VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Siddhant sahare (1BM23CS326)

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 **Siddhant sahare(1BM23CS326),** who is Bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

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

Index

Name: Siddhant Suhare
Dept: CSE
USN: 1BM23CS326

Index

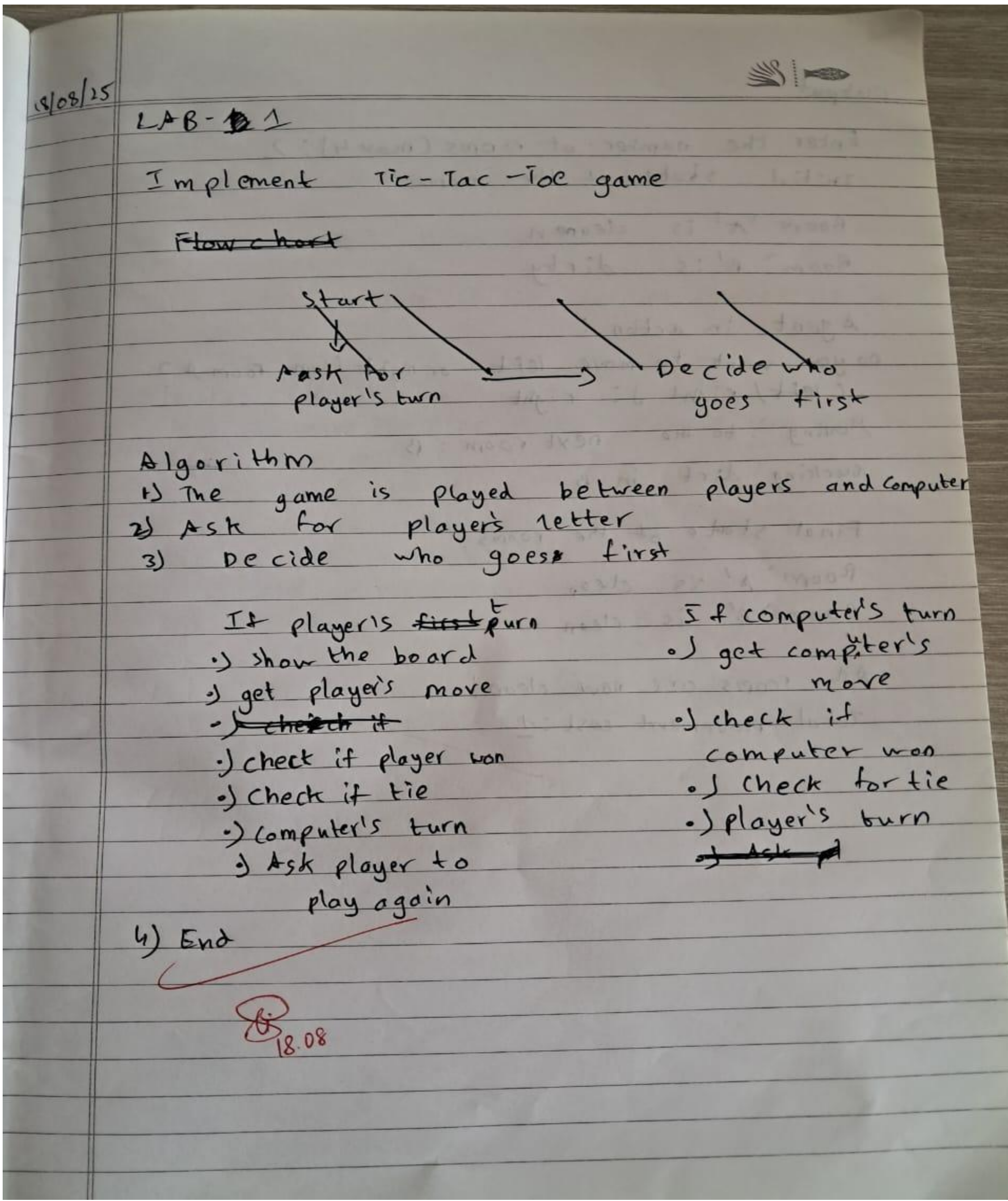
o	Date	Title	Teacher sign	Marks
	18/08/25	Tic Tac Toe		
.	24/08/25	Vaccum Cleaner		
2.	01/09/25	a) BFS & DFS without heuristic		
		b) BFS with heuristic		
		c) iterative Deepening [DFS]		
.	08/09/25	A* misplaced tiles		
		Manhattan distance	22.09	
.	15/09/25	4 Queens hill Climbing		
		Simulated Annealing		
	22/09/25	Enumeration Algorithm	13.10	
	06/09/25	Unification algorithm	13.10	
.	13/10/25	Foward Reasoning algo		
	27/10/25	KB of First order logic		
	27/10/25	Alphabeta pruning	10.11	

Github Link : https://github.com/Siddhant0Sahare/1BM23CS326-AI

<u>Program 1.1</u>

Implement Tic –Tac –Toe Game

Algorithm:



18/08/25

LAB - 1

Implement Tic - Tac - Toe game

~~Flow chart~~

Start

Ask for player's turn

Decide who goes first

Algorithm

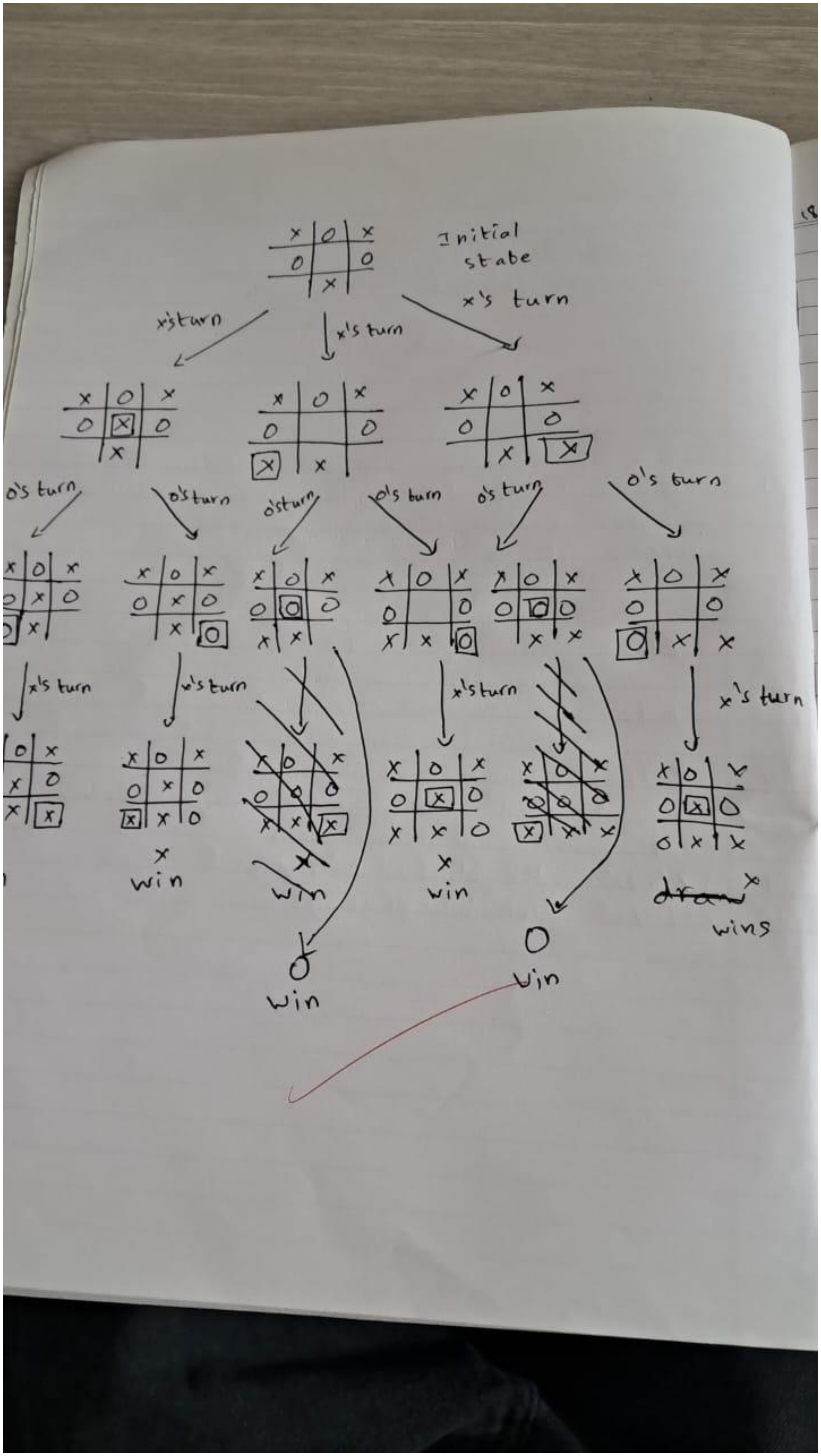
1) The game is played between players and computer
2) Ask for player's letter
3) Decide who goes first

If player's ~~first~~ turn
- show the board
- get player's move
- ~~check if~~
- check if player won
- Check if tie
- computer's turn
- Ask player to play again

If computer's turn
- get computer's move
- check if computer won
- Check for tie
- player's turn
- ~~Ask~~

4) End

18.08


Initial stabe
x's turn
x's turn
x's turn
o's turn
o's turn
o's turn
o's turn
o's turn
o's turn
x's turn
x's turn
x's turn
x's turn
x win
x win
O win
O win
x wins

Code:

```
import random

def display_board(board):
    """Displays the current state of the board."""
    print("-------------")
    for i in range(3):
        print("|", board[i*3], "|", board[i*3+1], "|", board[i*3+2], "|")
        print("-------------")

def is_winner(board, player):
    """Checks if the given player has won."""
    # Check rows, columns, and diagonals
    return ((board[0] == board[1] == board[2] == player) or
            (board[3] == board[4] == board[5] == player) or
            (board[6] == board[7] == board[8] == player) or
            (board[0] == board[3] == board[6] == player) or
            (board[1] == board[4] == board[7] == player) or
            (board[2] == board[5] == board[8] == player) or
            (board[0] == board[4] == board[8] == player) or
            (board[2] == board[4] == board[6] == player))

def is_board_full(board):
    """Checks if the board is full."""
    return " " not in board

def get_player_move(board, player):
    """Gets the player's move."""
    while True:
        try:
            move = int(input(f"Player {player}, enter your move (1-9): ")) - 1
            if move >= 0 and move <= 8 and board[move] == " ":
                return move
            else:
                print("Invalid move. Please try again.")
        except ValueError:
            print("Invalid input. Please enter a number between 1 and 9.")

def play_game():
    """Plays a game of Tic Tac Toe between two players."""
    # Initial board state with 3 empty tiles (adjust as needed)
    board = ["X", "O", "x",
             " ", " ", "O",
             "O", " ", "X"]

    # Check if the initial state has a winner
```

```
    if is_winner(board, "X") or is_winner(board, "O"):
        print("Initial board state has a winner. Please change the initial board configuration.")
        return

    display_board(board)

    current_player = "1"

    while True:
        player_symbol = "X" if current_player == "1" else "O"
        player_move = get_player_move(board, current_player)
        board[player_move] = player_symbol
        display_board(board)

        if is_winner(board, player_symbol):
            print(f"Player {current_player} wins!")
            break
        if is_board_full(board):
            print("It's a tie!")
            break

        # Switch player
        current_player = "2" if current_player == "1" else "1"

if __name__ == "__main__":
    play_game()

print("siddhant sahare 1bm23cs326")
```

Output:

```
| X |   | X |
-------------
Player 2, enter your move (1-9): 5
-------------
| X | O | x |
-------------
| O | O | O |
-------------
| X |   | X |
-------------
Player 2 wins!
siddhant sahare 1bm23cs326
```

```
-------------
| x | o | x |
-------------
|   |   | o |
-------------
| o |   | x |
-------------
Player 1, enter your move (1-9): 5
-------------
| x | o | x |
-------------
|   | x | o |
-------------
| o |   | x |
-------------
Player 1 wins!
siddhant sahare 1bm23cs326
```

```
| o |   | x |
-------------
Player 1, enter your move (1-9): 8
-------------
| x | o | x |
-------------
| x | o | o |
-------------
| o | x | x |
-------------
It's a tie!
siddhant sahare 1bm23cs326
```

Program 1.2

Implement Vacuum Cleaner Agent

Algorithm:

LAB-2

Implement vaccum cleaner agent

~~Algorithm~~ Output: Clean dirt

The current location of the vaccum cleaner is the ~~inp~~ input

Pseudo code:

```
~~clean~~ Note: The robot is at the room given by user
  a = 0
while ( a!= 4)
  if current room is clean:
     if not clean it
  Ask user for next room location
    a++
```

LAB-~~1~~2

Implement vaccum cleaner agent

~~Algorithm~~ Output: Clean dirt

The current location of the vaccum cleaner is the ~~inp~~ input

Pseudo code:

~~clean~~ Note: The ~~p~~ robot is at the room given by user

```
a = 0
while ( a != 4)
  if current_room is clean:
      if not clean it
  Ask user for next room location
    a++
```

Code:

```
import random

NAME = "Siddhant "
USN = "1BM23CS326"

def reflex_vacuum_agent(location, status):
    if status == "Dirty":
        return "Suck"
    elif location == "A":
        return "Right"
    elif location == "B":
        return "Left"

def vacuum_world():
    locations = {"A": random.choice(["Clean", "Dirty"]),
                 "B": random.choice(["Clean", "Dirty"])}
    location = random.choice(["A", "B"])

    print("Initial State:", locations, "| Vacuum at:", location)

    steps = 5
    for _ in range(steps):
        status = locations[location]
        action = reflex_vacuum_agent(location, status)
        print(f"Vacuum at {location} | Status: {status} -> Action: {action}")

        if action == "Suck":
            locations[location] = "Clean"
        elif action == "Right":
            location = "B"
        elif action == "Left":

        print("World State:", locations)

    print("\nFinal State:", locations)
    print(f"\nSubmitted by: {NAME}, USN: {USN}")

vacuum_world()
```

Output:

```
Enter the number of rooms (max 4): 2
Initial state of the rooms:
Room 'A' is clean
Room 'B' is dirty

Agent in action:
Do you want to move left or right from room A? (left/right): right
Moving to the next room: B
Sucking dirt in B

Final state of the rooms:
Room 'A' is clean
Room 'B' is clean

All rooms are now clean!
Total movement cost: 1

SIDDHANT SAHARE

1BM23CS326
```

Program 2

Implement 8 puzzle problems using Depth First Search (BFS):

25 LAB-3

To implement 8 puzzle program using non-heuristic approach in bfs

Algorithm

1) start - goal state
2) Take string input
3) bfs(start state)
4) ~~state~~ Make move accordingly but don't do invalid moves
5) Add each state to a queue and each move to a path
6) If the state = goal state
7) Return path

Output:

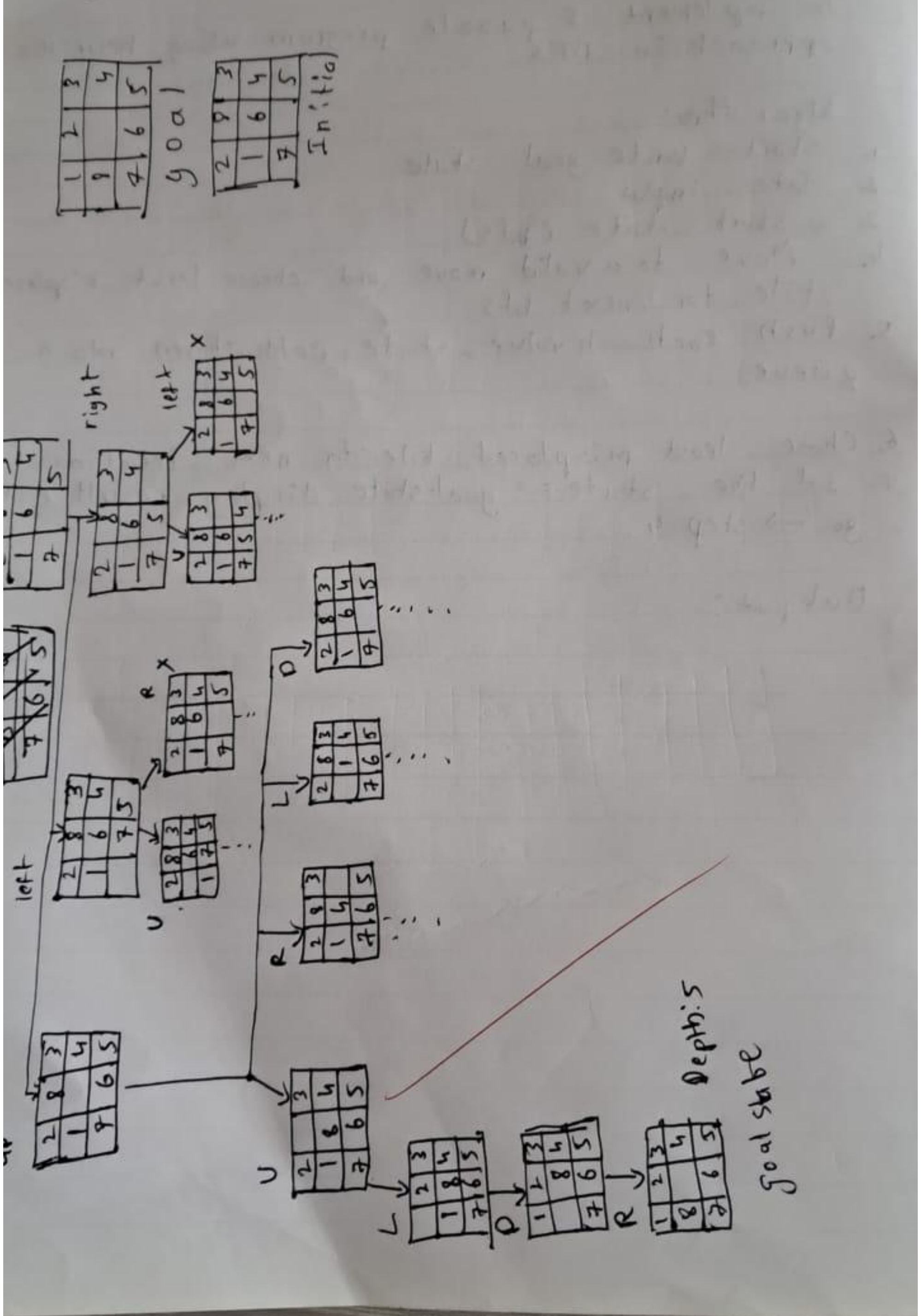
Enter initial & final state

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

Path : RDR

No. of moves : 3

using DFS Solve 8 puzzle without heuristic



Code (BFS):

```
from collections import deque

# Goal state for the 8 puzzle
goal_state = '123804765'  # Using '0' as the empty tile

# Moves: U, D, L, R (Up, Down, Left, Right)
moves = {
    'U': -3,
    'D': 3,
    'L': -1,
    'R': 1
}

# Valid moves for each index to prevent wrapping around rows/columns
invalid_moves = {
    0: ['U', 'L'], 1: ['U'], 2: ['U', 'R'],
    3: ['L'],        5: ['R'],
    6: ['D', 'L'], 7: ['D'], 8: ['D', 'R']
}

# Function to generate new puzzle state after moving the blank
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)
    # Swap 0 with the target tile
    state_list[index], state_list[new_index] = state_list[new_index], state_list[index]
    return ''.join(state_list)

# Helper function to print the state in 3x3 format
def print_state(state):
    for i in range(0, 9, 3):
        # Replace '0' with space for readability
        print(' '.join(state[i:i+3]).replace('0', ' '))
    print()  # Blank line for spacing

# BFS Algorithm with printing all visited states
def bfs(start_state):
    visited = set()
    queue = deque([(start_state, [])])  # Each element is (state, path)

    while queue:
        current_state, path = queue.popleft()
```

```
        if current_state in visited:
            continue

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

        if current_state == goal_state:
            return path

        visited.add(current_state)

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

    return None  # No solution found

# Input initial state as a string (e.g., '123456780' where 0 is the blank)
start = input("Enter start state (e.g., 724506831): ")

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

    result = bfs(start)

    if result is not None:
        print("Solution found!")
        print("Moves:", ' '.join(result))
        print("Number of moves:", len(result))
        print("1BM23CS26 Siddhant sahare\n")

        # Print puzzle states after each move in solution path
        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.")
else:
    print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")
```

Output (BFS):

```
Solution found!
Moves: U U L D R
Number of moves: 5
1BM23CS26 Siddhant sahare

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

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

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

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

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

Output (DFS):

```
  6 5

Move 44: U
2 8 3
  1 4
7 6 5

Move 45: R
2 8 3
1   4
7 6 5

Move 46: U
2   3
1 8 4
7 6 5

Move 47: L
  2 3
1 8 4
7 6 5

Move 48: D
1 2 3
  8 4
7 6 5

Move 49: R
1 2 3
8   4
7 6 5
```

Program 3

Implement A* search algorithm

LAB - ~~3~~ 4

Apply A* Algorithm.
-> Misplaced tiles

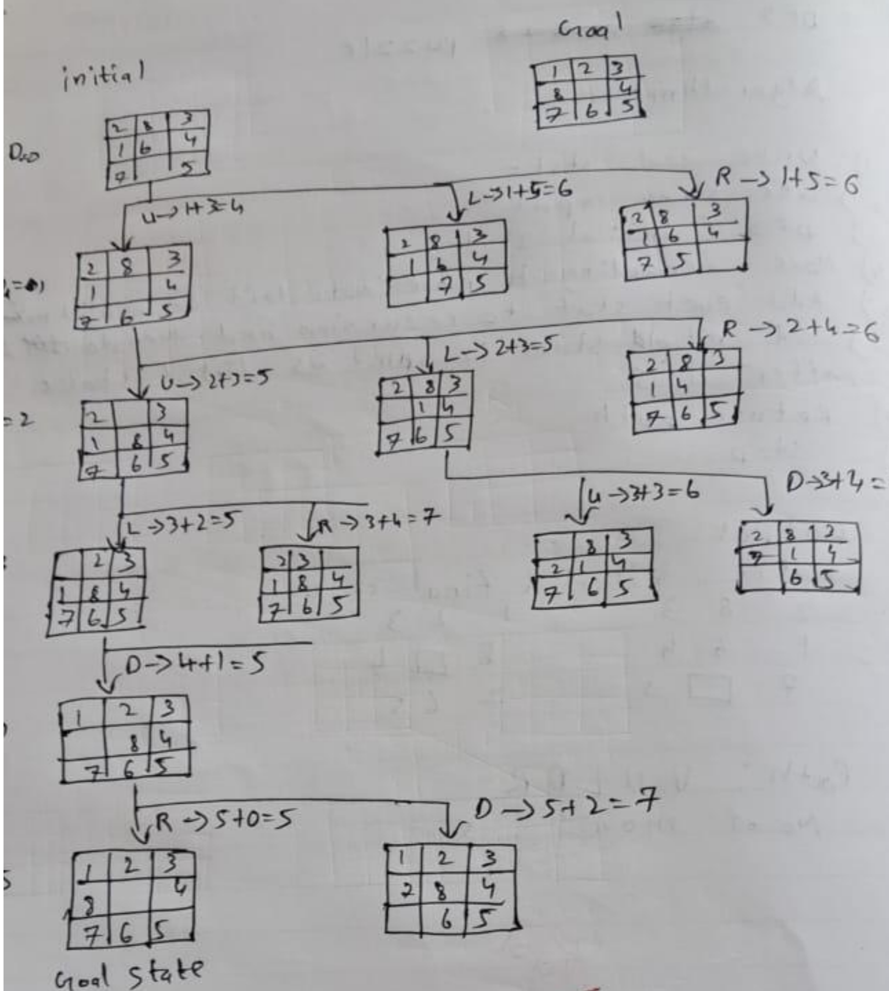
1) A* search evaluates nodes by combining g(n), the cost to reach the node and h(n), the cost to get from the node to the goal

-> $f(n) = g(n) + h(n)$

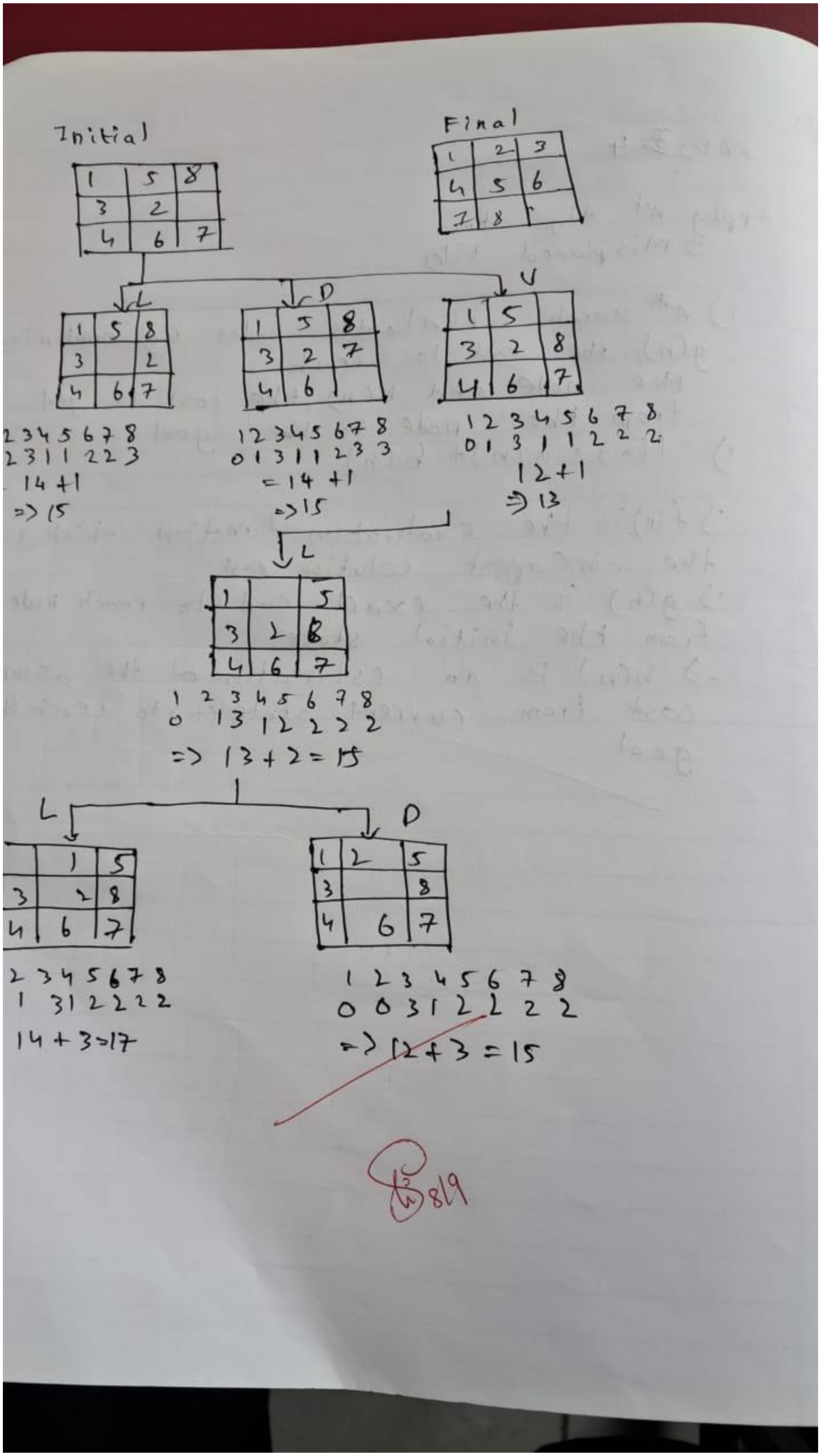
-> f(n) is the evaluation function which gives the cheapest solution cost

-> g(n) is the exact cost to reach node n from the initial state

-> h(n) is an estimation of the assumed cost from current state (n) to reach the goal



Path length = 5
Path taken = UULDR


Initial
Final
L
D
U
12345678
01311233
=14+1
=>15
12345678
01311222
12+1
=>13
L
12345678
013 12222
=> 13+2=15
L
D
14+3=17
12345678
00312222
=> 12+3=15

Code (Misplaced Tiles):

```
import copy
import heapq


NAME = "SIDDHANT "
 USN = "1BM23CS326"

print(f"Name: {NAME}")
print(f"USN : {USN}\n")


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

def heuristic_misplaced(current_state):
    misplaced = 0
    for i in range(3):
        for j in range(3):
            if current_state[i][j] != 0 and current_state[i][j] != goal_state[i][j]:
                misplaced += 1
    return misplaced

def is_goal(state):
    return state == goal_state

def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(-1,0), (1,0), (0,-1), (0,1)]
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny <3:
            new_state = copy.deepcopy(state)
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(new_state)
    return neighbors

def a_star_misplaced(initial_state):
```

```
    open_list = []
    heapq.heappush(open_list, (heuristic_misplaced(initial_state), 0, initial_state, []))
    visited = set()

    while open_list:
        heuristic, g, current_state, path = heapq.heappop(open_list)
        state_id = str(current_state)
        if state_id in visited:
            continue
        visited.add(state_id)

        if is_goal(current_state):
            return path + [current_state]

        for neighbor in get_neighbors(current_state):
            heapq.heappush(open_list, (heuristic_misplaced(neighbor) + g + 1, g + 1, neighbor, path +
[current_state]))
    return None

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

solution = a_star_misplaced(initial_state)

print("\nSolution Steps:\n")
for step in solution:
    for row in step:
        print(row)
    print()
```

Code (Manhattan Distance):

```
import copy
import heapq

NAME = "Siddhant"
USN = "1BM23CS326"

print(f"Name: {NAME}")
print(f"USN : {USN}\n")

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

def heuristic_manhattan(current_state):
```

```
    distance = 0
    for i in range(3):
        for j in range(3):
            value = current_state[i][j]
            if value != 0:
                goal_x, goal_y = divmod(value - 1, 3)
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance

def is_goal(state):
    return state == goal_state

def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(-1,0), (1,0), (0,-1), (0,1)]
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny <3:
            new_state = copy.deepcopy(state)
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(new_state)
    return neighbors

def a_star_manhattan(initial_state):
    open_list = []
    heapq.heappush(open_list, (heuristic_manhattan(initial_state), 0, initial_state, []))
    visited = set()

    while open_list:
        heuristic, g, current_state, path = heapq.heappop(open_list)
        state_id = str(current_state)
        if state_id in visited:
            continue
        visited.add(state_id)

        if is_goal(current_state):
            return path + [current_state]

        for neighbor in get_neighbors(current_state):
            heapq.heappush(open_list, (heuristic_manhattan(neighbor) + g + 1, g + 1, neighbor, path +
```

```
[current_state]))
    return None

# Initial state example
initial_state = [[2, 8, 3],
                 [1, 6, 4],
                 [7, 0, 5]]

solution = a_star_manhattan(initial_state)

print("\nSolution Steps:\n")
for step in solution:
   for row in step:
       print(row)
    print()
```

Output (Misplaced Tiles):

```
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
SIDDHANT SAHARE 1BM23CS326

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

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

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

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

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

Output (Manhattan Distance):

```
Solution Steps:

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

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

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

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

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

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

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

LAB - 5

Implement hill climbing search algorithm to solve N-queens problem.

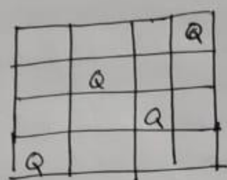
Algorithm

1) Define current state
2) loop until goal state is achieved more operation is applied.
3) Apply an operator
4) Compare new state with goal
5) quit
6) Evaluate new state
7) Compare
8) If new state is close to goal state update current state

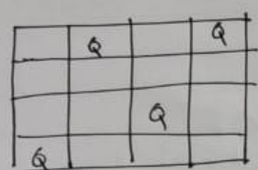
Initial state
$\{ x_0 = 3, x_1 = 1, x_2 = 2, x_3 = 0 \}$

$h = 2$

1)



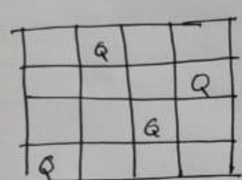
2)



$\{ x_0 = 3, x_1 = 0, x_2 = 2, x_3 = 0 \}$

$h = 1$

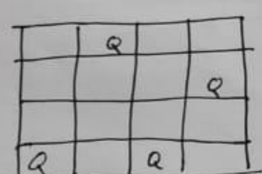
3)



$\{ x_0 = 3, x_1 = 0, x_2 = 2, x_3 = 1 \}$

$h = 1$

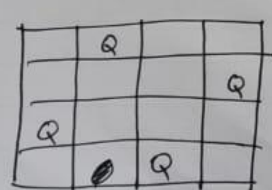
4)



$\{ x_0 = 3, x_1 = 0, x_2 = 3, x_3 = 1 \}$

$h = 1$

5)



$\{ x_0 = 2, x_1 = 0, x_2 = 3, x_3 = 1 \}$

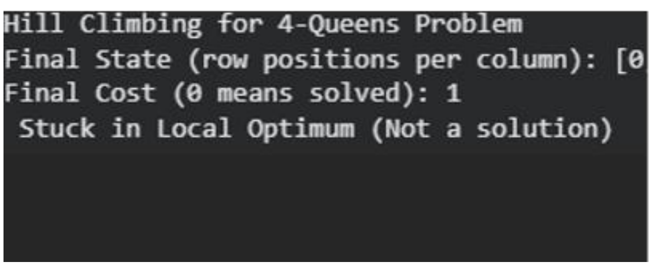
$h = 0$

Code:

```
import random


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]:
                cost += 1
            if abs(state[i] - state[j]) == abs(i - j):
                cost += 1
    return cost

def random_state(n):
    return [random.randint(0, n-1) for _ in range(n)]


def get_best_neighbor(state):
    n = len(state)
    best_state = state[:]
    best_cost = calculate_cost(state)

    for col in range(n):
        for row in range(n):
            if state[col] != row:
                new_state = state[:]
                new_state[col] = row
                new_cost = calculate_cost(new_state)
                if new_cost < best_cost:
                    best_cost = new_cost
                    best_state = new_state
    return best_state, best_cost


def hill_climbing(n):
    current = random_state(n)
    current_cost = calculate_cost(current)

    while True:
        neighbor, neighbor_cost = get_best_neighbor(current)

        if neighbor_cost < current_cost:
            current, current_cost = neighbor, neighbor_cost
        else:
```

```
            break

    return current, current_cost

solution, cost = hill_climbing(4)

print("Name: Siddhant")
print("USN : 1BM23CS326")
print("\nHill Climbing for 4-Queens Problem")
print("Final State (row positions per column):", solution)
print("Final Cost (0 means solved):", cost)
if cost == 0:
    print("Solution Found: Queens are safe!")
else:
    print(" Stuck in Local Optimum (Not a solution)")
```

Output:



```
Hill Climbing for 4-Queens Problem
Final State (row positions per column): [0
Final Cost (0 means solved): 1
 Stuck in Local Optimum (Not a solution)
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:

~~LAB~~ LAB-5

write a program to implement simulated Annealing Algorithm

```
1) current ← initial state
2) T ← a large positive value
3) 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
                                            P = e
       end if
       decrease T
   end while
return current
```

output:

Initial state :[1, 0, 2, 3]

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

step 0 : state:[1,0, 2, 3], cost= 2

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

step 1 : state = [1, 3, 2, 0], cost = 1

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

step 2 : state = [1, 3, 0, 2], cost = 0

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

Solution found at step 2

Final state : [1, 3, 0, 2]

Final cost (number of attacking pairs): 0

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

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]:
                cost += 1
            if abs(state[i] - state[j]) == abs(i - j):
                cost += 1
    return cost


def random_state(n):
    return [random.randint(0, n-1) for _ in range(n)]


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


def simulated_annealing(n, max_steps=100000, start_temp=100, cooling_rate=0.99):
    current = random_state(n)
    current_cost = calculate_cost(current)
    T = start_temp

    for step in range(max_steps):
        if current_cost == 0:
            break
        neighbor = get_neighbor(current)
        neighbor_cost = calculate_cost(neighbor)
        delta = neighbor_cost - current_cost

        if delta < 0 or random.uniform(0, 1) < math.exp(-delta / T):
            current, current_cost = neighbor, neighbor_cost

        T = T * cooling_rate
```

```
        if T < 1e-5:
            break

    return current, current_cost


solution, cost = simulated_annealing(8)

print("Name: Siddhant")
print("USN : 1BM23CS326")
print("\nSimulated Annealing for 8-Queens Problem")
print("Final State (row positions per column):", solution)
print("Final Cost (0 means solved):", cost)
if cost == 0:
    print("Solution Found: Queens are safe!")
else:
    print("Stuck in Local Optimum (Not a solution)")
```

Output:

```
Simulated Annealing for 8-Queens Problem
Final State (row positions per column): [4, 2, 0, 5,
Final Cost (0 means solved): 0
Solution Found: Queens are safe!
```

Program 6

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

Algorithm:

LAB-6

Implementation of truthtable enumeration algorithm for deciding Propositional entailment

Truth table 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	~~true~~ false	false	true	false
true	~~tarue~~ true	false	true	true	true

Algorithm

1. TT- Entails ? (KB, a)

   Purpose : check if the query a is always true given the knowledge base KB

   ·) get all the symbols in KB and a

   ·) Call the helper function TT-Check-all with KB, a, the list of symbols and an empty mod

   ·) Return the result (true or false)

2. TT- Check - ALL (a, symbols, KB, model)

   Purpose : Recursively check all truth assignments for the symbols.

   ·) If no symbol left:

   If KB is true under current model, return whether a is true under the model.

   If KB is false, return true (query holds b default when KB is false).

22/09

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

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

·) Otherwise :

- ·) Pick the first symbol P from the list
- ·) Recursively check with P = true added to the model
- ·) Recursively check with P = false added to the model
- ·) Return true only if both recursive calls return true.

S,T as variable

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

b : $(S \wedge T)$

c : $T \vee \neg T$

write TT

① a entails b

② a entails C

S	T	$\vdash (S \vee T)$ a	b	c
T	T	F	T	T
T	F	F	F	T
F	T	F	F	T
F	F	T	F	T

$a \not\models b$	$a \models C$
F	F
F	F
F	F
F	T

Code:

```
import itertools

def pl_true(expr, model):
    if isinstance(expr, str):
        return model[expr]
    elif isinstance(expr, tuple):
        op = expr[0]
        if op == "not":
            return not pl_true(expr[1], model)
        elif op == "and":
            return pl_true(expr[1], model) and pl_true(expr[2], model)
        elif op == "or":
            return pl_true(expr[1], model) or pl_true(expr[2], model)
        elif op == "implies":
            return (not pl_true(expr[1], model)) or pl_true(expr[2], model)
    return False

def get_symbols(expr):
    if isinstance(expr, str):
        return {expr}
    elif isinstance(expr, tuple):
        return get_symbols(expr[1]) | (get_symbols(expr[2]) if len(expr) > 2 else set())
    return set()

def tt_entails_print(KB, query):
    symbols = list(get_symbols(KB) | get_symbols(query))
    all_models = list(itertools.product([True, False], repeat=len(symbols)))

    entailment = True
    print("\nTruth Table Evaluation:")
    print("-" * 50)
    print("Model".ljust(20), "KB".ljust(10), "Query".ljust(10))
    print("-" * 50)

    for values in all_models:
        model = dict(zip(symbols, values))
        kb_val = pl_true(KB, model)
        q_val = pl_true(query, model)

        print(str(model).ljust(20), str(kb_val).ljust(10), str(q_val).ljust(10))

        if kb_val and not q_val:
            entailment = False

    print("-" * 50)
```

```
    return entailment

KB = ("and", ("implies", "P", "Q"), "P")
query = "Q"

result = tt_entails_print(KB, query)

print("\nName: Siddhant")
print("USN : 1BM23CS326")
print("\nKnowledge Base: (P → Q) and (P)")
print("Query: Q")
print("Does KB entail Query? :", "YES " if result else "NO ")
```

Output:

```
Truth Table:
```

	A	B	C	A or C	B or not C	KB	α
0	False	False	False	False	True	False	False
1	False	False	True	True	False	False	False
2	False	True	False	False	True	False	True
3	False	True	True	True	True	True	True
4	True	False	False	True	True	True	True
5	True	False	True	True	False	False	True
6	True	True	False	True	True	True	True
7	True	True	True	True	True	True	True

```
KB entails α
Siddhant sahare 1BM23CS326
```

Program 7

Implement unification in first order logic:

Algorithm:

LAB-7

Algorithm : Unity (Ψ_1, Ψ_2)

step 1 : If Ψ_1 or Ψ_2 is a variable or constant, then:
a) If Ψ_1 or Ψ_2 are identical, then return NIL.
b) Else if Ψ_1 is a variable,
a) then if Ψ_1 occurs in Ψ_2, then return failure,
b) else return { (Ψ_2 / Ψ_1) }.
c) Else if Ψ_2 is a variable,
a) If Ψ_2 occurs in Ψ_1 then return Failure,
b) Else return { (Ψ_1 / Ψ_2) }.
d) Else return Failure

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

step 3: If Ψ_1 and Ψ_2 have a different number of arguement, then return failure.

step 4: Set substitution set (subst) to NIL.

step 5: For i = 1 to the number of elements in Ψ_1.
a) Call unify function with the ith element of Ψ_1 and ith element of Ψ_2 and put the result into S.
b) If S = failure then returns failure
c) If S ≠ NIL then do,
a) apply S to the remainder to both L1 and L2
b) subst = Append (S, subst)

Step 6: Return Subst.

1) Unify {p(b, x, f(g(z))) and p(z, f(y), f(y))}

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

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

θ = b/z

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

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

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

{p(z, x, x}

θ = f(y)/x

{p(z, x, f(y))} g(z)/y

{p(z, x, x)}

{p(z, x, x)}

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

θ = y/John

Unify (knows(John, x), knows(John, bill))

θ = x/Bill

Unify knows(John, bill), knows(John, Bill)

3) Unify(prime(11) and prime(y)}

θ = y/11

{prime(11) and prime(11)}

4) Unify knows(John, x), knows(y, mother(y))}

θ = y/John

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

θ = x, mother(John)

Unify {knows(John, mother(John)), knows(John, mother(John))}

5) Unify {p(f(a), g(y)), p(x, x)}

failure

Code:

```
import json

# --- Helper Functions for Term Manipulation ---

def is_variable(term):
    """Checks if a term is a variable (a single capital letter string)."""
    return isinstance(term, str) and len(term) == 1 and 'A' <= term[0] <= 'Z'

def occurs_check(variable, term, sigma):
    """
    Checks if 'variable' occurs anywhere in 'term' under the current substitution 'sigma'.
    This prevents infinite recursion (e.g., unifying X with f(X)).
    """
    term = apply_substitution(term, sigma) # Check the substituted term

    if term == variable:
        return True

    # If the term is a list (function/predicate), check its arguments recursively
    if isinstance(term, list):
        for arg in term[1:]:
            if occurs_check(variable, arg, sigma):
                return True

    return False

def apply_substitution(term, sigma):
    """
    Applies the current substitution 'sigma' to a 'term' recursively.
    """
    if is_variable(term):
        # If the variable is bound in sigma, apply the binding
        if term in sigma:
            # Recursively apply the rest of the substitutions to the binding's value
            # This is critical for chains like X/f(Y), Y/a -> X/f(a)
            return apply_substitution(sigma[term], sigma)
        return term

    if isinstance(term, list):
        # Apply substitution to the arguments of the function/predicate
        new_term = [term[0]] # Keep the function/predicate symbol
        for arg in term[1:]:
            new_term.append(apply_substitution(arg, sigma))
        return new_term
```

```
    # Term is a constant or an unhandled type, return as is
    return term

def term_to_string(term):
    """
    Converts the internal list representation of a term into standard logic notation string.
    e.g., ['f', 'Y'] -> "f(Y)"
    """
    if isinstance(term, str):
        return term

    if isinstance(term, list):
        # Term is a function or predicate
        symbol = term[0]
        args = [term_to_string(arg) for arg in term[1:]]
        return f"{symbol}({', '.join(args)})"

    return str(term)


# --- Main Unification Function ---

def unify(term1, term2):
    """
    Implements the Unification Algorithm to find the MGU for term1 and term2.
    Returns the MGU as a dictionary or None if unification fails.
    """
    # Initialize the substitution set (MGU)
    sigma = {}

    # Initialize the list of pairs to resolve (the difference set)
    diff_set = [[term1, term2]]

    print(f"--- Unification Process Started ---")
    print(f"Initial Terms:")
    print(f"L1: {term_to_string(term1)}")
    print(f"L2: {term_to_string(term2)}")
    print("-" * 35)

    while diff_set:
        # Pop the current pair of terms to unify
        t1, t2 = diff_set.pop(0)

        # 1. Apply the current MGU to the terms before comparison
        t1_prime = apply_substitution(t1, sigma)
        t2_prime = apply_substitution(t2, sigma)
```

```
    print(f"Attempting to unify: {term_to_string(t1_prime)} vs {term_to_string(t2_prime)}")

    # 2. Check if terms are identical
    if t1_prime == t2_prime:
      print(f"  -> Identical. Current MGU: {term_to_string(sigma)}")
      continue

    # 3. Handle Variable-Term unification
    if is_variable(t1_prime):
      var, term = t1_prime, t2_prime
    elif is_variable(t2_prime):
      var, term = t2_prime, t1_prime
    else:
      var, term = None, None

    if var:
      # Check if term is a variable, and if so, don't bind V/V
      if is_variable(term):
         print(f"  -> Both are variables. Skipping {var} / {term}")
         # Ensure they are added back if not identical (which is caught by step 2).
         # If V1 != V2, we add V1/V2 or V2/V1 to sigma. Since step 2 handles V/V, this means V1
!= V2 here.
         if var != term:
            sigma[var] = term
            print(f"  -> Variable binding added: {var} / {term_to_string(term)}. New MGU:
{term_to_string(sigma)}")
      # Occurs Check: Fail if the variable occurs in the term it's being bound to
      elif occurs_check(var, term, sigma):
        print(f"  -> OCCURS CHECK FAILURE: Variable {var} occurs in
{term_to_string(term)}")
        return None

      # Create a new substitution {var / term}
      else:
        sigma[var] = term
        print(f"  -> Variable binding added: {var} / {term_to_string(term)}. New MGU:
{term_to_string(sigma)}")

    # 4. Handle Complex Term (Function/Predicate) unification
    elif isinstance(t1_prime, list) and isinstance(t2_prime, list):
      # Check functor/predicate symbol and arity (number of arguments)
      if t1_prime[0] != t2_prime[0] or len(t1_prime) != len(t2_prime):
        print(f"  -> FUNCTOR/ARITY MISMATCH: {t1_prime[0]} != {t2_prime[0]} or arity
mismatch.")
        return None
```

```
            # Add corresponding arguments to the difference set
            # Start from index 1 (after the symbol)
            for arg1, arg2 in zip(t1_prime[1:], t2_prime[1:]):
                diff_set.append([arg1, arg2])
            print(f"  -> Complex terms matched. Adding arguments to difference set.")

        # 5. Handle Constant-Constant or other mismatches (Fail)
        else:
            print(f"  -> TYPE/CONSTANT MISMATCH: {term_to_string(t1_prime)} and
{term_to_string(t2_prime)} cannot be unified.")
            return None

    print("-" * 35)
    print("--- Unification Successful ---")

    # Final cleanup to ensure all bindings are fully resolved
    final_mgu = {k: apply_substitution(v, sigma) for k, v in sigma.items()}
    return final_mgu

# --- Define the Input Terms ---

# L1 = Q(a, g(X, a), f(Y))
literal1 = ['Q', 'a', ['g', 'X', 'a'], ['f', 'Y']]

# L2 = Q(a, g(f(b), a), X)
literal2 = ['Q', 'a', ['g', ['f', 'b'], 'a'], 'X']

# --- Run the Unification ---

mgu_result = unify(literal1, literal2)

if mgu_result is not None:
    print("\n[ Final MGU Result ]")

    # Format the final MGU for display using the new helper function
    clean_mgu = {k: term_to_string(v) for k, v in mgu_result.items()}
    final_output = ', '.join([f"{k} / {v}" for k, v in clean_mgu.items()])
    print(f"Final MGU: {{ {final_output} }}")

    # --- Verification ---
    print("\n[ Verification ]")
    unified_l1 = apply_substitution(literal1, mgu_result)
    unified_l2 = apply_substitution(literal2, mgu_result)

    print(f"L1 after MGU: {term_to_string(unified_l1)}")
    print(f"L2 after MGU: {term_to_string(unified_l2)}")
```

```
    if unified_l1 == unified_l2:
        print("-> SUCCESS: L1 and L2 are identical after applying the MGU.")
    else:
        print("-> ERROR: Unification failed verification.")
else:
    print("\nUnification FAILED.")

print("Siddhant 1BM23CS326")
```

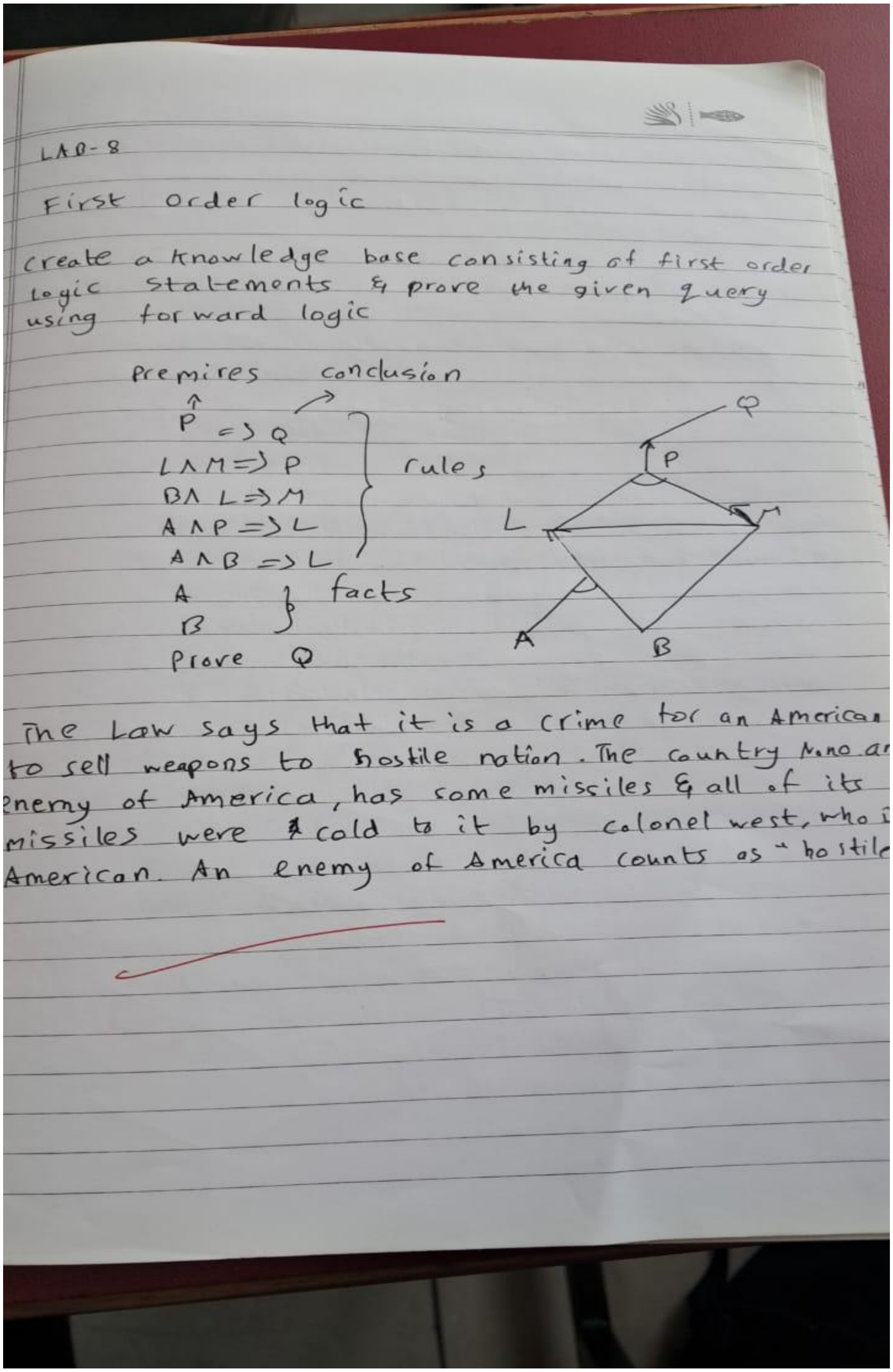
Output:

```
--- Unification Process Started ---
Initial Terms:
L1: Q(a, g(X, a), f(Y))
L2: Q(a, g(f(b), a), X)
-----------------------------------
Attempting to unify: Q(a, g(X, a), f(Y)) vs Q(a, g(f(b), a), X)
  -> Complex terms matched. Adding arguments to difference set.
Attempting to unify: a vs a
  -> Identical. Current MGU: {}
Attempting to unify: g(X, a) vs g(f(b), a)
  -> Complex terms matched. Adding arguments to difference set.
Attempting to unify: f(Y) vs X
  -> Variable binding added: X / f(Y). New MGU: {'X': ['f', 'Y']}
Attempting to unify: f(Y) vs f(b)
  -> Complex terms matched. Adding arguments to difference set.
Attempting to unify: a vs a
  -> Identical. Current MGU: {'X': ['f', 'Y']}
Attempting to unify: Y vs b
  -> Variable binding added: Y / b. New MGU: {'X': ['f', 'Y'], 'Y': 'b'}
-----------------------------------
--- Unification Successful ---

[ Final MGU Result ]
Final MGU: { X / f(b), Y / b }

[ Verification ]
L1 after MGU: Q(a, g(f(b), a), f(b))
L2 after MGU: Q(a, g(f(b), a), f(b))
-> SUCCESS: L1 and L2 are identical after applying the MGU.
Siddhant sahare 1BM23CS326
```

Program 8

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

Algorithm:



LAB-8

First order logic

create a knowledge base consisting of first order logic statements & prove the given query using forward logic

premires conclusion

$P \Rightarrow Q$

$L \wedge M \Rightarrow P$

$B \wedge L \Rightarrow M$

$A \wedge P \Rightarrow L$

$A \wedge B \Rightarrow L$

rules

A

B

facts

Prove Q

The Law says that it is a crime for an American to sell weapons to hostile nation. The country Nono an enemy of America, has some missiles & all of its missiles were sold to it by colonel west, who i American. An enemy of America counts as "hostile

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
      for each θ such that subst(θ, p1 ∧ ... ∧ pn)
                  = subst(θ, p1' ∧ ... ∧ pn')
          for some p1' ... pn' in KB
      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
```

13.10

Code :

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

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

        # For each rule in KB
        for rule in KB:
            # Standardize the rule variables to avoid conflicts
            standardized_rule = standardize_variables(rule)
            p1_to_pn, q = standardized_rule  # Premises (p1, ..., pn) and conclusion (q)

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

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

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

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

    return False  # Return false if no substitution is found


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


def get_matching_substitutions(premises, KB):
    """
    Get matching substitutions for the premises in the KB.
```

```
    This is a placeholder to represent how substitutions would be found.
    """
    # Implement substitution matching here
    return []  # This should return a list of valid substitutions


def apply_substitution(theta, expression):
    """
    Apply a substitution θ to an expression.
    This function will replace variables in expression with their corresponding terms from θ.
    """
    if isinstance(expression, str) and expression.startswith('?'):
        return theta.get(expression, expression)  # Apply substitution to variable
    elif isinstance(expression, tuple):
        return tuple(apply_substitution(theta, arg) for arg in expression)
    return expression


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

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

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

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

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

```
        return {**subst, _psi2: _psi1}
    else:
        return 'FAILURE'

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

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

  return subst

# Knowledge Base (KB)
KB = set()

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

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

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

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

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

Output:

```
Inferred: ('Weapon', 'm1') from [('Missile', 'm1')]
Inferred: ('Criminal', 'Robert') from [('American', 'Robert'), ('Weapon', 'm1'), ('Hostile', 'CountryA'), ('Sells', 'Rober
t', 'CountryA', 'm1')]

Final facts:
('Missile', 'm1')
('Weapon', 'm1')
('Criminal', 'Robert')
('Sells', 'Robert', 'CountryA', 'm1')
('American', 'Robert')
('Hostile', 'CountryA')

Conclusion: Robert is criminal.
Siddhant sahare 1BM23CS326
```

Program 9

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

Algorithm:

7/10/25

Week - 9 First order logic Resolution

1. Eliminate biconditionals and implications:
   - Eliminate $\Leftrightarrow$ replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
   - Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg \alpha \vee \beta$

2. move $\neg$ inwards.
   - $\neg (\forall x\ P) \equiv \exists x\ \neg P$
   - $\neg (\exists x\ P) \equiv \forall x\ \neg P$
   - $\neg (\alpha \vee \beta) \equiv \neg \alpha \wedge \neg \beta$
   - $\neg (\alpha \wedge \beta) \equiv \neg \alpha \vee \neg \beta$
   - $\neg \neg \alpha \equiv \alpha$

3. Standardize variables apart by renaming them: each quantifier should use a different variable

4. Skolemize each existential variable is replaced by a skolem should use a difference variable
   - For instance $\exists x\ Rich(x)$ becomes $Rich(G1)$ where $G1$ is a skolem constant.
   - Everyone has 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)

5. Drop universal quatifiers
   - For instance, $\forall x\ Person(x)$ becomes $person(x)$

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

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

Proof of resolution (output)

Given KB

John likes all kind of food

Apple and vegetables are food

Anything anyone eats and not killed is food

Anil eats peanuts and still alive

Harry eats every thing that anil eats

Anyone who is alive implies not killed

Anyone who is not killed implies alive

John likes peanuts

a) $\forall x: food(x) \rightarrow likes(John, x)$

b) $food(apple) \wedge food(vegetables)$

c) $\forall x \forall y: eats(x,y) \wedge \neg killed(x) \rightarrow food(y)$

d) $\forall$ $eats(Anil, Peanuts) \wedge alive(Anil)$

e) $\forall x: eats(Anil, x) \rightarrow eats(Harry, x)$

f) $\forall x: \neg killed(x) \rightarrow alive(x)$

g) $\forall x: alive(x) \rightarrow \neg killed(x)$

h) $likes(John, peanuts$

1) Convert all sentences to CNF

2) Negate conclusion S & convert result to CNF

3) Add negated conclusion S to premise clauses

4) Repeat until contradition or no progress is made:
   a) Select 2 clauses (call them parent clauses)
   b) Resolve them together, performing all required unification
   c) If resolvent is empty clauses a contradiction has been found (i.e S follows from premises)
   d) If not add resolvent to premises

Proof of resolution (output)
Given KB
John likes all kind of food
Apple and vegetables are food
Anything anyone eats and ~~still alive~~ not killed is food
Anil eats peanuts and still alive
Mary eats everything that anil eats
Anyone who is alive implies not killed
Anyone who is not killed implies alive
John likes peanuts.

a) $\forall x : food(x) \rightarrow likes(John, x)$
b) $food(apple) \wedge food(vegetables)$
c) $\forall x \forall y : eats(x, y) \wedge \neg killed(x) \rightarrow food(y)$
d) $eats(Anil, Peanuts) \wedge alive(Anil)$
e) $\forall z : eats(Anil, x) \rightarrow eats(Harry, x)$

f) ∀x : ¬ killed (x) → alive (x)
g) ∀x : alive (x) → ¬ killed (x)
h) likes (John, peanuts)

Proof by resolution
a) ¬ food (x) ∨ likes (John, x)
b) food (Apple)
c) food (vegatables)
d) ¬ eats (y, z) ∨ killed (y) ∨ food (z)
e) eats (Anil, peanuts)
f) alive (Anil)
g) ¬ eats (Anil, w) ∨ eats (Harry, w)
h) killed (g) ∨ alive (g)
i) killed (g) ∨ alive (g)
i) likes (John, peanuts)

¬likes (John, Peanuts)   ¬ food (x) ∨ likes (John, x)

¬ food (Peanuts)   ¬ eats (y, z) ∨ killed (y) ∨ food (z) {peanuts/z}

¬ eats (peanuts) ∨ killed (y)   eats (Anil, peanuts)

killed (Anil)   ¬ alive (k) ∨ ¬ killed (k) [Anil/k]

¬ (alive (Anil))   (alive (Anil))

{ } Hence proved.

Code:

```
from itertools import combinations

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

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

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

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

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

def resolve(clause1, clause2):
    resolvents = []
```

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

def resolution(kb, query):
    negated_query = frozenset({negate(query)})
    clauses = kb | {negated_query}
    new = set()

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

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

# Run resolution
result = resolution(kb, query)

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

Output:

```
Proved by resolution: John likes peanuts.
```

Program 10

Implement Alpha-Beta Pruning

Week 10 (Alpha beta pruning)

function Alpha-Beta-search (state) returns an action

V ← Max-value (state, −∞, +∞)

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 ← −∞

for each a in Actions (state) do

v ← Max (v, Min value (Result (s, a), α, β))

if v ≥ β then return v

α ← Max (α, v)

return v

function Min_value (state, α, β) return a utility value

if Terminal_test (state) then return utility

v ← +∞

for each a in Actions (state) do

v ← MIN (v, Max-value (Result (s, a) α, β))

if v ≤ α then return v

β ← Min (β, v)

return v

Output

Enter Max depth of tree: 3

For depth 3 tree will have 8 roots

leaf 1: 10

leaf 2: 9

leaf 3: 14

leaf 4: 18

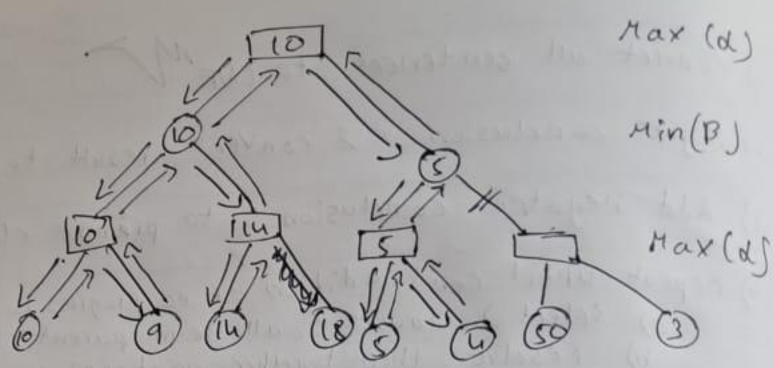
leaf 5: 5

leaf 6: 40

leaf 7: 50

leaf 8: 3

10/11



Best value for root Max (10)
Total moves 11

Code:

```
import math

def alpha_beta_search(state):
    return max_value(state, -math.inf, math.inf)

def max_value(state, alpha, beta):
    if terminal_test(state):
        return utility(state)
    v = -math.inf
    for a in actions(state):
        v = max(v, min_value(result(state, a), alpha, beta))
        if v >= beta:
            return v
        alpha = max(alpha, v)
    return v

def min_value(state, alpha, beta):
    if terminal_test(state):
        return utility(state)
    v = math.inf
    for a in actions(state):
        v = min(v, max_value(result(state, a), alpha, beta))
        if v <= alpha:
            return v
        beta = min(beta, v)
    return v

values = [3, 5, 6, 9, 1, 2, 0, -1]
max_depth = 3

def terminal_test(state):
    return state >= len(values) // 2**(max_depth - depth[state])

def utility(state):
    return values[state]

def actions(state):
    if depth[state] == max_depth:
        return []
    return [0, 1]
```

```
def result(state, action):
    child = state * 2 + 1 + action
    depth[child] = depth[state] + 1
    return child

depth = {0: 0}
print("Best value for maximizer:", alpha_beta_search(0))
print("SIDDHANT / 1BM23CS326")
```

Output :

```
🧠 ALPHA-BETA PRUNING — Interactive Demo
==========================================

Enter maximum depth of the game tree: 3
For depth 3, the tree will have 8 leaf nodes.

Enter the leaf node values from LEFT to RIGHT:
Value of leaf 1: 3
Value of leaf 2: 5
Value of leaf 3: 6
Value of leaf 4: 9
Value of leaf 5: 1
Value of leaf 6: 22
Value of leaf 7: 0
Value of leaf 8: -1

🔁 Running Alpha-Beta pruning...

MAX: Depth=2, Node=3, Alpha=3, Beta=inf
MAX: Depth=2, Node=3, Alpha=5, Beta=inf
MIN: Depth=1, Node=1, Alpha=-inf, Beta=5
MAX: Depth=2, Node=4, Alpha=6, Beta=5
🚫 PRUNED at MAX node 4 (α ≥ β)
MIN: Depth=1, Node=1, Alpha=-inf, Beta=5
MAX: Depth=0, Node=0, Alpha=5, Beta=inf
MAX: Depth=2, Node=5, Alpha=5, Beta=inf
MAX: Depth=2, Node=5, Alpha=22, Beta=inf
MIN: Depth=1, Node=2, Alpha=5, Beta=22
MAX: Depth=2, Node=6, Alpha=5, Beta=22
MAX: Depth=2, Node=6, Alpha=5, Beta=22
MIN: Depth=1, Node=2, Alpha=5, Beta=0
🚫 PRUNED at MIN node 2 (α ≥ β)
MAX: Depth=0, Node=0, Alpha=5, Beta=inf

✅ Final Result:
Value of the root node (best achievable for MAX): 5
```