VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Srujan K R(1BM23CS340)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Aug 2025 to Dec 2025

B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Srujan K R (1BM23CS340),** 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.

Lab faculty Incharge Name Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link: https://github.com/Srujan4812/AI_LAB_SRUJAN_K_R

Index

1) 18/8/25	Tic Tac Toe
2) 25/8/25	vaccum cleaner
3) 25/8/25	8-puzzle problem
4) 1/9/25	8-puzzle DFS (non-heuristic
5) 1/9/25	8-puzzle BFS (non heuristic)
6) 1/9/25	8-puzzle heuristic
7) 8/9/25	8-puzzle (heuristic) A* misp tiles, A* manhattan distanc
8. 15/9/25	Hill-climbing Algo & greedy
9. 15/9/25	Simulated Annealing & greedy
10/ 22/9/25	proposional logic
11) 13/10/25	Unification (FOL)
12) 13/10/25	Foward chaining (FOL)
13) 27-10-25	First order logic
14) 27-10-25	Adversarial search

Program 1

Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:
Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Code:
Implement Tic –Tac –Toe Game

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

def check_winner(board, player):
    for row in board:
        if all(s == player for s in row):
            return True
    for col in range(3):
        if all(board[row][col] == player for row in range(3)):
            return True
    if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in range(3)):
        return True
    return False

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

def play_game():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = "X"
    while True:
        print_board(board)
        row, col = map(int, input(f"Player {current_player}, enter row and col (0-2, space separated):
").split())
        if board[row][col] == " ":
            board[row][col] = current_player
            if check_winner(board, current_player):
                print_board(board)
                print(f"Player {current_player} wins!")
```

```
            break
        if is_full(board):
            print_board(board)
            print("It's a draw!")
            break
        current_player = "O" if current_player == "X" else "X"
    else:
        print("Cell already taken, try again.")
```

```
TIC TAC TOE GAME

 | |
 | |
 | |

Player X, enter row and col (0-2, space separated): 0 0
X| |
 | |
 | |

Player O, enter row and col (0-2, space separated): 1 1
X| |
 |O|
 | |

Player X, enter row and col (0-2, space separated): 0 1
X|X|
 |O|
 | |

Player O, enter row and col (0-2, space separated): 2 2
X|X|
 |O|
 | |O

Player X, enter row and col (0-2, space separated): 0 2
X|X|X
 |O|
 | |O

Player X wins!

PROGRAM BY: SRUJAN K R, USN: 1BM23CS340
```

AI - Lab

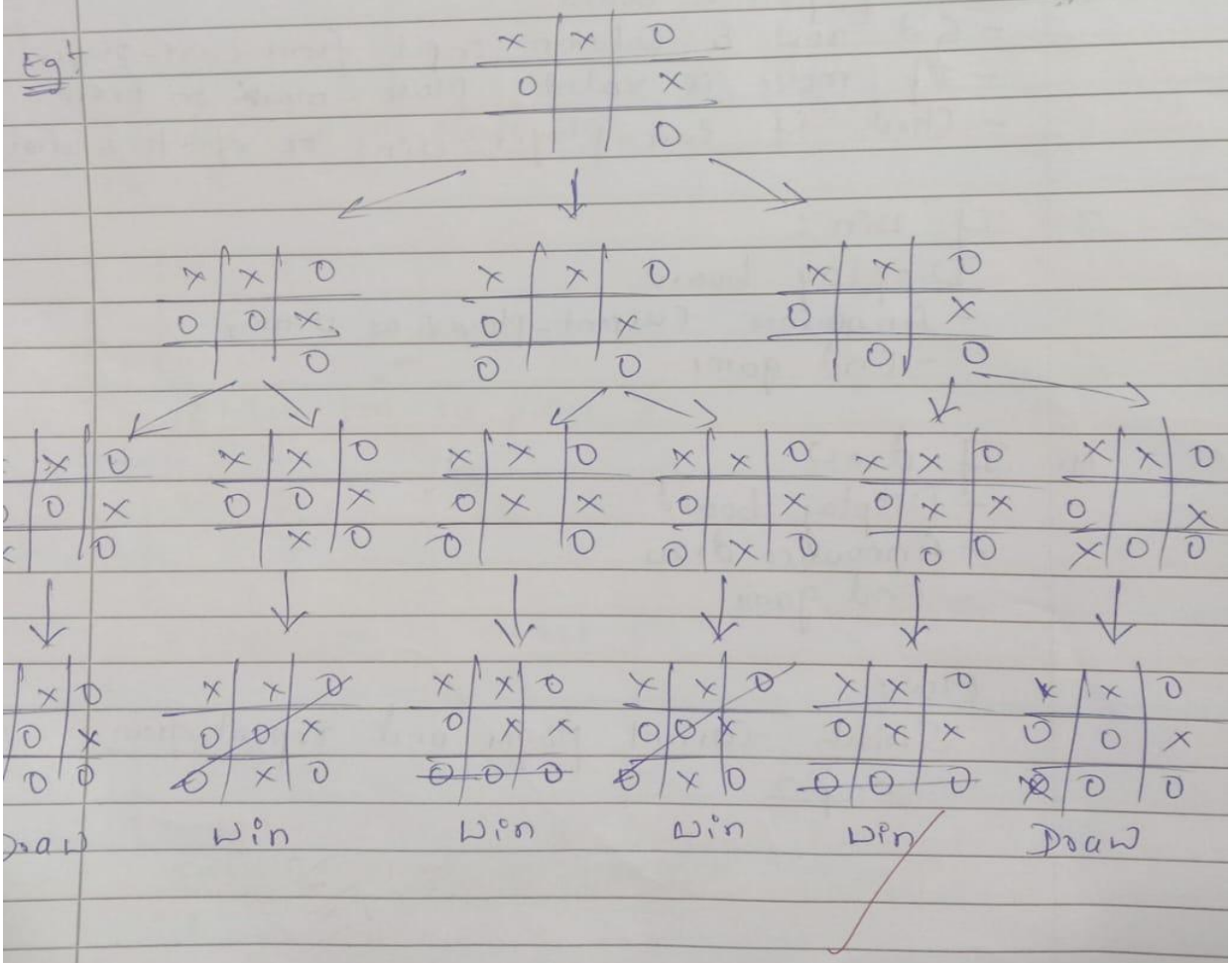
18/02/24

Problem Statement:

Implementation Tic-Tac-Toe:

Implement a two-player Tic Tac toe game in python where players take turns marking spaces in a 3×3 grid 'x' and 'O'.

Eg:



Pseudo code: Algorithm :

START

Initialize 3×3 board with empty spaces
- Set Current_player to 'x'

2. Loop until Win or draw:
- Display the board
- Get row & Column input from curr_player
- If move is valid, place mark on board
- Check if curr_player wins or if its a draw

3. If Win:
Display board
- Announce Current_player as winner
- End game.

4. If draw
- Display board
- Announce draw
- End game

Else:
Switch Current player and repeat from Step 2

Output

Enter pos to place 'X'

```
[ ]
```

↳ 1,1

```
[ X ]
```

Enter pos to place 0:

↳ 1,2

```
[ X  0 ]
```

Enter pos to place X:

↳ 2,2

```
[ X  0 ]
[    X ]
```

Enter pos to place 0

↳ 3,3

```
[ X  0 ]
[    X ]
[      0 ]
```

Enter position to place X:

↳ 3,1

```
[ X  0  X ]
[    X ]
[ 0     0 ]
```

↳ Enter pos to place 0:

```
[ X  0  X ]
[    X ]
[ 0  0  0 ]
```

0 Wins

Game over:

Implement vacuum cleaner agent

```
rooms = {
    '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): ")),
    'C': int(input("Enter state of C (0 for clean, 1 for dirty): ")),
    'D': int(input("Enter state of D (0 for clean, 1 for dirty): "))
}

start = input("Enter starting location (A, B, C, or D): ").upper()
order = ['A', 'B', 'C', 'D']

if start not in order:
    print("Invalid starting location.")
    exit()

start_index = order.index(start)
visited_order = order[start_index:] + order[:start_index]

cost = 0

for i in range(len(visited_order)):
    current = visited_order[i]
    print(f"\nVacuum is in room {current}")

    if rooms[current] == 1:
        print(f"{current} is dirty.")
        print(f"Cleaning {current}...")
        rooms[current] = 0
        cost += 1
    else:
        print(f"{current} is clean.")

    if i < len(visited_order) - 1:
        next_room = visited_order[i + 1]
        print(f"Moving vacuum to {next_room}")
        cost += 1

print(f"\nCost: {cost}")
print(rooms)
```

```
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 0
Enter state of C (0 for clean, 1 for dirty): 1
Enter state of D (0 for clean, 1 for dirty): 1
Enter starting location (A, B, C, or D): B

Vacuum is in room B
B is clean.
Moving vacuum to C

Vacuum is in room C
C is dirty.
Cleaning C...
Moving vacuum to D

Vacuum is in room D
D is dirty.
Cleaning D...
Moving vacuum to A

Vacuum is in room A
A is dirty.
Cleaning A...

Total Cost: 6
Final room states: {'A': 0, 'B': 0, 'C': 0, 'D': 0}

PROGRAM BY: SRUJAN K R, USN: 1BM23CS340
```

Vaccum cleaner Agent

25/09

Implement the vaccum cog. cleaner agent problem in python:

Algorithm

1. Initialize the Matrix (2d)
2. Place the cleaner in a room.
3. While there is dirt in the room:
   a. Check current pos and clean if dirty
   b. move to a valid adjacent cell (avoid walls).
4. Repeat until all reachable dirt is cleaned
5. Stop and print room status.

output

```
Initial States :[1,1]
cleaner in room 1

cleaner is in room 1
Room is dirty
clean? (y/n): y
Room is now clean

Current Room Status [0,1]
Enter the room no -10
move to (1 or 2): 1
```

__/__/__

Cleaner is in room: 1
Room is already

Current room state [0, 1]
Enter room to move to (1 or 2): 2

cleaner is in room: 2
room is dirty
clean (y/n): y
room is now clean

Current room states: [0, 0]
All rooms are clean program ends.

Program 2

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

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

        if current_state in visited:
            continue

        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


# ---------- MAIN PROGRAM ----------
start = input("Enter start state (e.g., 724506831): ")

if len(start) == 9 and set(start) == set('012345678'):
    print("\nStart 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))
        print("\n1BM23CS340 SRUJAN K R\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 or max depth reached.")
else:
    print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")
```

```
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

1BM23CS340 SRUJAN K R

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

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

Initial
Goal:
up
down
left
right

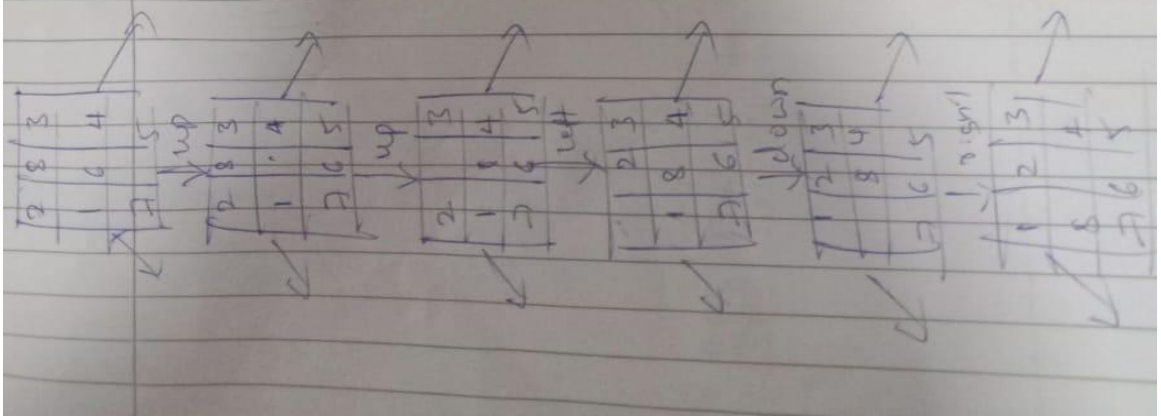
Algorithm BFS

1) Check if puzzle can be solved at all using the ~~inversions~~ invers rule if it can't stop
2. Use waiting list a queue to keep tracks of all possible board arrangements be used to check.
3. Explore, take the first arrangement from the list.
4. If is the arrangemt solved puzzle? If yes solution found.
5. find next moves, find all the new arrangement you can make by blank tile moving
6. Add to list for each new arrangement you haven't seen before add it to the end of the waiting list

=> Explore until you find the solution

01.09

Transition



Output: (2nd)

BFS : DFS

Start from initial state, and compare with goal matrix
In DFS move the space to the up down top left
Then move the space to available space to the left most & check if it matches the solution or else Backtrack to Stored matrix them compare
If found return the matrix or N of iteration

Output

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

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

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

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

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

moves: up → up → Left → Down → right
Total step to goal: 5
Total unique states visited: 35

__/__/____

8 - puzzle problem (non - heuristic using DFS)

Start from initial state Here in BFS
so level by level.
Here it takes all the possible level if found return.
No Backtracking just checking and going level by level as up, down, right, left.
If found return the matrix or No of iteration.

Output

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

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

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

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

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

Moves : Up -> Up -> Left -> Down -> Right
Total steps : 5
Total unique state : 6

Heuristic

Algo

1) Initialize the bound with the heuristic value of the start state.
2. Perform a depth first search (DFS) but only explore paths where the total estimated cost $f = g + h$ is less than or equal to the current bound.
3. If during DFS you find the goal state return the solution immediately.
4. If no solution is found within the current bound, record the smallest f value that exceeded the bound.
5) Repeat DFS
6) Continue iterating until the goal is found or no more states can be expanded.

output

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

Step 5

1 2 3

8 0 4

7 6 5

```
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  # Count visited nodes
```

```
    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]
    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 PROGRAM ----------
start = input("Enter start state (e.g., 724506831): ")

if len(start) == 9 and set(start) == set('012345678'):
    print("\nStart 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("\n1BM23CS340 SRUJAN K R\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.")
```

```
Enter start state (e.g., 724506831): 123405678

Start state:
1 2 3
4   5
6 7 8

Total states visited: 24298
Solution found!
Moves: R D L L U R D R U L L D R R
Number of moves: 14
```

1BM23CS340 SRUJAN K R

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

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

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

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

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

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

Move 7: D

1 2 3
5 6 8
4 7

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

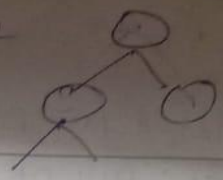
Move 9: U
1 2 3
5 6
4 7 8

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

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

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

IDS



Algo:

1. Start
2. Set depth unit d=0
3. Perform a depth limited dfs with current depth limited
4. Explore the search to depth only if a goal is formed return to path.
5. If no sol is formed at depth d increment by 1.
6. Repeat Setep 3 until solution is formed

Output:

Moves to Goal: UULDR
Total steps: 5
Unique states visited 20

Steps 0:

2 8 3
1 6 4
7 0 5

Step 1 (U):

2 8 3
1 0 4
7 6 5

Step 2 (U):

2	0	3
1	8	4
7	6	5

Step 3 (L)

0	2	3
1	8	4
7	6	5

Step 4 (D):

1	2	3
0	8	4
7	6	5

Step 5 (R):

1	2	3
8	0	4
7	6	5

Space complexity : O(bd)

Program 3

```
import heapq

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 misplaced_tiles(state):
    """Heuristic: count tiles not in goal position (excluding zero)."""
    return sum(1 for i, val in enumerate(state) if val != '0' and val != goal_state[i])

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

    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 + misplaced_tiles(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("\nStart 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("\n1BM23CS340 SRUJAN K R\n")

        current_state = start
        g = 0
        for i, move in enumerate(result, 1):
            new_state = move_tile(current_state, move)
            g += 1
            h = misplaced_tiles(new_state)
            f = g + h
            print(f"Move {i}: {move}")
            print_state(new_state)
            print(f"g(n) = {g}, h(n) = {h}, f(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.")
```

```
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
1BM23CS340 SRUJAN K R

Move 1: U
2 8 3
1   4
7 6 5
g(n) = 1, h(n) = 3, f(n) = 4

Move 2: U
2   3
1 8 4
7 6 5
g(n) = 2, h(n) = 3, f(n) = 5

Move 3: L
2 3
1 8 4
7 6 5
g(n) = 3, h(n) = 2, f(n) = 5

Move 4: D
1 2 3
8 4
7 6 5
g(n) = 4, h(n) = 1, f(n) = 5

Move 5: R
1 2 3
8 4 5
7 6
g(n) = 5, h(n) = 0, f(n) = 5
```

```
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):
```

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

    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("\nStart 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("\n1BM23CS340 SRUJAN K R\n")

        current_state = start
        g = 0
        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) = {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.")
```

```
Total states visited: 21
Solution found!
Moves: L U L D R R U L D R
Number of moves: 10
1BM23CS340 SRUJAN K R

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

Move 2: U
1 2 3
6   8
4 7 5
g(n) = 2, h(n) = 8, f(n) = 10

Move 3: L
1 2 3
  6 8
4 7 5
g(n) = 3, h(n) = 7, f(n) = 10

Move 4: D
1 2 3
4 6 8
7   5
g(n) = 4, h(n) = 6, f(n) = 10

Move 5: R
1 2 3
4 6 8
7 5
g(n) = 5, h(n) = 5, f(n) = 10

Move 6: R
1 2 3
4 6 8
7 5
g(n) = 6, h(n) = 4, f(n) = 10

Move 7: U
1 2 3
4 6
7 5 8
g(n) = 7, h(n) = 3, f(n) = 10

Move 8: L
1 2 3
4 6
7 5 8
g(n) = 8, h(n) = 2, f(n) = 10

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

Move 10: R
1 2 3
4 5 6
7 8
g(n) = 10, h(n) = 0, f(n) = 10
```

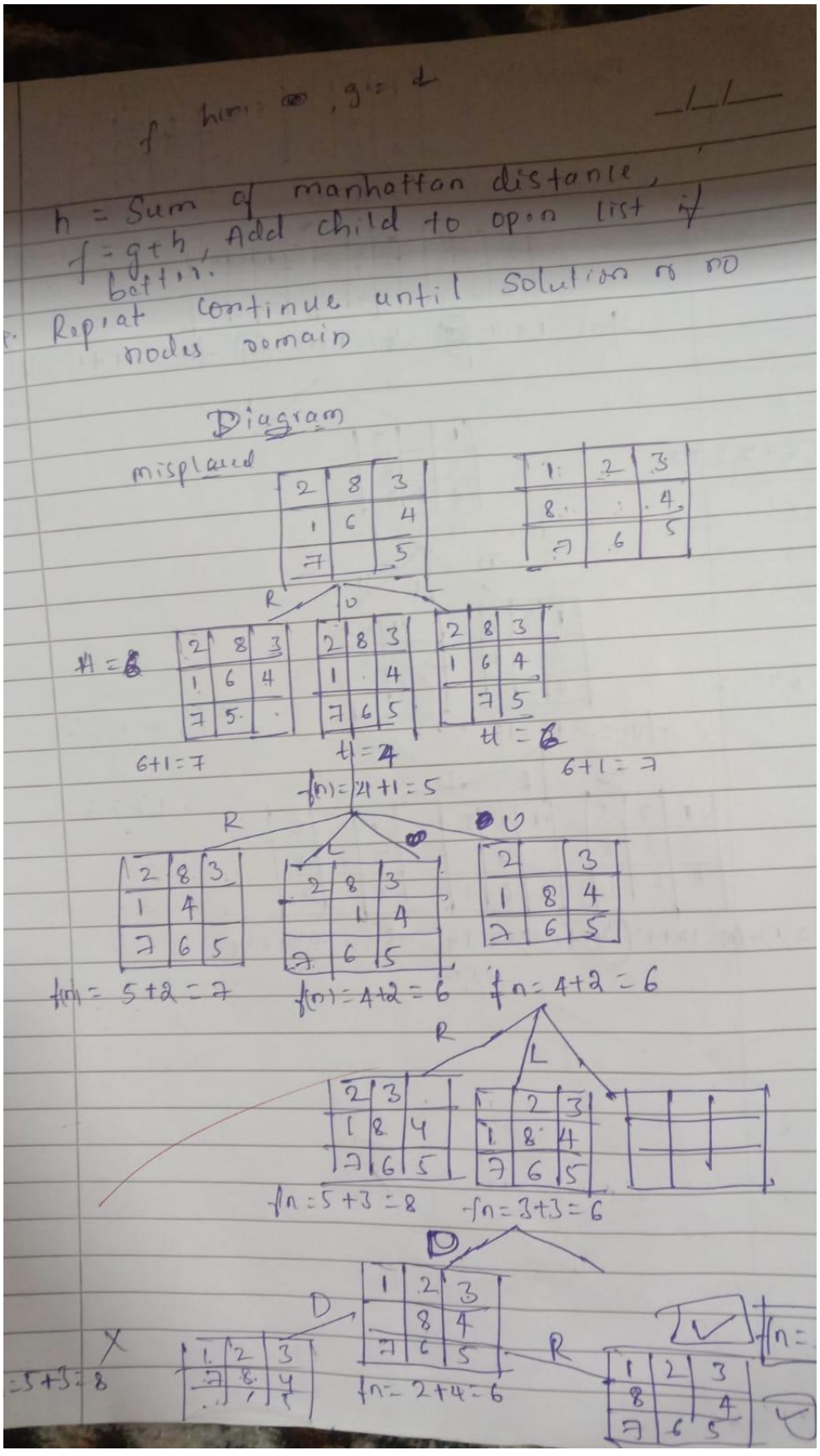
f hon: ∞ , g = 0

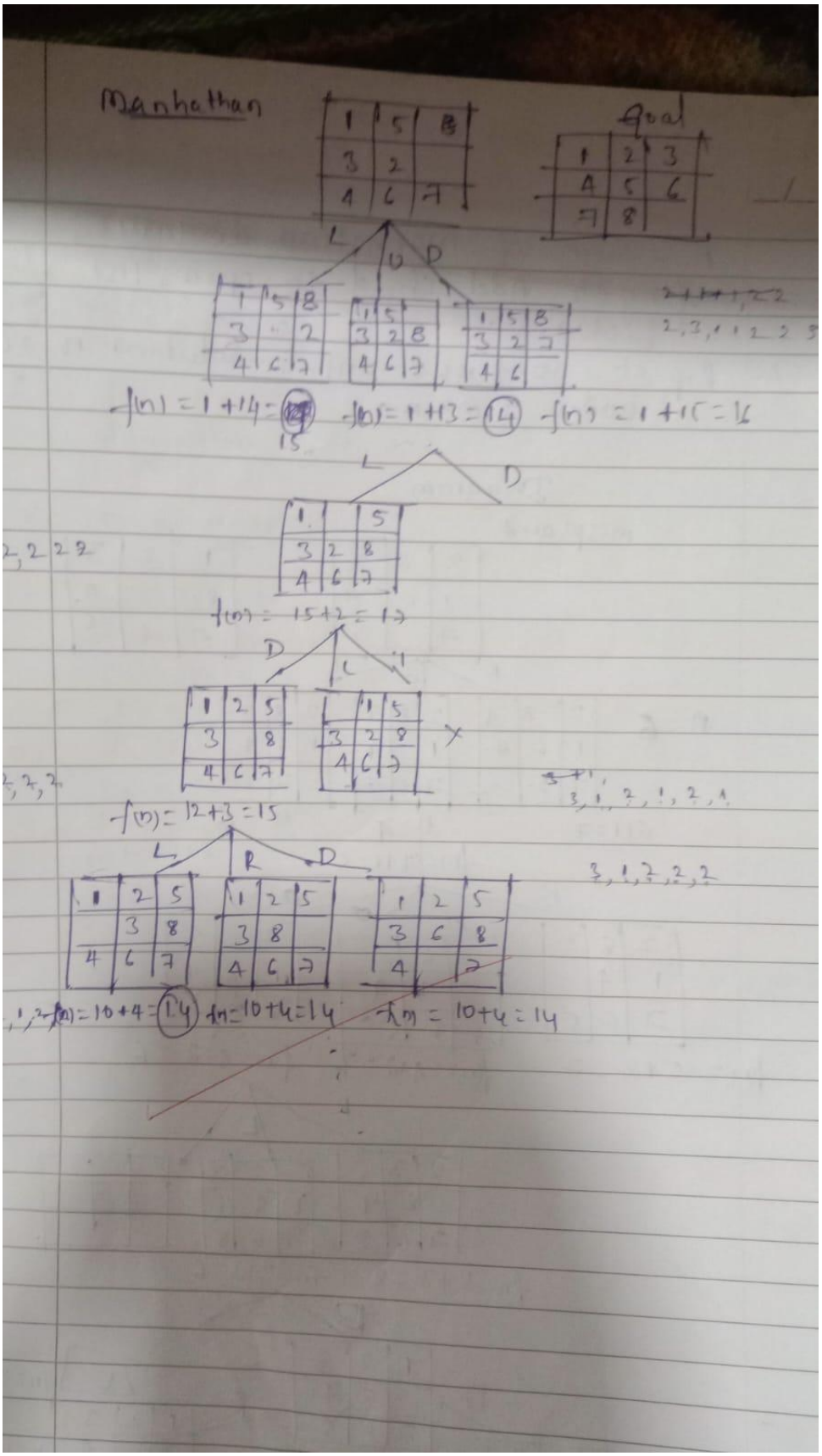
h = Sum of manhattan distance,
f = g + h, Add child to open list if better.

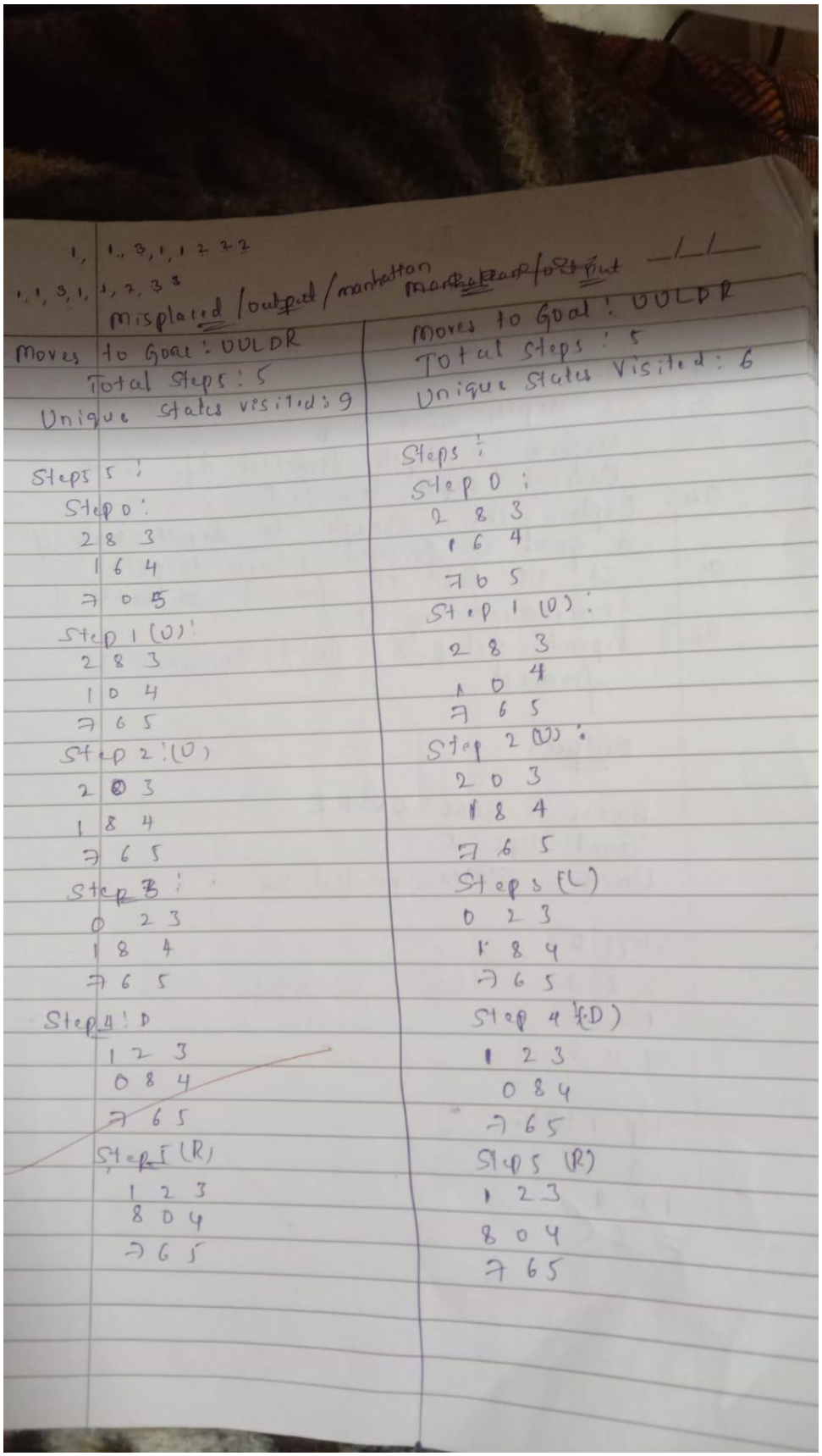
Repeat continue until solution or no nodes remain

Diagram

misplaced




Manhattan
Goal
L
U
D
f(n) = 1 + 14 = 15
f(n) = 1 + 13 = 14
f(n) = 1 + 15 = 16
f(n) = 15 + 2 = 17
f(n) = 12 + 3 = 15
R
f(n) = 10 + 4 = 14
f(n) = 10 + 4 = 14
f(n) = 10 + 4 = 14



1, 1, 3, 1, 1 2 2 2
1, 1, 3, 1, 1, 2, 3 3
Misplaced / output / manhattan
Manhattan for output

Moves to Goal: UULDR
Total Steps: 5
Unique states visited: 9

Steps 5 ;
Step 0:
2 8 3
1 6 4
7 0 5
Step 1 (U):
2 8 3
1 0 4
7 6 5
Step 2: (U)
2 0 3
1 8 4
7 6 5
Step 3 :
0 2 3
1 8 4
7 6 5
Step 4: D
1 2 3
0 8 4
7 6 5
Step 5 (R)
1 2 3
8 0 4
7 6 5

Moves to Goal: UULDR
Total steps: 5
Unique states visited: 6

Steps :
Step 0 ;
2 8 3
1 6 4
7 0 5
Step 1 (U):
2 8 3
1 0 4
7 6 5
Step 2 (U):
2 0 3
1 8 4
7 6 5
Steps (L)
0 2 3
1 8 4
7 6 5
Step 4 (D)
1 2 3
0 8 4
7 6 5
Steps (R)
1 2 3
8 0 4
7 6 5

Lab Program-5

Implement A* using 2 methods:

1. No. of misplaced tiles
2. Manhattan distance.

Algorithm

1) No. of misplaced tiles Heuristic

1. Initalize:
   Start: Initialize Start node with g = 0, Calculate h = no of tiles out of place, f = g + h. put it in open list.
2. Select: Take node with lowest f from open list. If goal, stop.
3. Expand: Generate children from possible moves. For each child, Calculate g = parent.g+1, h = misplaced titles count, f = g + h. Add to open list better
4) Repeat: loop until goal found or no nodes left

2) A* with Manhattan Distance Heuristic:

1. Start: Initialize Start node with g = 0, Calculate h = Sum of manhattan distances of tiles to goal, f = g + h, put it in open list
2. Select: pick node with lowest f from open list. If goal, done.
3. Expand: Generate child states For each child, g = parent.g + 1,

Program 4

```
import random
import time

def print_board(state):
    """Prints the chessboard for a given state."""
    n = len(state)
    for row in range(n):
        line = ""
        for col in range(n):
            if state[col] == row:
                line += "Q "
            else:
                line += ". "
        print(line)
    print()

def compute_heuristic(state):
    """Computes the number of attacking pairs of queens."""
    h = 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):
                h += 1
    return h

def get_neighbors(state):
    """Generates all possible neighbors by moving one queen in its column."""
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if row != state[col]:
                neighbor = list(state)
                neighbor[col] = row
                neighbors.append(neighbor)
    return neighbors

def hill_climb_verbose(initial_state, step_delay=0.5):
    """Hill climbing algorithm with verbose output at each step."""
    current = initial_state
    current_h = compute_heuristic(current)
```

```
    step = 0

    print(f"Initial state (heuristic: {current_h}):")
    print_board(current)
    time.sleep(step_delay)

    while True:
        neighbors = get_neighbors(current)
        next_state = None
        next_h = current_h

        for neighbor in neighbors:
            h = compute_heuristic(neighbor)
            if h < next_h:
                next_state = neighbor
                next_h = h

        if next_h >= current_h:
            print(f"Reached local minimum at step {step}, heuristic: {current_h}")
            return current, current_h

        current = next_state
        current_h = next_h
        step += 1
        print(f"Step {step}: (heuristic: {current_h})")
        print_board(current)
        time.sleep(step_delay)

def solve_n_queens_verbose(n, max_restarts=1000):
    """Solves N-Queens problem using hill climbing with random restarts."""
    for attempt in range(max_restarts):
        print(f"\n=== Restart {attempt + 1} ===\n")
        initial_state = [random.randint(0, n - 1) for _ in range(n)]
        solution, h = hill_climb_verbose(initial_state)
        if h == 0:
            print(f"✅ Solution found after {attempt + 1} restart(s):")
            print_board(solution)
            return solution
        else:
            print(f"+ No solution in this attempt (local minimum).\n")
    print("❌ Failed to find a solution after maximum restarts.")
    return None

# --- Run the algorithm ---
if __name__ == "__main__":
```

```
N = int(input("Enter the number of queens (N): "))
solve_n_queens_verbose(N)
print("\nPROGRAM BY: SRUJAN K R, USN: 1BM23CS340")
```

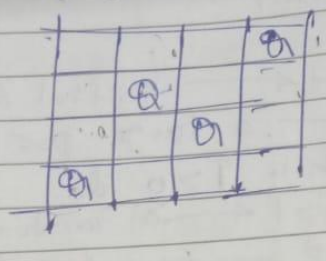
```
Enter the number of queens (N): 4

=== Restart 1 ===

Initial state (heuristic: 3):
Q . Q .
. Q . .
. . . Q
. . . .

Step 1: (heuristic: 1)
. . Q .
. Q . .
. . . Q
Q . . .

Reached local minimum at step 1, heuristic: 1
+ No solution in this attempt (local minimum).
```

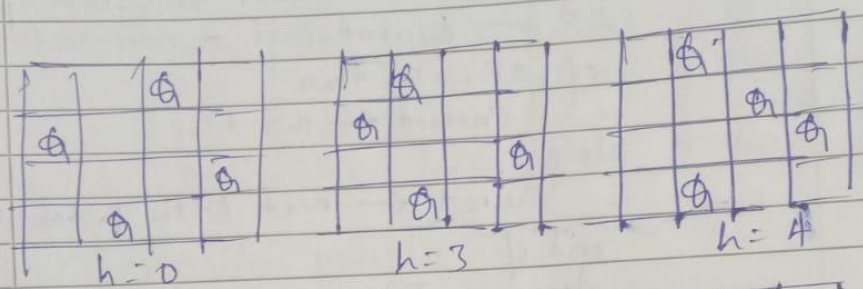
```
=== Restart 2 ===

Initial state (heuristic: 3):
. Q . .
. . Q .
. . . .
Q . . Q

Step 1: (heuristic: 1)
. Q . .
. . Q .
Q . . .
. . . Q

Reached local minimum at step 1, heuristic: 1
+ No solution in this attempt (local minimum).
```

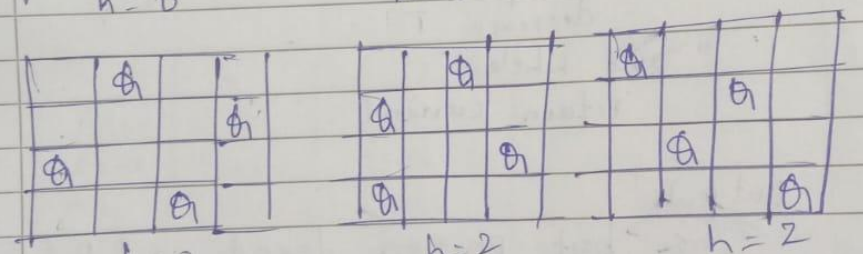
```
=== Restart 3 ===

Initial state (heuristic: 2):
. . . .
. Q . Q
. . . .
Q . Q .

Step 1: (heuristic: 1)
. Q . .
. . . Q
. . . .
Q . Q .

Step 2: (heuristic: 0)
. Q . .
. . . Q
Q . . .
. . Q .

✅ Solution found after 3 restart(s):

. Q . .
. . . Q
Q . . .
. . Q .

PROGRAM BY: SRUJAN K R, USN: 1BM23CS340
```

Transition diagram

Initial = [3, 1, 2, 0]







Output:

Total possible States : 256

State : (0, 0, 0, 0), Attacks : 6
State : (0, 0, 0, 1), Attacks : 4
State : (0, 0, 0, 2), Attacks : 4
State : (0, 0, 0, 3), Attacks : 4

15/09/2025

5. Implement Hill climbing algorithm to solve N-Queens problem

Algorithm

function Hill climbing (problem) returns a state that is a local maximum
Current ← MAKE-NODE (problem.Initial State)

loop do

neighbour ← a highest-valued successor of current if neighbour VALUE ≤ Current. Value then return Current STATE
Current ← neighbour

State: 4 queens on the board No pair of queens are attacking each other.

- Variables : x_0, x_1, x_2, x_3 where x_i is the row position of the queen in column i. Assum, that there is one queen per Column.

Initial State: a random state
Goal State: 4 queens on the board. No pair of queens are attacking each other

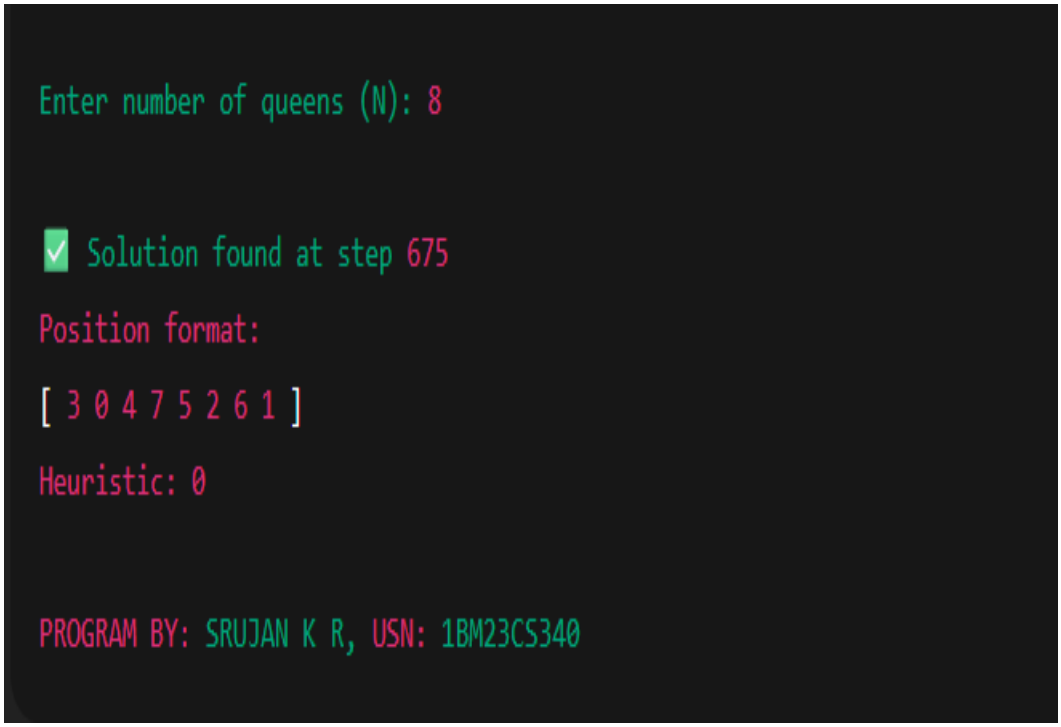
Neighbour relation : Swap the row positions of two queens.

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

Program 5

```
import random
import math

def compute_heuristic(state):
    """Computes number of attacking pairs of queens."""
    h = 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):
                h += 1
    return h

def random_neighbor(state):
    """Generates a random neighbor by changing one queen's row."""
    n = len(state)
    neighbor = state[:]
    col = random.randint(0, n - 1)
    old_row = neighbor[col]
    new_row = random.choice([r for r in range(n) if r != old_row])
    neighbor[col] = new_row
    return neighbor

def dual_simulated_annealing(n, max_iter=10000, initial_temp=100.0, cooling_rate=0.99):
    """Simulated Annealing algorithm with dual acceptance (standard + small uphill moves)."""
    current = [random.randint(0, n - 1) for _ in range(n)]
    current_h = compute_heuristic(current)
    temperature = initial_temp

    for step in range(max_iter):
        if current_h == 0:
            print(f"\n✅ Solution found at step {step}")
            return current

        neighbor = random_neighbor(current)
        neighbor_h = compute_heuristic(neighbor)
        delta = neighbor_h - current_h

        # Always accept better states
        if delta < 0:
            current = neighbor
            current_h = neighbor_h
        else:
            # Accept worse states with probability e^(-delta / T)
            probability = math.exp(-delta / temperature)
            if random.random() < probability:
                current = neighbor
                current_h = neighbor_h
```

```
        # Decrease temperature
        temperature *= cooling_rate

        # Restart if temperature gets too low (to avoid stagnation)
        if temperature < 1e-5:
            temperature = initial_temp
            current = [random.randint(0, n - 1) for _ in range(n)]
            current_h = compute_heuristic(current)

    print("\n❌ Failed to find a solution within max iterations.")
    return None

# --- MAIN PROGRAM ---
if __name__ == "__main__":
    N = int(input("Enter number of queens (N): "))
    solution = dual_simulated_annealing(N)

    if solution:
        print("\nPosition format:")
        print("[", " ".join(str(x) for x in solution), "]")
        print("Heuristic:", compute_heuristic(solution))
        print("\nPROGRAM BY: SRUJAN K R, USN: 1BM23CS340")
```



15/09/25

3) Simulate Annealing : 8 queens

Algorithm:

```
Current ← initial state
T ← a large positive value
While T > 0 do
    next ← a random neighbour 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
```

Outputs:

The best position found is [0 8 5 2 6 3 7 4]

The No. of queens that are not attacking each other: 8

Program 6

```
from itertools import product

# ----------------- Propositional Logic Symbols -----------------

class Symbol:
    def __init__(self, name):
        self.name = name

    def __invert__(self):     # ~P
        return Not(self)

    def __and__(self, other):  # P & Q
        return And(self, other)

    def __or__(self, other):   # P | Q
        return Or(self, other)

    def __rshift__(self, other):  # P >> Q  (Implication)
        return Or(Not(self), other)

    def __eq__(self, other):   # P == Q (Biconditional)
        return And(Or(Not(self), other), Or(Not(other), self))

    def eval(self, model):
        return model[self.name]

    def symbols(self):
        return {self.name}

    def __repr__(self):
        return self.name


class Not:
    def __init__(self, operand):
        self.operand = operand

    def eval(self, model):
        return not self.operand.eval(model)

    def symbols(self):
        return self.operand.symbols()
```

```
    def __repr__(self):
        return f"~{self.operand}"


class And:
    def __init__(self, left, right):
        self.left, self.right = left, right

    def eval(self, model):
        return self.left.eval(model) and self.right.eval(model)

    def symbols(self):
        return self.left.symbols() | self.right.symbols()

    def __repr__(self):
        return f"({self.left} & {self.right})"


class Or:
    def __init__(self, left, right):
        self.left, self.right = left, right

    def eval(self, model):
        return self.left.eval(model) or self.right.eval(model)

    def symbols(self):
        return self.left.symbols() | self.right.symbols()

    def __repr__(self):
        return f"({self.left} | {self.right})"


# ----------------- Truth Table Entailment Checking -----------------

def tt_entails(kb, alpha, show_table=False):
    symbols = sorted(list(kb.symbols() | alpha.symbols()))
    if show_table:
        print_truth_table(kb, alpha, symbols)
    return tt_check_all(kb, alpha, symbols, {})


def tt_check_all(kb, alpha, symbols, model):
    if not symbols:
        if kb.eval(model):       # If KB is true
            return alpha.eval(model)
        else:
            return True          # If KB false, entailment holds vacuously
```

```
    else:
        P, rest = symbols[0], symbols[1:]
        model_true = model.copy()
        model_true[P] = True
        result_true = tt_check_all(kb, alpha, rest, model_true)

        model_false = model.copy()
        model_false[P] = False
        result_false = tt_check_all(kb, alpha, rest, model_false)

        return result_true and result_false


# ----------------- Truth Table Printer -----------------
def print_truth_table(kb, alpha, symbols):
    header = symbols + ["KB", "Query"]
    print(" | ".join(f"{h:^5}" for h in header))
    print("-" * (7 * len(header)))

    for values in product([False, True], repeat=len(symbols)):
        model = dict(zip(symbols, values))
        kb_val = kb.eval(model)
        alpha_val = alpha.eval(model)
        row = [str(model[s]) for s in symbols] + [str(kb_val), str(alpha_val)]
        print(" | ".join(f"{r:^5}" for r in row))
    print()


# ----------------- MAIN SECTION -----------------
P = Symbol("P")
Q = Symbol("Q")

# Knowledge Base: (P | (Q & P))
kb = (P | (Q & P))

# Query: (Q | P)
query = (Q | P)

print("Knowledge Base:", kb)
print("Query:", query)
print()

result = tt_entails(kb, query, show_table=True)
print("Does KB entail Query?", result)
print("\nPROGRAM BY: SRUJAN K R, USN: 1BM23CS340")
```

```
Knowledge Base: (P | (Q & P))
Query: (Q | P)

  P    |   Q   |   KB   | Query
-----------------------------------
False | False | False  | False
False | True  | False  | True
True  | False | True   | True
True  | True  | True   | True

Does KB entail Query? True

PROGRAM BY: SRUJAN K R, USN: 1BM23CS340
```

22/9/2025

2. Propositional logic

Implementation of TT enumeration algorithm for deciding propositional entailment.

Ex: T.T

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

Implication operator

P	Q	$P \Rightarrow Q$
false	false	true
false	true	true
true	false	false
true	true	true

KB =

α(?)

KB ⊨ ?

P → Q

where KB is true

if it one row KB ⇒ Q ∴ KB ⊨ Q

Consider S & T as variables
& following relation

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

Write truth table & Show
whether

(i) a entails b
(ii) a entails c

S	T	$a = S \vee T$	$b = S \wedge T$	$c = \neg S \vee \neg T$
T	T	T	T	T
T	F	T	F	T
F	T	T	F	T
F	F	F	F	T

(i) Row 2 : $a = T$, $b = F$ ✗
Row 3 : $a = T$, $b = T$ ✓
∴ No a doesnot entail b

(ii) $a \models c$?
$a \models c$
c is tautology
∴ yes a entails c

22/9/25

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

Checking that $KB \models \alpha$

Output

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) Input: A knowledge base (KB) and a query (α)
2) List all Symbols: Collect all propositional variables from KB & α
3) Generate truth assignments: Create all possible truth value combinations for those symbols
4) If KB is true in that model:
   - Then check if the query α is also true in that same model.
   - If there is any model where KB is true but α is false → entailment fails
5) If in all models where KB is true α is also true (return true)

Output

Name: Srujan
USN: 1BM23CS140
KB: $(P \rightarrow Q) \wedge P$
Query: Q
Does KB entail Query?: Yes ☑

Program 7

Implement unification in first order logic

```
# ---------- Unification Algorithm ----------

class UnificationError(Exception):
    """Custom exception for unification errors."""
    pass



def occurs_check(var, term):
    """Check if a variable occurs inside a term (prevents infinite recursion)."""
    if var == term:
        return True
    if isinstance(term, tuple):  # compound term (like f(g(X)))
        return any(occurs_check(var, subterm) for subterm in term)
    return False



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

    # Case 1: identical terms → no new substitution
    if term1 == term2:
        return substitutions
```

```
# Case 2: term1 is a variable
elif isinstance(term1, str) and term1.isupper():
    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

# Case 3: term2 is a variable
elif isinstance(term2, str) and term2.isupper():
    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

# Case 4: both are compound terms
elif isinstance(term1, tuple) and isinstance(term2, tuple):
    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
```

```
    # Case 5: constants mismatch → fail
    else:
        raise UnificationError(f"Cannot unify: {term1} with {term2}")


# ---------- MAIN PROGRAM ----------

term1 = ('p', 'b', 'X', ('f', ('g', 'Z')))
term2 = ('p', 'Z', ('f', 'Y'), ('f', 'Y'))

try:
    print("Term 1:", term1)
    print("Term 2:", term2)
    print("\nPerforming Unification...\n")

    result = unify(term1, term2)
    print("✅ Unification Successful!")
    print("Most General Unifier (MGU):", result)

except UnificationError as e:
    print("❌ Unification Failed:", e)

print("\nPROGRAM BY: SRUJAN K R, USN: 1BM23CS340")
```

```
Term 1: ('p', 'b', 'X', ('f', ('g', 'Z')))
Term 2: ('p', 'Z', ('f', 'Y'), ('f', 'Y'))

Performing Unification...

✅ Unification Successful!
Most General Unifier (MGU): {'Z': 'b', 'X': ('f', 'Y')}

PROGRAM BY: SRUJAN K R, USN: 1BM23CS340
```

Unification Algorithm

It is the process used to find Substitution that make different FOL (first order logic)

Algorithm / Definition

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

$\theta = x/Jane/Jane$

$x/Jane$

Unify (knows (John, x), knows (y, Bill))

$\theta = y/John$

knows (John, x), knows (John, Bill)

$\theta = x/Bill$

knows (John, Bill), knows (John, Bill)

2. Find the MGU of

~~{p(b, x, f(g(z))}~~

~~{p(z, f(y), f(y))}~~

Algo

Unify(Ψ_1, Ψ_2)

1. If Ψ 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 $\{(\Psi_2 / \Psi_1)\}$
   (c) Else if Ψ_2 is a variable.
      (a) If Ψ_2 occurs in Ψ_1 then return FAILURE
      (b) Else return $\{(\Psi_1 / \Psi_2)\}$
   (d) Else return FAILURE
2. If the initial predicate symbol in Ψ_1 & Ψ_2 are not same, then return FAILURE.
3. If Ψ_1 & Ψ_2 have a different number of argument then return FAILURE
4. Set Substitution set (SUBST) to Nil.
5. For i=1 to the no of element in Ψ_1.
   (a) Call Unify function with the i^{th} element of Ψ_1 & i^{th} element of Ψ_2, & put the result into S.
   (b) If S = failure then returns failure.
   (c) If S ≠ NIL then do,
      (a) Apply S to the remainder of both L_1 & L_2
      (b) SUBST = Append (S, SUBST)
6. Return SUBST.

Solve the following

1) Find most General Unifier (MGU) of
{p(b, x, f(g(z)) and p(z, f(y), f(y))}

S = {}
z → b → S = {z → b}
x → f(y) → S = {z → b, x → f(y)}
y → g(z) → Substitute into x → x → f(g(z))

Final MGU: {z → b, x → f(g(z)), y → g(z)}

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

S = {}
x → f(b) → S = {x → f(b)}
x → y → b → S = {x → f(b), y → b}

Final MGU: {x → f(b), y → b}

3. {p(f(a), g(y)), p(x, x)}
S = {}
from arg1: x → f(a) → S = {x → f(a)}
from arg2: x must = g(y) → requires f(y) = g(y)
∴ Unification fails.

4) {prime(11), prime(y)}
S = {}
y → 11
Final MGU: {y → 11}

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

S = { }

y = John → S = { y → John }

x → mother (John) → S = { y → John, x → mother (John) }

Final MGU: { y → John, x → mother (John) }

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

S = { }

y → John → S = { y → John }

x → Bill → S = { y → John, x → Bill }

Final MGU: { y → John, x → Bill }

Output (1) Astr

Input :

p (b, x, f (g (z))

p (z, f (y), f (y))

Trace :

Step 1 : Unify p (b, x, f (g (z))) with p (z, f (y), f (y)) S = { }

Step 2 : Unify b with z S = { z : b }

Step 3 : Unify x with f (y) S = { 'z' : b }

Step 4 : Unify f (g (z)) with f (y) S = { 'z' : b, 'x' : f (y) }

Step 5 : Unify g (z) with y S = { 'z' : b, 'x' : f (y) }

Final MGU :

z → b

x → f (g (z))

y → g (z)

Program 8

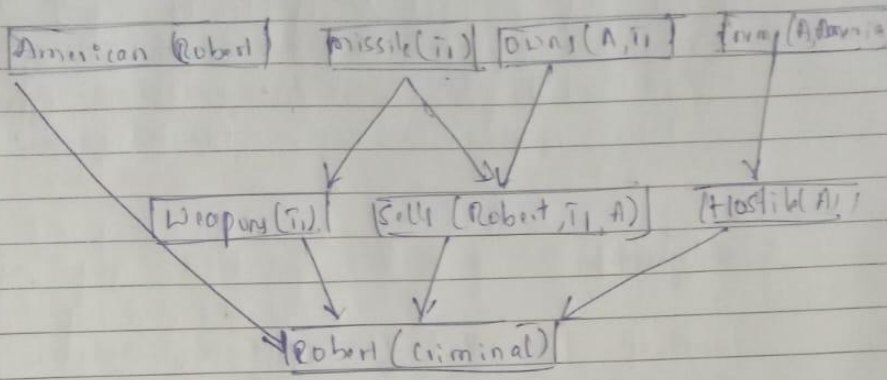
Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

```
# ---------- Forward Chaining Algorithm ----------

facts = {
    "American(Robert)",
    "Enemy(A, America)",
    "Missile(T1)",
    "Owns(A, T1)"
}

rules = [
    ("Enemy(x, America)", "Hostile(x)"),
    ("Missile(x)", "Weapon(x)"),
    ("Missile(x) ∧ Owns(A, x)", "Sells(Robert, x, A)"),
    ("American(p) ∧ Weapon(q) ∧ Sells(p, q, r) ∧ Hostile(r)", "Criminal(p)")
]


def apply_rules(facts):
    new_facts = set()

    # Rule 1: Enemy(A, America) → Hostile(A)
    if "Enemy(A, America)" in facts and "Hostile(A)" not in facts:
        print("Step 1: Enemy(A, America) → Hostile(A)")
        new_facts.add("Hostile(A)")

    # Rule 2: Missile(T1) → Weapon(T1)
    if "Missile(T1)" in facts and "Weapon(T1)" not in facts:
        print("Step 2: Missile(T1) → Weapon(T1)")
        new_facts.add("Weapon(T1)")

    # Rule 3: Missile(T1) ∧ Owns(A, T1) → Sells(Robert, T1, A)
    if "Missile(T1)" in facts and "Owns(A, T1)" in facts and "Sells(Robert, T1, A)" not in
facts:
        print("Step 3: Missile(T1) ∧ Owns(A, T1) → Sells(Robert, T1, A)")
        new_facts.add("Sells(Robert, T1, A)")

    # Rule 4: American(Robert) ∧ Weapon(T1) ∧ Sells(Robert, T1, A) ∧ Hostile(A) →
Criminal(Robert)
    if {
        "American(Robert)",
        "Weapon(T1)",
        "Sells(Robert, T1, A)",
        "Hostile(A)"
    } <= facts and "Criminal(Robert)" not in facts:
        print("Step 4: American(Robert) ∧ Weapon(T1) ∧ Sells(Robert, T1, A) ∧ Hostile(A) →
Criminal(Robert)")
        new_facts.add("Criminal(Robert)")
```

```
    return new_facts


# ---------- Inference Loop ----------
step = 1
while True:
    new_facts = apply_rules(facts)
    if not new_facts:
        break
    facts |= new_facts
    step += 1

# ---------- Final Output ----------
print("\n✅ Final Facts:")
for fact in facts:
    print(fact)

print("\nPROGRAM BY: SRUJAN K R, USN: 1BM23CS340")
```

```
Step 1: Enemy(A, America) → Hostile(A)
Step 2: Missile(T1) → Weapon(T1)
Step 3: Missile(T1) ∧ Owns(A, T1) → Sells(Robert, T1, A)
Step 4: American(Robert) ∧ Weapon(T1) ∧ Sells(Robert, T1, A) ∧ Hostile(A) → Criminal(Robert)

✅ Final Facts:
Weapon(T1)
Criminal(Robert)
Hostile(A)
Sells(Robert, T1, A)
Missile(T1)
Enemy(A, America)
Owns(A, T1)
American(Robert)

PROGRAM BY: SRUJAN K R, USN: 1BM23CS340
```

Forward chaining proof



Output

Step 1 : Enemy (A, America) → Hostile (A)
Step 2 : missile (T1) → Weapon (T1)
Step 3 : missile (T1) ∧ owns (A, T1) → Sells (Robert, T1, A)
Step 4 : American (Robert) ∧ Weapon (T1) ∧
Sells (Robert, T1, A) ∧ Hostile (A) → Criminal (Robert)

Final facts

American (Robert)
Enemy (A, America)
missile (T1)
Owns (A, T1)
Hostile (A)
Weapon (T1)
Sells (Robert, T1, A)
Criminal (Robert)

Week 8

12. First order logic

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

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
    (p1 ∧ ... ∧ pn => q) ← STANDARDIZE-VARIABLES(rule)
    for each θ such that SUBST(θ, p1 ∧ ... ∧ pn)
      = SUBST(θ, p1' ∧ ... ∧ pn')
      q' ← SUBST(θ, q)
      if q' doesn't ~~unify~~ with some sentence 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.

---------- Program 9: Resolution Algorithm ----------

```
class Literal:
    def __init__(self, name, negated=False):
        self.name = name
        self.negated = negated

    def __repr__(self):
        return f"¬{self.name}" if self.negated else self.name

    def __eq__(self, other):
        return isinstance(other, Literal) and self.name == other.name and self.negated == other.negated

    def __hash__(self):
        return hash((self.name, self.negated))


def convert_to_cnf(sentence):
    """Placeholder for CNF conversion (already assumed CNF input)."""
    return sentence


def negate_literal(literal):
    """Return the negation of a literal."""
    return Literal(literal.name, not literal.negated)


def resolve(clause1, clause2):
    """Apply resolution rule to two clauses and return resolvents."""
    resolvents = []
    for literal1 in clause1:
        for literal2 in clause2:
            if literal1.name == literal2.name and literal1.negated != literal2.negated:
                new_clause = (set(clause1) - {literal1}) | (set(clause2) - {literal2})
                resolvents.append(frozenset(new_clause))
    return resolvents


def prove_conclusion(premises, conclusion):
    cnf_premises = [convert_to_cnf(p) for p in premises]
    cnf_conclusion = convert_to_cnf(conclusion)
    negated_conclusion = frozenset({negate_literal(lit) for lit in cnf_conclusion})

    clauses = set(frozenset(p) for p in cnf_premises)
    clause_list = list(clauses)
    id_map = {}
    parents = {}
    clause_id = 1

    for c in clause_list:
        id_map[c] = f"C{clause_id}"
```

```
        clause_id += 1

    id_map[negated_conclusion] = f"C{clause_id} (¬Conclusion)"
    clauses.add(negated_conclusion)
    clause_list.append(negated_conclusion)
    clause_id += 1

    print("\n--- Initial Clauses ---")
    for c in clause_list:
        print(f"{id_map[c]}: {set(c)}")

    print("\n--- Resolution Steps ---")
    while True:
        new_clauses = set()
        for i in range(len(clause_list)):
            for j in range(i + 1, len(clause_list)):
                resolvents = resolve(clause_list[i], clause_list[j])
                for r in resolvents:
                    if r not in id_map:
                        id_map[r] = f"C{clause_id}"
                        parents[r] = (id_map[clause_list[i]], id_map[clause_list[j]])
                        clause_id += 1

                        print(f"{id_map[r]} = RESOLVE({id_map[clause_list[i]]}, {id_map[clause_list[j]]}) -> {set(r)}")

                        if not r:  # Empty clause derived
                            print("\nEmpty clause derived!")
                            print("\n--- Resolution Tree ---")
                            print_resolution_tree(parents, id_map, r)
                            return True

                        new_clauses.add(r)

        if new_clauses.issubset(clauses):
            print("\nNo new clauses can be derived.")
            return False

        clauses |= new_clauses
        clause_list = list(clauses)


def print_resolution_tree(parents, id_map, empty_clause):
    """Print resolution tree leading to the empty clause."""
    def recurse(clause):
        if clause not in parents:
            print(f" {id_map[clause]}: {set(clause)}")
            return
        left, right = parents[clause]
        print(f" {id_map[clause]} derived from {left} and {right}")
        for parent_clause, parent_name in zip(
            [k for k, v in id_map.items() if v in [left, right]], [left, right]
        ):
```

```
            recurse(parent_clause)

    recurse(empty_clause)


def parse_literal(lit_str):
    """Convert string to Literal object."""
    lit_str = lit_str.strip()
    if lit_str.startswith("¬") or lit_str.startswith("-"):
        return Literal(lit_str[1:], True)
    return Literal(lit_str, False)


def get_user_input():
    """Input premises and conclusion."""
    premises = []
    num_premises = int(input("Enter number of premises: "))
    for i in range(num_premises):
        clause_str = input(f"Enter clause {i+1} (e.g., A, ¬B, C): ")
        literals = {parse_literal(l) for l in clause_str.split(",")}
        premises.append(literals)

    conclusion_str = input("Enter conclusion (e.g., C or ¬C): ")
    conclusion = {parse_literal(l) for l in conclusion_str.split(",")}
    return premises, conclusion


# ---------- MAIN PROGRAM ----------
if __name__ == "__main__":
    premises, conclusion = get_user_input()

    if prove_conclusion(premises, conclusion):
        print("\n✅ Conclusion can be proven from the premises.")
    else:
        print("\n❌ Conclusion cannot be proven from the premises.")

                                print("\nPROGRAM BY: SRUJAN K R, USN: 1BM23CS340")
```

```
Enter number of premises: 4
Enter clause 1 (e.g., A, ¬B, C): A
Enter clause 2 (e.g., A, ¬B, C): ¬A
Enter clause 3 (e.g., A, ¬B, C): ¬B
Enter clause 4 (e.g., A, ¬B, C): C
Enter conclusion (e.g., C or ¬C): C

--- Initial Clauses ---
C1: {A}
C2: {¬A}
C3: {¬B}
C4: {C}
C5 (¬Conclusion): {¬C}

--- Resolution Steps ---
C6 = RESOLVE(C4, C5 (¬Conclusion)) -> set()

Empty clause derived!

--- Resolution Tree ---
 C6 derived from C4 and C5 (¬Conclusion)
 C4: {'C'}
 C5 (¬Conclusion): {'¬C'}

✅ Conclusion can be proven from the premises.

PROGRAM BY: SRUJAN K R, USN: 1BM23CS340
```

Week-9 27/10/2025

First Order Logic

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

Algorithm

Eliminate biconditions and implications:

-> Eliminate $\Leftrightarrow$ replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$

-> Eliminate $\Rightarrow$ replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$

Move $\neg$ inwards:

-> $\neg(\forall x\ p) \equiv \exists x\ \neg p$

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

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

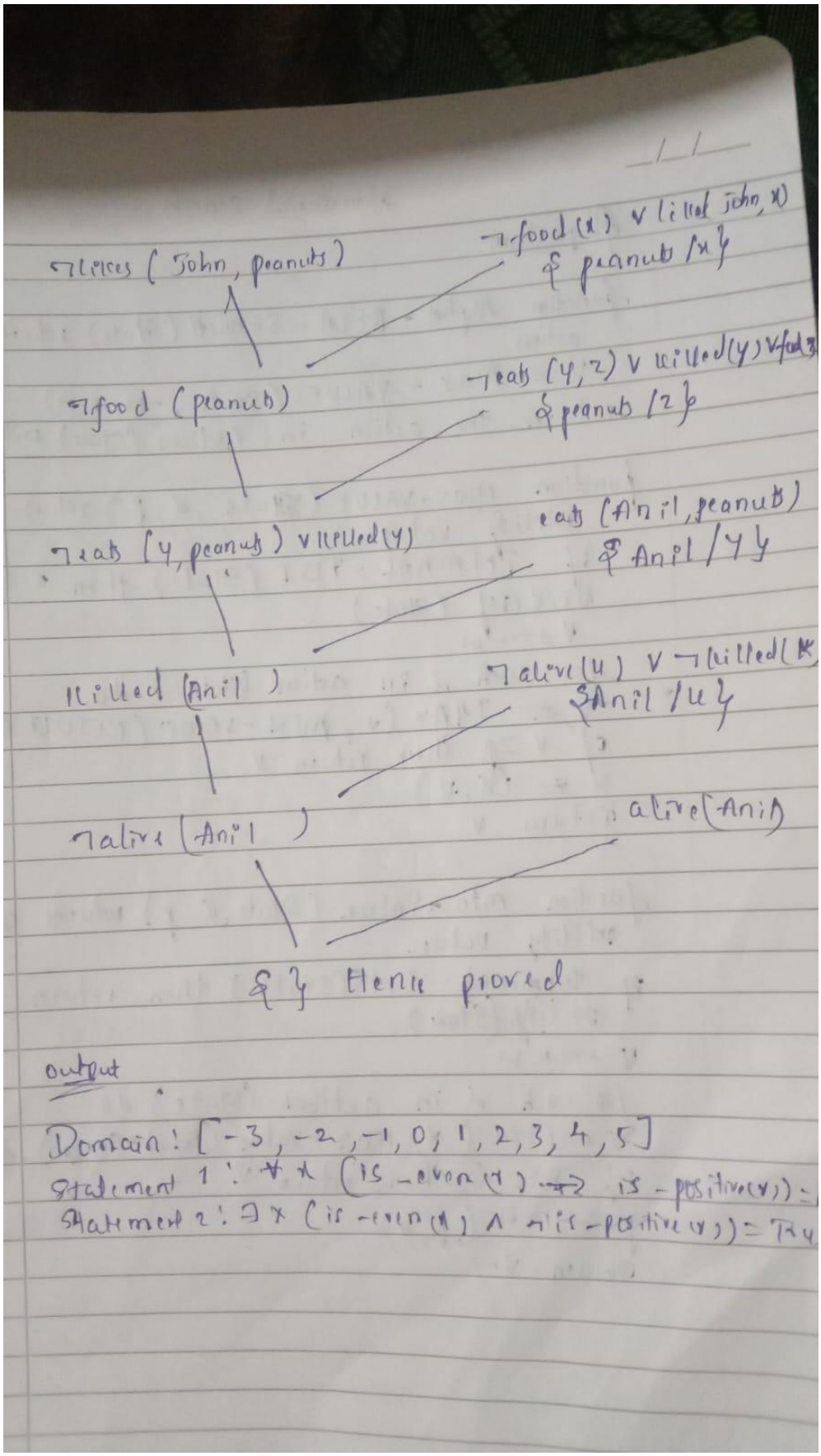
$\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$,

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

Standardize: Variables apart by renaming each quantifier should use a different variable.

Skolemize each existential variable is replaced by a skolem constant or skolem function of the enclosing universally quantified variables.

For instance, $\exists x$ Rich(x) becomes Rich(G_1), where G_1 is a new skolem constant.



output

Domain: [-3, -2, -1, 0, 1, 2, 3, 4, 5]

Statement 1: ∀x (is_even(x) → is_positive(x)) =

Statement 2: ∃x (is_even(x) ∧ ¬is_positive(x)) = Tru

Everyone has a heart" $\forall x$ person $(x) \Rightarrow$
$\exists y$ Heart $(y) \wedge$ has (x, y) becomes $\forall x$
person $(x) \Rightarrow$ Heart $(H(x)) \wedge$ Has $(x, H(x))$
where H is a new symbol (Skolem function)

2) Drop universal quantifiers
→ For instance, $\forall x$ person (x) becomes person (x)

3. Distribute $\wedge$ over $\vee$:
$(\alpha \wedge \beta) \vee \gamma \equiv (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$

Output

$\neg$ Food $(x) \vee$ Likes (John, x)
Food (Apple)
$\neg$ Eats $(y, z) \vee$ Killed $(y) \vee$ Food (z)
Eats (Anil, Peanuts)
Alive (Anil)
$\neg$ Eats (Anil, w) $\vee$ Eats (Harry, w)
Killed $(g) \vee$ Alive (g)
$\neg$ Alive $(u) \vee \neg$ Killed (u)
Likes (John, Peanuts)

--------- Program 10: Alpha-Beta Pruning ---------

```
import math

def alpha_beta(node, depth, alpha, beta, maximizingPlayer, game_tree):
    """Perform Alpha-Beta pruning on the game tree."""

    # Base case: Leaf node or depth limit reached
    if depth == 0 or isinstance(game_tree[node], int):
        return game_tree[node]

    # Maximizing player's turn
    if maximizingPlayer:
        maxEval = -math.inf
        for child in game_tree[node]:
            eval = alpha_beta(child, depth - 1, alpha, beta, False, game_tree)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:  # Beta cut-off
                break
        return maxEval

    # Minimizing player's turn
    else:
        minEval = math.inf
        for child in game_tree[node]:
            eval = alpha_beta(child, depth - 1, alpha, beta, True, game_tree)
            minEval = min(minEval, eval)
            beta = min(beta, eval)
            if beta <= alpha:  # Alpha cut-off
                break
        return minEval


# ---------- Game Tree Representation ----------
game_tree = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': 3,
    'E': 5,
    'F': 2,
    'G': 9
}

# ---------- Run Alpha-Beta Search ----------
best_value = alpha_beta(
    'A', depth=3, alpha=-math.inf, beta=math.inf, maximizingPlayer=True, game_tree=game_tree
)

print("Best value for maximizer:", best_value)
```

print("\nPROGRAM BY: SRUJAN K R, USN: 1BM23CS340")

```
Best value for maximizer: 3

PROGRAM BY: SRUJAN K R, USN: 1BM23CS340
```

Week - 10

Adversarial search

27/10/25

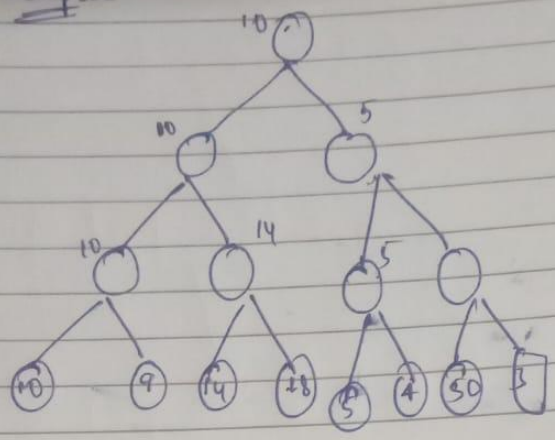
Algorithm

function Alpha - BETA - SEARCH (State) returns an action
$V \leftarrow$ max - VALUE (State, $-\infty, +\infty$)
return the action in Actions (State) with value V

function max-VALUE (State, α, β) returns a utility value
if TERMINAL - TEST (State) then return UTILITY (State)
$V \leftarrow -\infty$
for each α in actions (State) do
$V \leftarrow$ MAX (V, MIN - VALUE (RESULT (s, a), α, β))
if $V \geq \beta$ then return V
$\alpha \leftarrow$ (α, V)
return V

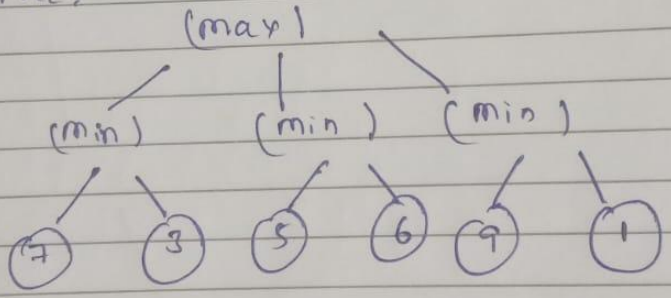
function min-Value (State, α, β) returns a utility value.
if terminal - test (State) then return utility (State)
$V \leftarrow +\infty$
for each α in action (State) do
$V \leftarrow$ min (V, max - value (Result (s, a), α, β))
if $V \leq \alpha$ then return V
$\beta \leftarrow$ min (β, V)
return V.

Output



Output

Game Tree:



Calculated value at Root (max) = 5