4. Repeat DFS until new depth limit
5. Continue until goal is found.

OUTPUT:

Enter the initial state

2 8 3
1 6 4
7 0 5

Enter goal state

1 2 3
8 0 4
7 6 5

Search with depth limit : 0
Search with depth limit : 1
Search with depth limit : 2
Search with depth limit : 3
Search with depth limit : 4
Search with depth limit : 5

IDDFS solution path

```
283      283      203      023
164  →   104  →   184  →   184
705      765      765      765
                            ↓
         123               123
         804      ←        084
         765               765
```

Code:

```
def get_neighbors(state):
    neighbors = []
    idx = state.index("0")
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    x, y = divmod(idx, 3)

    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_idx = nx * 3 + ny
            state_list = list(state)
            state_list[idx], state_list[new_idx] = state_list[new_idx], state_list[idx]
            neighbors.append("".join(state_list))
    return neighbors

def dfs_limit(start_state, goal_state, limit):
    stack = [(start_state, 0)]
    visited = set()
    parent = {start_state: None}
    path = []
```

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

        if current_state == goal_state:
            while current_state:
                path.append(current_state)
                current_state = parent[current_state]
            return path[::-1]

        if depth < limit and current_state not in visited:
            visited.add(current_state)
            neighbors = get_neighbors(current_state)
            neighbors.reverse()  # Maintain consistent exploration order
            for neighbor in neighbors:
                if neighbor not in visited:
                    parent[neighbor] = current_state
                    stack.append((neighbor, depth + 1))
    return None

def iddfs(start_state, goal_state, max_depth):
    for limit in range(max_depth + 1):
        print(f"Searching with depth limit: {limit}")
        solution = dfs_limit(start_state, goal_state, limit)
        if solution:
            return solution
    return None

print("Sharada Koundinya 1BM23CS310")
print("Enter the initial state (enter 3 digits per row, separated by spaces, 0 for empty):")
initial_state_rows = []
for i in range(3):
    row = input(f"Row {i+1}: ").split()
    initial_state_rows.extend(row)
initial_state = "".join(initial_state_rows)

print("\nEnter the goal state (enter 3 digits per row, separated by spaces, 0 for empty):")
goal_state_rows = []
for i in range(3):
    row = input(f"Row {i+1}: ").split()
    goal_state_rows.extend(row)
goal_state = "".join(goal_state_rows)

max_depth = 50

solution = iddfs(initial_state, goal_state, max_depth)

if solution:
    print("\nIDDFS solution path:")
    for s in solution:
        print(s[:3])
        print(s[3:6])
        print(s[6:])
```

```
        print()
else:
    print(f"\nNo solution found within the maximum depth of {max_depth}.")
```

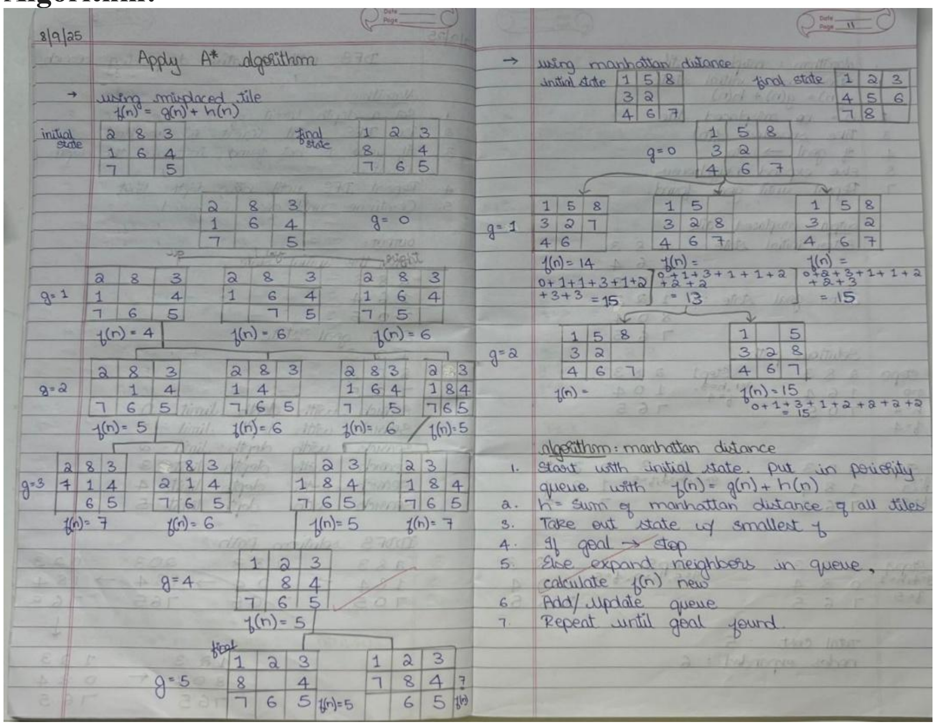
Output:

```
Sharada Koundinya 1BM23CS310
Enter the initial state (enter 3 digits per row, separated by spaces, 0 for empty):
Row 1: 283
Row 2: 164
Row 3: 705

Enter the goal state (enter 3 digits per row, separated by spaces, 0 for empty):
Row 1: 123
Row 2: 804
Row 3: 765
Searching with depth limit: 0
Searching with depth limit: 1
Searching with depth limit: 2
Searching with depth limit: 3
Searching with depth limit: 4
Searching with depth limit: 5

IDDFS solution path:
283
164
705

283
104
765

203
184
765

023
184
765

123
084
765

123
804
765
```

Program3

Implement A* search algorithm

Algorithm:



8/9/25

Apply A* algorithm

→ using misplaced tile

f(n) = g(n) + h(n)

initial state

2	8	3
1	6	4
7		5

final state

1	2	3
8		4
7	6	5

g = 0, g = 1: f(n) = 4, f(n) = 6, f(n) = 6

g = 2: f(n) = 5, f(n) = 6, f(n) = 6, f(n) = 5

g = 3: f(n) = 7, f(n) = 6, f(n) = 5, f(n) = 7

g = 4: f(n) = 5

g = 5: final, f(n) = 5

→ using manhattan distance

initial state

1	5	8
3	2	
4	6	7

final state

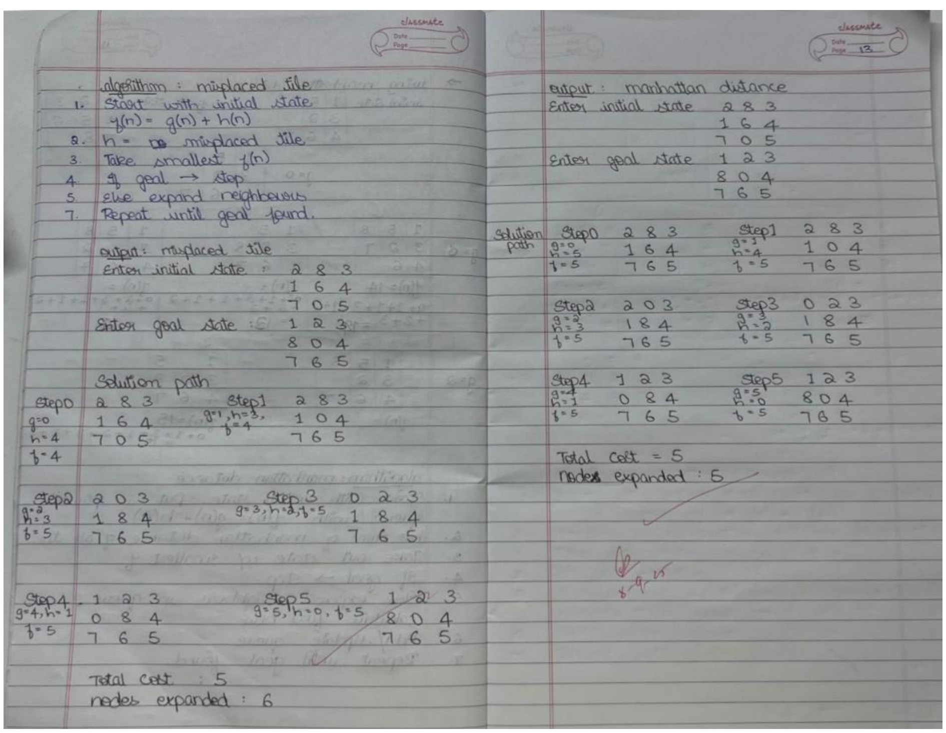
1	2	3
4	5	6
7	8	

g = 0, g = 1: f(n) = 14 (0+1+1+3+1+2+3+3 = 15), f(n) = 13, f(n) = 15

g = 2: f(n) = 15

algorithm: manhattan distance

1. Start with initial state. put in priority queue with f(n) = g(n) + h(n)
2. h = sum of manhattan distance of all tiles
3. Take out state w/ smallest f
4. If goal → stop
5. Else expand neighbors in queue, calculate f(n) new
6. Add/update queue
7. Repeat until goal found.



algorithm: misplaced tile

1. Start with initial state, f(n) = g(n) + h(n)
2. h = misplaced tile
3. Take smallest f(n)
4. If goal → stop
5. Else expand neighbours
7. Repeat until goal found.

output: misplaced tile

Enter initial state: 2 8 3 / 1 6 4 / 7 0 5

Enter goal state: 1 2 3 / 8 0 4 / 7 6 5

Solution path

Step0 (g=0, h=4, f=4): 2 8 3 / 1 6 4 / 7 0 5

Step1 (g=1, h=3, f=4): 2 8 3 / 1 0 4 / 7 6 5

Step2 (g=2, h=3, f=5): 2 0 3 / 1 8 4 / 7 6 5

Step3 (g=3, h=2, f=5): 0 2 3 / 1 8 4 / 7 6 5

Step4 (g=4, h=1, f=5): 1 2 3 / 0 8 4 / 7 6 5

Step5 (g=5, h=0, f=5): 1 2 3 / 8 0 4 / 7 6 5

Total cost : 5

nodes expanded : 6

output: manhattan distance

Enter initial state: 2 8 3 / 1 6 4 / 7 0 5

Enter goal state: 1 2 3 / 8 0 4 / 7 6 5

Solution path

Step0 (g=0, h=5, f=5): 2 8 3 / 1 6 4 / 7 6 5

Step1 (g=1, h=4, f=5): 2 8 3 / 1 0 4 / 7 6 5

Step2 (g=2, h=3, f=5): 2 0 3 / 1 8 4 / 7 6 5

Step3 (g=3, h=2, f=5): 0 2 3 / 1 8 4 / 7 6 5

Step4 (g=4, h=1, f=5): 1 2 3 / 0 8 4 / 7 6 5

Step5 (g=5, h=0, f=5): 1 2 3 / 8 0 4 / 7 6 5

Total cost = 5

nodes expanded : 5

8-9-25

Code:

```
#MISPLACED TILE
import heapq
from itertools import count

def misplaced_heuristic(board, goal):
    """h(n): number of tiles not in their goal position (excluding blank 0)."""
    n = len(board)
    misplaced = 0
    for i in range(n):
        for j in range(n):
            if board[i][j] != 0 and board[i][j] != goal[i][j]:
                misplaced += 1
    return misplaced

def find_blank(board):
    n = len(board)
    for i in range(n):
        for j in range(n):
            if board[i][j] == 0:
                return i, j
    raise ValueError("Board does not contain a blank tile (0)")

def neighbors(board):
    """Generate neighboring boards by sliding one tile into the blank."""
    n = len(board)
    x, y = find_blank(board)
    dirs = [(0,1),(0,-1),(1,0),(-1,0)]
    res = []
    for dx, dy in dirs:
        nx, ny = x + dx, y + dy
        if 0 <= nx < n and 0 <= ny < n:
            b = [list(row) for row in board]
            b[x][y], b[nx][ny] = b[nx][ny], b[x][y]
            res.append(tuple(tuple(row) for row in b))
    return res

def flatten(board):
    return [x for row in board for x in row]

def inversion_count(seq):
    arr = [x for x in seq if x != 0]
    inv = 0
    for i in range(len(arr)):
        for j in range(i+1, len(arr)):
            if arr[i] > arr[j]:
                inv += 1
    return inv

def blank_row_from_bottom(board):
    n = len(board)
    for i in range(n):
```

```
        for j in range(n):
            if board[i][j] == 0:
                return n - i
    raise ValueError("Board does not contain a blank tile (0)")

def is_solvable(start, goal):
    """General n-puzzle solvability test (odd/even width)."""
    n = len(start)
    start_flat = flatten(start)
    goal_flat = flatten(goal)

    pos = {val: idx for idx, val in enumerate(goal_flat)}
    start_perm = [pos[val] for val in start_flat]

    inv = inversion_count(start_perm)

    if n % 2 == 1:
        # odd grid: inversions parity must be even
        return inv % 2 == 0
    else:
        # even grid: blank row from bottom parity matters
        blank_row = blank_row_from_bottom(start)
        goal_blank_row = blank_row_from_bottom(goal)
        # When using relative permutation to goal, parity of blank rows must match
        return (inv + blank_row) % 2 == (0 + goal_blank_row) % 2

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

def a_star_misplaced(start, goal):
    start = tuple(tuple(row) for row in start)
    goal = tuple(tuple(row) for row in goal)

    if len(start) != len(start[0]) or len(goal) != len(goal[0]) or len(start) != len(goal):
        raise ValueError("Initial and goal must be square boards of the same size.")

    start_vals = sorted(flatten(start))
    goal_vals = sorted(flatten(goal))
    if start_vals != goal_vals:
        raise ValueError("Initial and goal must contain the same set of tiles.")

    if not is_solvable(start, goal):
        return None, None, 0, 0  # unsolvable

    counter = count()  # tie-breaker

    h0 = misplaced_heuristic(start, goal)
```

```
    g_score = {start: 0}
    f0 = h0

    open_heap = [(f0, next(counter), start)]
    open_set = {start: f0}
    closed = set()
    came_from = {}

    expansions = 0

    while open_heap:
        _, _, current = heapq.heappop(open_heap)
        if current in closed:
            continue
        closed.add(current)

        if current == goal:
            path = reconstruct_path(came_from, current)
            return path, g_score[current], expansions, len(closed)

        expansions += 1

        for nb in neighbors(current):
            tentative_g = g_score[current] + 1
            if nb in closed:
                continue
            if nb not in g_score or tentative_g < g_score[nb]:
                came_from[nb] = current
                g_score[nb] = tentative_g
                h = misplaced_heuristic(nb, goal)
                f = tentative_g + h
                if nb not in open_set or f < open_set[nb]:
                    heapq.heappush(open_heap, (f, next(counter), nb))
                    open_set[nb] = f

    return None, None, expansions, len(closed)

def read_board(n, prompt):
    print(prompt)
    board = []
    for i in range(n):
        row = list(map(int, input().split()))
        if len(row) != n:
            raise ValueError(f"Row {i+1} must contain exactly {n} integers.")
        board.append(row)
    return board

def print_board(board):
    for row in board:
        print(" ".join(f"{x}" for x in row))

def main():
```

```
    try:
        n = int(input("Enter puzzle size n (e.g., 3 for 3x3): ").strip())
        initial = read_board(n, "Enter initial state row by row (use 0 for blank):")
        goal = read_board(n, "Enter goal state row by row (use 0 for blank):")

        result = a_star_misplaced(initial, goal)
        path, cost, expansions, explored = result

        if path is None:
            print("No solution (unsolvable with given start/goal).")
            return

        print("\nSolution path (each state shows g, h, f):\n")
        for idx, state in enumerate(path):
            g = idx  # each step costs 1
            h = misplaced_heuristic(state, tuple(tuple(r) for r in goal))
            f = g + h
            print(f"Step {idx}: g={g}, h={h}, f={f}")
            print_board(state)
            print()

        print(f"Total cost (number of moves): {cost}")
        print(f"Nodes expanded: {expansions}")
        print(f"Nodes explored (unique): {explored}")
        print("Sharada Koundinya,1BM23CS310")

    except Exception as e:
        print("Error:", e)

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

Output:

```
Enter puzzle size n (e.g., 3 for 3x3): 3
Enter initial state row by row (use 0 for blank):
2 8 3
1 6 4
7 0 5
Enter goal state row by row (use 0 for blank):
1 2 3
8 0 4
7 6 5

Solution path (each state shows g, h, f):

Step 0: g=0, h=4, f=4
2 8 3
1 6 4
7 0 5

Step 1: g=1, h=3, f=4
2 8 3
1 0 4
7 6 5

Step 2: g=2, h=3, f=5
2 0 3
1 8 4
7 6 5

Step 3: g=3, h=2, f=5
0 2 3
1 8 4
7 6 5

Step 4: g=4, h=1, f=5
1 2 3
0 8 4
7 6 5

Step 5: g=5, h=0, f=5
1 2 3
8 0 4
7 6 5

Total cost (number of moves): 5
Nodes expanded: 6
Nodes explored (unique): 7
Sharada Koundinya,1BM23CS310
```

Code:

```
#MANHATTAN DISTANCE
import heapq
from itertools import count

def misplaced_heuristic(board, goal):
    misplaced = 0
    n = len(board)
    for i in range(n):
        for j in range(n):
            if board[i][j] != 0 and board[i][j] != goal[i][j]:
                misplaced += 1
    return misplaced

def manhattan_heuristic(board, goal):
    n = len(board)
    # Map goal positions for each tile
    goal_pos = {}
    for i in range(n):
        for j in range(n):
            goal_pos[goal[i][j]] = (i, j)

    dist = 0
    for i in range(n):
        for j in range(n):
            val = board[i][j]
            if val != 0:
                gi, gj = goal_pos[val]
                dist += abs(i - gi) + abs(j - gj)
    return dist

def find_blank(board):
    n = len(board)
    for i in range(n):
        for j in range(n):
            if board[i][j] == 0:
                return i, j
    raise ValueError("Board does not contain a blank tile (0)")

def neighbors(board):
    n = len(board)
    x, y = find_blank(board)
    dirs = [(0,1),(0,-1),(1,0),(-1,0)]
    res = []
    for dx, dy in dirs:
        nx, ny = x + dx, y + dy
        if 0 <= nx < n and 0 <= ny < n:
            b = [list(row) for row in board]
            b[x][y], b[nx][ny] = b[nx][ny], b[x][y]
            res.append(tuple(tuple(row) for row in b))
    return res
```

```
def flatten(board):
    return [x for row in board for x in row]

def inversion_count(seq):
    arr = [x for x in seq if x != 0]
    inv = 0
    for i in range(len(arr)):
        for j in range(i+1, len(arr)):
            if arr[i] > arr[j]:
                inv += 1
    return inv

def blank_row_from_bottom(board):
    n = len(board)
    for i in range(n):
        for j in range(n):
            if board[i][j] == 0:
                return n - i
    raise ValueError("Board does not contain a blank tile (0)")

def is_solvable(start, goal):
    n = len(start)
    start_flat = flatten(start)
    goal_flat = flatten(goal)

    pos = {val: idx for idx, val in enumerate(goal_flat)}
    start_perm = [pos[val] for val in start_flat]

    inv = inversion_count(start_perm)

    if n % 2 == 1:
        return inv % 2 == 0
    else:
        blank_row = blank_row_from_bottom(start)
        goal_blank_row = blank_row_from_bottom(goal)
        return (inv + blank_row) % 2 == (0 + goal_blank_row) % 2

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

def a_star_manhattan(start, goal):
    start = tuple(tuple(row) for row in start)
    goal = tuple(tuple(row) for row in goal)

    if len(start) != len(start[0]) or len(goal) != len(goal[0]) or len(start) != len(goal):
        raise ValueError("Initial and goal must be square boards of the same size.")
```

```
    start_vals = sorted(flatten(start))
    goal_vals = sorted(flatten(goal))
    if start_vals != goal_vals:
        raise ValueError("Initial and goal must contain the same set of tiles.")

    if not is_solvable(start, goal):
        return None, None, 0, 0

    counter = count()
    h0 = manhattan_heuristic(start, goal)
    g_score = {start: 0}
    f0 = h0

    open_heap = [(f0, next(counter), start)]
    open_set = {start: f0}
    closed = set()
    came_from = {}

    expansions = 0

    while open_heap:
        _, _, current = heapq.heappop(open_heap)
        if current in closed:
            continue
        closed.add(current)

        if current == goal:
            path = reconstruct_path(came_from, current)
            return path, g_score[current], expansions, len(closed)

        expansions += 1

        for nb in neighbors(current):
            tentative_g = g_score[current] + 1
            if nb in closed:
                continue
            if nb not in g_score or tentative_g < g_score[nb]:
                came_from[nb] = current
                g_score[nb] = tentative_g
                h = manhattan_heuristic(nb, goal)
                f = tentative_g + h
                if nb not in open_set or f < open_set[nb]:
                    heapq.heappush(open_heap, (f, next(counter), nb))
                    open_set[nb] = f

    return None, None, expansions, len(closed)

def read_board(n, prompt):
    print(prompt)
    board = []
    for i in range(n):
        row = list(map(int, input().split()))
```

```
        if len(row) != n:
            raise ValueError(f"Row {i+1} must contain exactly {n} integers.")
        board.append(row)
    return board

def print_board(board):
    for row in board:
        print(" ".join(f"{x}" for x in row))

def main():
    try:
        n = int(input("Enter puzzle size n (e.g., 3 for 3x3): ").strip())
        initial = read_board(n, "Enter initial state row by row (use 0 for blank):")
        goal = read_board(n, "Enter goal state row by row (use 0 for blank):")

        result = a_star_manhattan(initial, goal)
        path, cost, expansions, explored = result

        if path is None:
            print("No solution (unsolvable with given start/goal).")
            return

        print("\nSolution path (each state shows g, h, f):\n")
        for idx, state in enumerate(path):
            g = idx
            h = manhattan_heuristic(state, tuple(tuple(r) for r in goal))
            f = g + h
            print(f"Step {idx}: g={g}, h={h}, f={f}")
            print_board(state)
            print()

        print(f"Total cost (number of moves): {cost}")
        print(f"Nodes expanded: {expansions}")
        print(f"Nodes explored (unique): {explored}")
        print("Sharada Koundinya,1BM23CS310")

    except Exception as e:
        print("Error:", e)

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

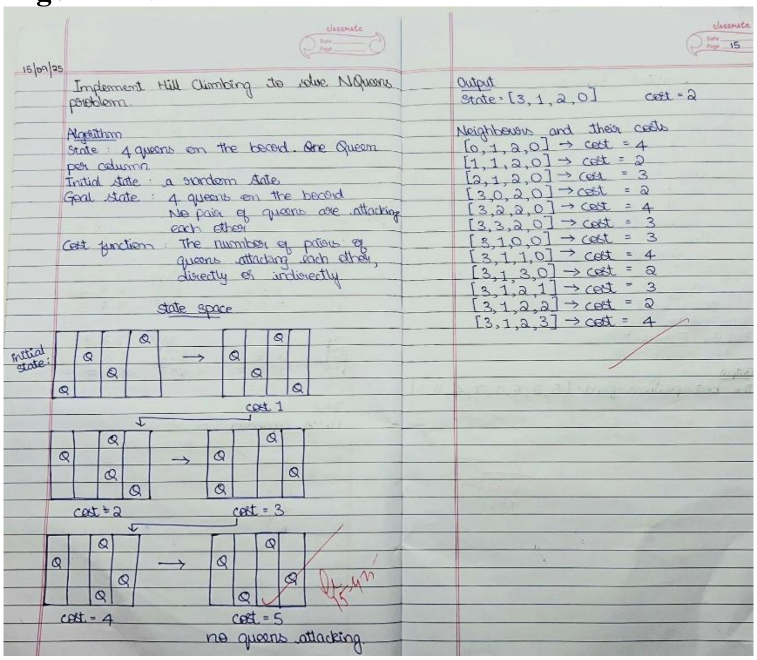
Output:

```
Enter puzzle size n (e.g., 3 for 3x3): 3
Enter initial state row by row (use 0 for blank):
2 8 3
1 6 4
7 0 5
Enter goal state row by row (use 0 for blank):
1 2 3
8 0 4
7 6 5

Solution path (each state shows g, h, f):

Step 0: g=0, h=5, f=5
2 8 3
1 6 4
7 0 5

Step 1: g=1, h=4, f=5
2 8 3
1 0 4
7 6 5

Step 2: g=2, h=3, f=5
2 0 3
1 8 4
7 6 5

Step 3: g=3, h=2, f=5
0 2 3
1 8 4
7 6 5

Step 4: g=4, h=1, f=5
1 2 3
0 8 4
7 6 5

Step 5: g=5, h=0, f=5
1 2 3
8 0 4
7 6 5

Total cost (number of moves): 5
Nodes expanded: 5
Nodes explored (unique): 6
Sharada Koundinya,1BM23CS310
```

Program4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



15/09/25

Implement Hill Climbing to solve NQueens problem.

Algorithm

State : 4 queens on the board. One Queen per column

Initial state : a random state

Goal state : 4 queens on the board
No pair of queens are attacking each other

Cost function : The number of pairs of queens attacking each other, directly or indirectly

state space

initial state:

cost 1

cost = 2

cost = 3

cost = 4

cost = 5

no queens attacking

Page 15

Output

State = [3, 1, 2, 0] cost = 2

Neighbours and their costs

[0,1,2,0] → cost = 4
[1,1,2,0] → cost = 2
[2,1,2,0] → cost = 3
[3,0,2,0] → cost = 2
[3,2,2,0] → cost = 4
[3,3,2,0] → cost = 3
[3,1,0,0] → cost = 3
[3,1,1,0] → cost = 4
[3,1,3,0] → cost = 2
[3,1,2,1] → cost = 3
[3,1,2,2] → cost = 2
[3,1,2,3] → cost = 4

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

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:
            break

        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

initial_state = [3, 1, 2, 0]

hill_climbing(initial_state)

print("SHARADA KOUNDINYA - 1BM23CS310")
```

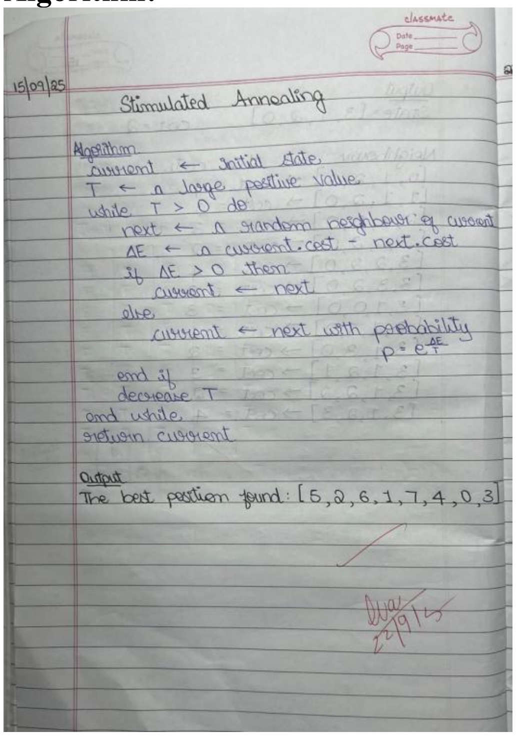
Output:

```
Week 7
Step 0: State = [3, 1, 2, 0], Cost = 2

Neighbors and their costs:
   [0, 1, 2, 0] -> Cost = 4
   [1, 1, 2, 0] -> Cost = 2
   [2, 1, 2, 0] -> Cost = 3
   [3, 0, 2, 0] -> Cost = 2
   [3, 2, 2, 0] -> Cost = 4
   [3, 3, 2, 0] -> Cost = 3
   [3, 1, 0, 0] -> Cost = 3
   [3, 1, 1, 0] -> Cost = 4
   [3, 1, 3, 0] -> Cost = 2
   [3, 1, 2, 1] -> Cost = 3
   [3, 1, 2, 2] -> Cost = 2
   [3, 1, 2, 3] -> Cost = 4
SHARADA KOUNDINYA - 1BM23CS310
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



15/09/25

Stimulated Annealing

Algorithm

current ← initial state

T ← a large positive value

while T > 0 do

next ← a random neighbour of current

ΔE ← a current.cost - next.cost

if ΔE > 0 then

current ← next

else

current ← next with probability $p = e^{\frac{\Delta E}{T}}$

end if

decrease T

end while

return current

Output

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

Code:

```
import random
import math

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

def get_random_neighbor(state):
    n = len(state)
    new_state = list(state)
    col = random.randint(0, n - 1)
    row = random.randint(0, n - 1)
    new_state[col] = row
    return new_state

def simulated_annealing(n=8, max_iterations=10000, initial_temp=100.0, cooling_rate=0.99):

    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

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

        if delta < 0 or random.random() < math.exp(-delta / temperature):
            current, current_cost = neighbor, neighbor_cost

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

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

    return best, best_cost

best_state, best_cost = simulated_annealing()

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

print("Sharada Koundinya - 1BM23CS310")
```

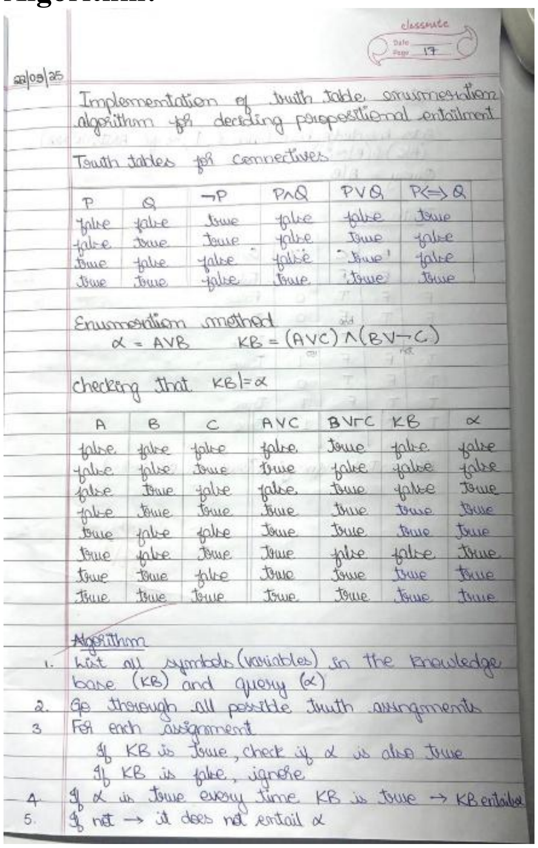
Output:

```
The best position found: [5, 2, 6, 1, 7, 4, 0, 3]
cost = 0
Sharada Koundinya - 1BM23CS310
```

Program 6

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

Algorithm:



22/09/25

Implementation of truth table enumeration algorithm for deciding propositional entailment

Truth tables for connectives

P	Q	¬P	P∧Q	P∨Q	P⇔Q
false	false	true	false	false	true
false	true	true	false	true	false
true	false	false	false	true	false
true	true	false	true	true	true

Enumeration method

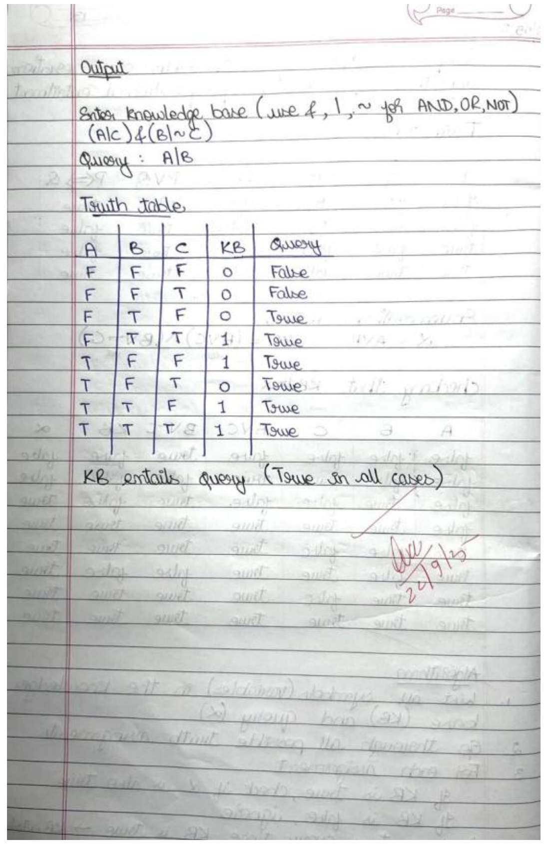
$\alpha = A \vee B \quad KB = (A \vee C) \wedge (B \vee \neg C)$

checking that $KB \models \alpha$

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

Algorithm

1. List all symbols (variables) in the knowledge base (KB) and query (α)
2. Go through all possible truth assignments
3. For each assignment
   If KB is true, check if α is also true
   If KB is false, ignore
4. If α is true every time KB is true → KB entails
5. If not → it does not entail α



Output

Enter knowledge base (use &, |, ~ for AND, OR, NOT)
(A|C)&(B|~C)
Query : A|B

Truth table

A	B	C	KB	Query
F	F	F	0	False
F	F	T	0	False
F	T	F	0	True
F	T	T	1	True
T	F	F	1	True
T	F	T	0	True
T	T	F	1	True
T	T	T	1	True

KB entails query (True in all cases)

22/9/25

Code:

```
import itertools

def eval_expr(expr, model):
    try:
        return eval(expr, {}, model)
    except:
        return False

def tt_entails(KB, query):
    symbols = sorted(set([ch for ch in KB + query if ch.isalpha()]))

    print("\nTruth Table:")
    print(" | ".join(symbols) + " | KB | Query")
    print("-" * (6 * len(symbols) + 20))

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

```
        kb_val = eval_expr(KB, model)
        query_val = eval_expr(query, model)

        row = " | ".join(["T" if model[s] else "F" for s in symbols])
        print(f"{row} |  {kb_val}  |  {query_val}")

        if kb_val and not query_val:
            entails = False

    return entails

KB = input("Enter Knowledge Base (use &, |, ~ for AND, OR, NOT): ")
query = input("Enter Query: ")

result = tt_entails(KB, query)

print("\nResult:")
if result:
    print("KB entails Query (True in all cases).")
else:
    print("KB does NOT entail Query.")

print("Sharada Koundinya - 1BM23CS310")
```

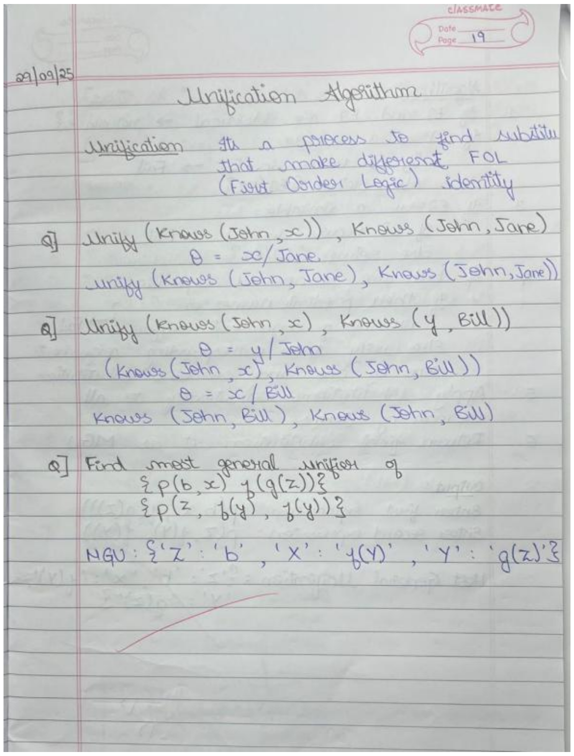
Output:

```
Enter Knowledge Base (use &, |, ~ for AND, OR, NOT): (A|C)&(B|~C)
Enter Query:  A|B

Truth Table:
A | B | C | KB | Query
--------------------------------------
F | F | F |  0  |  False
F | F | T |  0  |  False
F | T | F |  0  |  True
F | T | T |  1  |  True
T | F | F |  1  |  True
T | F | T |  0  |  True
T | T | F |  1  |  True
T | T | T |  1  |  True

Result:
KB entails Query (True in all cases).
Sharada Koundinya - 1BM23CS310
```

Program 7

Implement unification in first order logic

Algorithm:



Page 19

29/09/25

Unification Algorithm

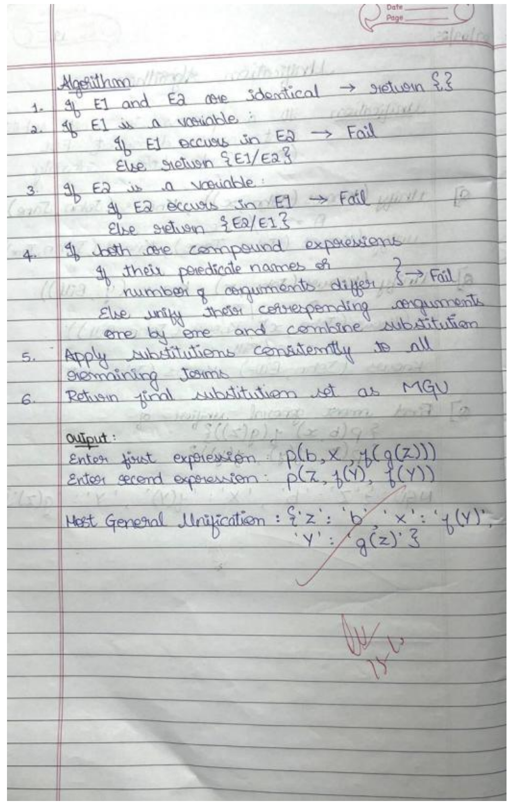
Unification: Its a process to find substitu that make different FOL (First Order Logic) identity

Q] Unify (Knows (John, x)), Knows (John, Jane)
θ = x/Jane
unify (Knows (John, Jane), Knows (John, Jane))

Q] Unify (Knows (John, x), Knows (y, Bill))
θ = y/John
(Knows (John, x), Knows (John, Bill))
θ = x/Bill
Knows (John, Bill), Knows (John, Bill)

Q] Find most general unifier of
{p(b, x) f(g(z))}
{p(z, f(y), f(y))}

MGU: {'Z': 'b', 'X': 'f(Y)', 'Y': 'g(Z)'}



Algorithm
1. If E1 and E2 are identical → return {}
2. If E1 is a variable:
If E1 occurs in E2 → Fail
Else return {E1/E2}
3. If E2 is a variable:
If E2 occurs in E1 → Fail
Else return {E2/E1}
4. If both are compound expressions
If their predicate names or number of arguments differ → Fail
Else unify their corresponding arguments one by one and combine substitution
5. Apply substitutions consistently to all remaining terms
6. Return final substitution set as MGU

Output:
Enter first expression: p(b, x, f(g(z)))
Enter second expression: p(z, f(Y), f(Y))

Most General Unification: {'z': 'b', 'x': 'f(Y)', 'Y': 'g(z)'}

Code:

```
def occurs_check(var, term, subst):
    if var == term:
        return True
    elif isinstance(term, tuple):
        return any(occurs_check(var, t, subst) for t in term)
    elif term in subst:
        return occurs_check(var, subst[term], subst)
    return False

def unify(x, y, subst):
    if subst is None:
        return None
    elif x == y:
        return subst
    elif isinstance(x, str) and x.isupper():
        return unify_var(x, y, subst)
    elif isinstance(y, str) and y.isupper():
        return unify_var(y, x, subst)
    elif isinstance(x, tuple) and isinstance(y, tuple):
        if x[0] != y[0] or len(x) != len(y):
            return None
```

```
        for a, b in zip(x[1:], y[1:]):
            subst = unify(a, b, subst)
            if subst is None:
                return None
        return subst
    else:
        return None

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

def parse_expr(s):
    s = s.replace(" ", "")
    if '(' not in s:
        return s
    name_end = s.index('(')
    name = s[:name_end]
    args = []
    depth = 0
    current = ""
    for c in s[name_end+1:-1]:
        if c == ',' and depth == 0:
            args.append(parse_expr(current))
            current = ""
        else:
            if c == '(':
                depth += 1
            elif c == ')':
                depth -= 1
            current += c
    if current:
        args.append(parse_expr(current))
    return tuple([name] + args)

def expr_to_str(expr):
    if isinstance(expr, tuple):
        return expr[0] + "(" + ",".join(expr_to_str(e) for e in expr[1:]) + ")"
    else:
        return expr

expr1_input = input("Enter first expression: ")
expr2_input = input("Enter second expression: ")

expr1 = parse_expr(expr1_input)
```

```
expr2 = parse_expr(expr2_input)

subst = unify(expr1, expr2, {})

if subst:
    formatted_subst = {var: expr_to_str(val) for var, val in subst.items()}
else:
    formatted_subst = None

print("Most General Unifier (MGU):", formatted_subst)
print("Sharada Koundinya,1BM23CS310")
```

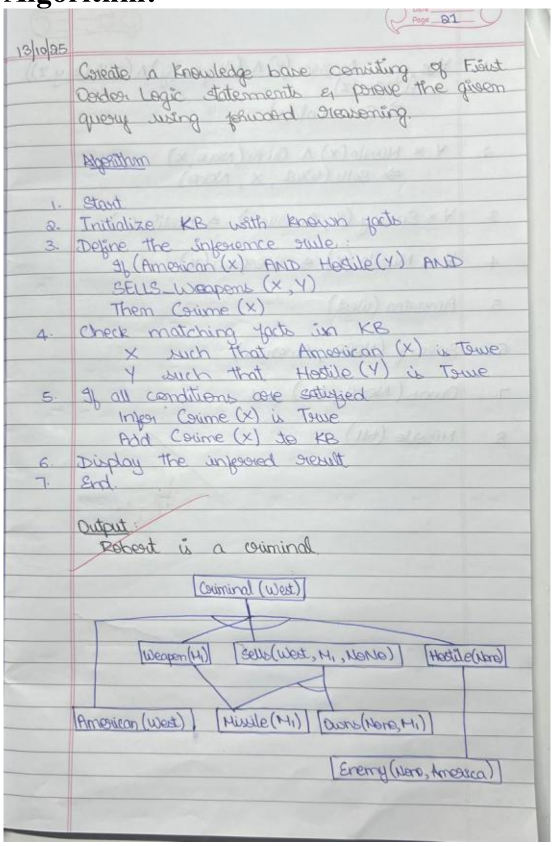
Output:

```
Enter first expression: p(b,X,f(g(Z)))
Enter second expression: p(Z,f(Y),f(Y))
Most General Unifier (MGU): {'Z': 'b', 'X': 'f(Y)', 'Y': 'g(Z)'}
Sharada Koundinya,1BM23CS310
```

Program 8

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

Algorithm:


Page 81

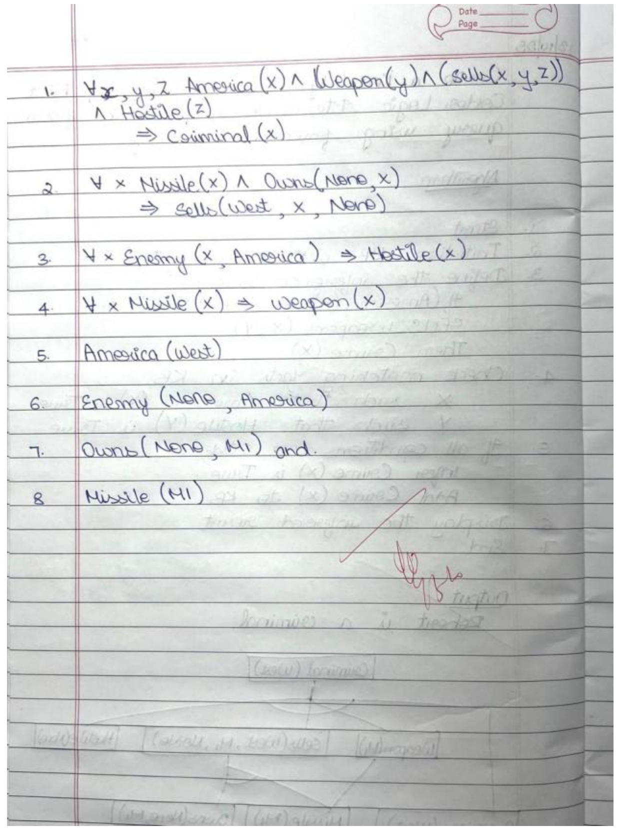
13/10/25

Create a Knowledge base consisting of First Order Logic statements & prove the given query using forward reasoning.

Algorithm

1. Start
2. Initialize KB with known facts
3. Define the inference rule:
   If (American(X) AND Hostile(Y) AND SELLS_Weapons(X, Y)
   Then Crime(X)
4. Check matching facts in KB
   X such that American(X) is True
   Y such that Hostile(Y) is True
5. If all conditions are satisfied
   Infer Crime(X) is True
   Add Crime(X) to KB
6. Display the inferred result
7. End

Output:
Robert is a criminal


1. $\forall x, y, z$ America(x) $\land$ Weapon(y) $\land$ (Sells(x, y, z)) $\land$ Hostile(z) $\Rightarrow$ Criminal(x)
2. $\forall x$ Missile(x) $\land$ Owns(Nono, x) $\Rightarrow$ Sells(West, x, Nono)
3. $\forall x$ Enemy(x, America) $\Rightarrow$ Hostile(x)
4. $\forall x$ Missile(x) $\Rightarrow$ weapon(x)
5. America(West)
6. Enemy(Nono, America)
7. Owns(Nono, M1) and
8. Missile(M1)

Code:

```
facts = {
   'American(Robert)': True,
   'Hostile(A)': True,
   'Sells_Weapons(Robert, A)': True
}

If American(X) and Hostile(Y) and Sells_Weapons(X, Y), then Crime(X)
def forward_reasoning(facts):

 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

forward_reasoning(facts)

if facts.get('Crime(Robert)', False):
   print("Robert is a criminal.")
```

```
else:
    print("Robert is not a criminal.")

print("Sharada Koundinya 1BM23CS310")
```

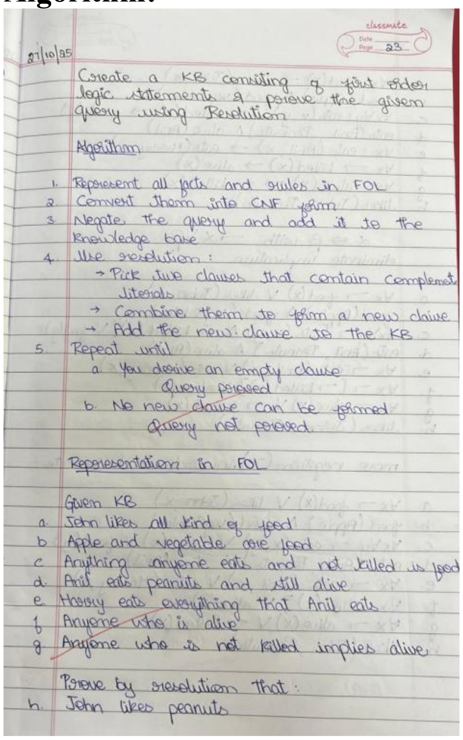
Output:

Robert is a criminal.
Sharada Koundinya 1BM23CS310

Program 9

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

Algorithm:



27/10/25

Date Page 23

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

Algorithm

1. Represent all facts and rules in FOL
2. Convert them into CNF form
3. Negate the query and add it to the knowledge base
4. Use resolution :
   - → Pick two clauses that contain complement literals
   - → Combine them to form a new claue
   - → Add the new clause to the KB
5. Repeat until
   - a. You derive an empty clause
     Query proved
   - b. No new clause can be formed
     Query not proved

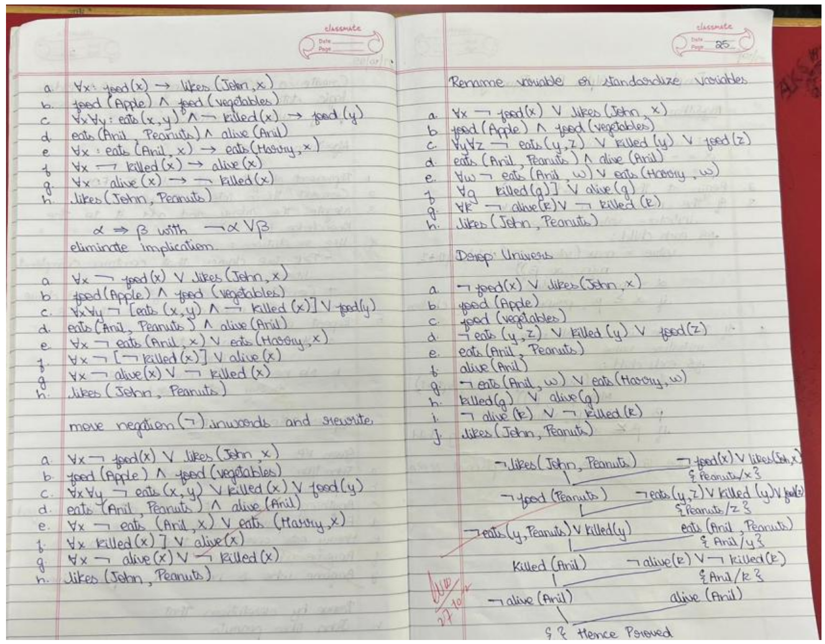
Representation in FOL

Given KB

a. John likes all kind of food
b. Apple and vegetable are food
c. Anything anyone eats and not killed is food
d. Anil eats peanuts and still alive
e. Harry eats everything that Anil eats
f. Anyone who is alive
g. Anyone who is not killed implies alive

Prove by resolution that :

h. John likes peanuts



a. $\forall x : food(x) \rightarrow likes(John, x)$
b. $food(Apple) \land food(vegetables)$
c. $\forall x \forall y : eats(x, y) \land \neg killed(x) \rightarrow food(y)$
d. $eats(Anil, Peanuts) \land alive(Anil)$
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. $likes(John, Peanuts)$

$\alpha \Rightarrow \beta$ with $\neg\alpha \lor \beta$

eliminate implication

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

move negation ($\neg$) inwards and rewrite

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

Date Page 25

Rename variable or standardize variables

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

Drop Univers

a. $\neg food(x) \lor likes(John, x)$
b. $food(Apple)$
c. $food(vegetables)$
d. $\neg eats(y, z) \lor killed(y) \lor food(z)$
e. $eats(Anil, Peanuts)$
f. $alive(Anil)$
g. $\neg eats(Anil, w) \lor eats(Harry, w)$
h. $killed(g) \lor alive(g)$
i. $\neg alive(k) \lor \neg killed(k)$
j. $likes(John, Peanuts)$

$\neg likes(John, Peanuts)$ | $\neg food(x) \lor likes(John, x)$ $\{Peanuts/x\}$

$\neg food(Peanuts)$ | $\neg eats(y, z) \lor killed(y) \lor food(z)$ $\{Peanuts/z\}$

$\neg eats(y, Peanuts) \lor killed(y)$ | $eats(Anil, Peanuts)$ $\{Anil/y\}$

$Killed(Anil)$ | $\neg alive(k) \lor \neg killed(k)$ $\{Anil/k\}$

$\neg alive(Anil)$ | $alive(Anil)$

{ } Hence Proved

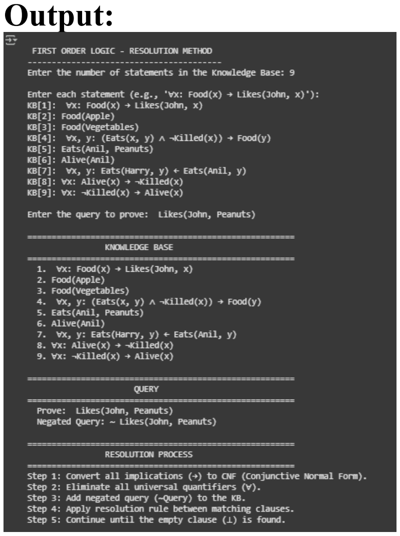
27/10

Code:

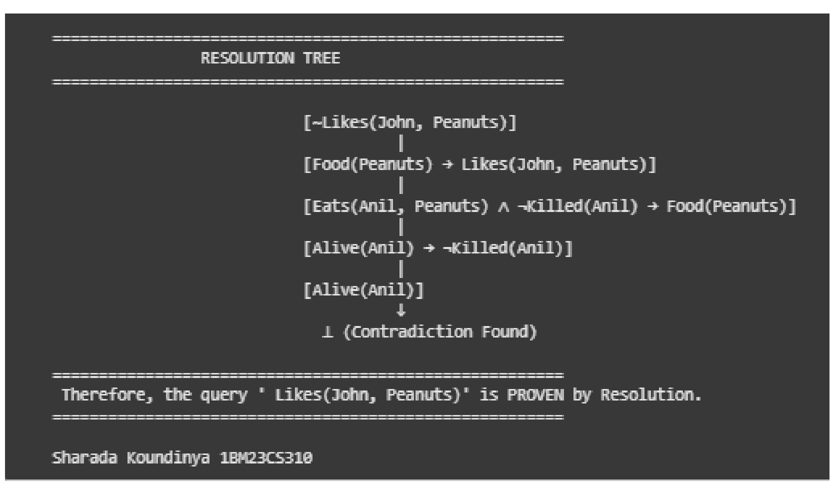
```
def fol_resolution(kb, query):
    print("\n" + "="*55)
    print("                KNOWLEDGE BASE")
    print("="*55)
    for i, clause in enumerate(kb, start=1):
        print(f"  {i}. {clause}")

    print("\n" + "="*55)
    print("                      QUERY")
    print("="*55)
    print(f"  Prove: {query}")
    print(f"  Negated Query: ~{query}\n")

    print("="*55)
    print("                RESOLUTION PROCESS")
    print("="*55)
    print("Step 1: Convert all implications (→) to CNF (Conjunctive Normal Form).")
    print("Step 2: Eliminate all universal quantifiers (∀).")
    print("Step 3: Add negated query (~Query) to the KB.")
    print("Step 4: Apply resolution rule between matching clauses.")
    print("Step 5: Continue until the empty clause (⊥) is found.\n")
```

```
    print("="*55)
    print("              RESOLUTION TREE")
    print("="*55)
    print("""
                        [~Likes(John, Peanuts)]
                                |
                        [Food(Peanuts) → Likes(John, Peanuts)]
                                |
                        [Eats(Anil, Peanuts) ∧ ¬Killed(Anil) → Food(Peanuts)]
                                |
                        [Alive(Anil) → ¬Killed(Anil)]
                                |
                        [Alive(Anil)]
                                ↓
                         ⊥ (Contradiction Found)
    """)

    print("="*55)
    print(f" Therefore, the query '{query}' is PROVEN by Resolution.")
    print("="*55 + "\n")

print("\n FIRST ORDER LOGIC - RESOLUTION METHOD")

n = int(input("Enter the number of statements in the Knowledge Base: "))

kb = []
print("\nEnter each statement (e.g., '∀x: Food(x) → Likes(John, x)'):")
for i in range(n):
    stmt = input(f"KB[{i+1}]: ")
    kb.append(stmt)

query = input("\nEnter the query to prove: ")

fol_resolution(kb, query)
print("Sharada Koundinya 1BM23CS310")
```

Output:



```
 FIRST ORDER LOGIC - RESOLUTION METHOD
----------------------------------------
Enter the number of statements in the Knowledge Base: 9

Enter each statement (e.g., '∀x: Food(x) → Likes(John, x)'):
KB[1]:  ∀x: Food(x) → Likes(John, x)
KB[2]: Food(Apple)
KB[3]: Food(Vegetables)
KB[4]:  ∀x, y: (Eats(x, y) ∧ ¬Killed(x)) → Food(y)
KB[5]: Eats(Anil, Peanuts)
KB[6]: Alive(Anil)
KB[7]:  ∀x, y: Eats(Harry, y) ← Eats(Anil, y)
KB[8]: ∀x: Alive(x) → ¬Killed(x)
KB[9]: ∀x: ¬Killed(x) → Alive(x)

Enter the query to prove:  Likes(John, Peanuts)

=======================================================
                KNOWLEDGE BASE
=======================================================
  1.  ∀x: Food(x) → Likes(John, x)
  2. Food(Apple)
  3. Food(Vegetables)
  4.  ∀x, y: (Eats(x, y) ∧ ¬Killed(x)) → Food(y)
  5. Eats(Anil, Peanuts)
  6. Alive(Anil)
  7.  ∀x, y: Eats(Harry, y) ← Eats(Anil, y)
  8. ∀x: Alive(x) → ¬Killed(x)
  9. ∀x: ¬Killed(x) → Alive(x)

=======================================================
                     QUERY
=======================================================
  Prove:  Likes(John, Peanuts)
  Negated Query: ~ Likes(John, Peanuts)

=======================================================
                RESOLUTION PROCESS
=======================================================
Step 1: Convert all implications (→) to CNF (Conjunctive Normal Form).
Step 2: Eliminate all universal quantifiers (∀).
Step 3: Add negated query (~Query) to the KB.
Step 4: Apply resolution rule between matching clauses.
Step 5: Continue until the empty clause (⊥) is found.
```

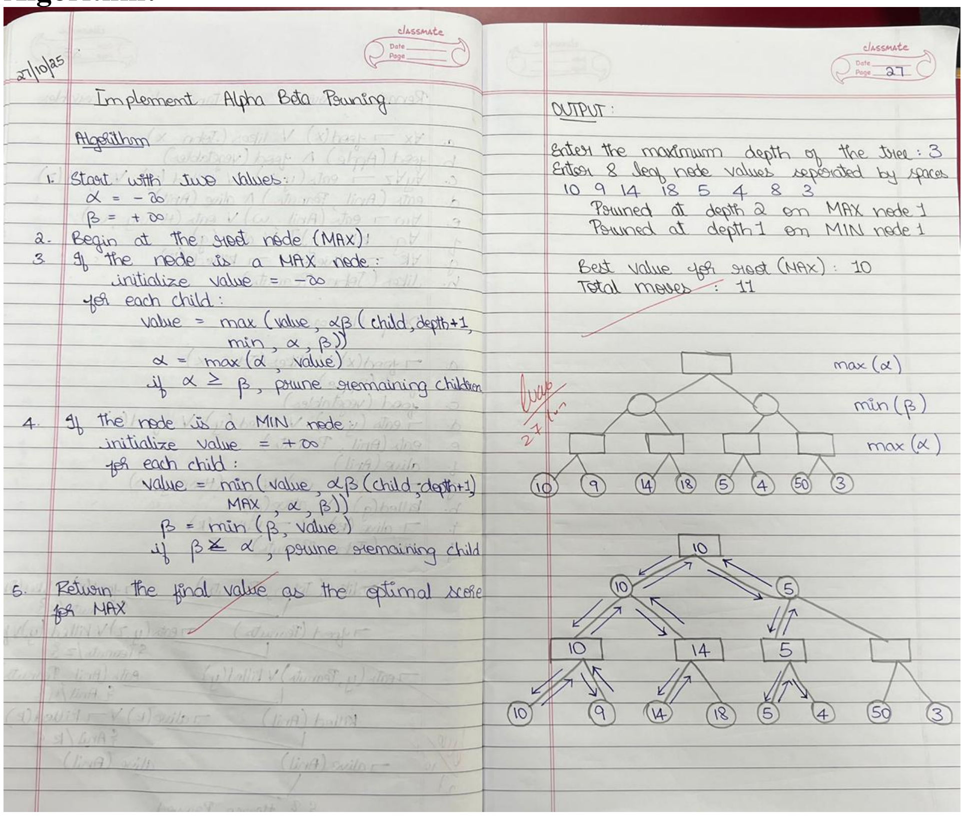


```
=======================================================
              RESOLUTION TREE
=======================================================

                        [~Likes(John, Peanuts)]
                                |
                        [Food(Peanuts) → Likes(John, Peanuts)]
                                |
                        [Eats(Anil, Peanuts) ∧ ¬Killed(Anil) → Food(Peanuts)]
                                |
                        [Alive(Anil) → ¬Killed(Anil)]
                                |
                        [Alive(Anil)]
                                ↓
                         ⊥ (Contradiction Found)

=======================================================
 Therefore, the query ' Likes(John, Peanuts)' is PROVEN by Resolution.
=======================================================

Sharada Koundinya 1BM23CS310
```

Program 10

Implement Alpha-Beta Pruning.

Algorithm:



27/10/25

Implement Alpha Beta Pruning

Algorithm

1. Start with two values:
   α = −∞
   β = +∞
2. Begin at the root node (MAX)
3. If the node is a MAX node:
   initialize value = −∞
   for each child:
   value = max (value, αβ (child, depth+1, min, α, β))
   α = max (α, value)
   if α ≥ β, prune remaining children
4. If the node is a MIN node:
   initialize value = +∞
   for each child:
   value = min (value, αβ (child, depth+1, MAX, α, β))
   β = min (β, value)
   if β ≤ α, prune remaining child
5. Return the final value as the optimal score for MAX

Page 27

OUTPUT:

Enter the maximum depth of the tree: 3
Enter 8 leaf node values separated by spaces
10 9 14 18 5 4 8 3
Pruned at depth 2 on MAX node 1
Pruned at depth 1 on MIN node 1

Best value for root (MAX): 10
Total moves: 11

Code:

```
move_count = 0

def alpha_beta(depth, node_index, is_maximizing, values, alpha, beta, max_depth):
    global move_count
    move_count += 1

    if depth == max_depth:
        return values[node_index]

    if is_maximizing:
        best = float('-inf')
        for i in range(2):  # binary tree
            val = alpha_beta(depth + 1, node_index * 2 + i, False, values, alpha, beta, max_depth)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                print(f" Pruned at depth {depth} on MAX node {node_index}")
                break
        return best
```

```
    else:
        best = float('inf')
        for i in range(2):
            val = alpha_beta(depth + 1, node_index * 2 + i, True, values, alpha, beta, max_depth)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                print(f" Pruned at depth {depth} on MIN node {node_index}")
                break
        return best

max_depth = int(input("Enter the maximum depth of the tree: "))

num_leaves = 2 ** max_depth
print(f"Enter {num_leaves} leaf node values separated by spaces:")
values = list(map(int, input().split()))

if len(values) != num_leaves:
    print(" Error: Number of values does not match 2^depth.")
else:
    move_count = 0
    best_value = alpha_beta(0, 0, True, values, float('-inf'), float('inf'), max_depth)
    print("\n Best value for root (MAX):", best_value)
    print(f" Total moves (nodes visited): {move_count}")
print("Sharada Koundinya 1BM23CS310")
```

Output:

```
Enter the maximum depth of the tree: 3
Enter 8 leaf node values separated by spaces:
10 9 14 18 5 4 8 3
 Pruned at depth 2 on MAX node 1
 Pruned at depth 1 on MIN node 1

 Best value for root (MAX): 10
 Total moves (nodes visited): 11
Sharada Koundinya 1BM23CS310
```