

**VISVESVARAYA TECHNOLOGICAL
UNIVERSITY**

“JnanaSangama”, Belgaum -590014, Karnataka.



**LAB REPORT
on**

Artificial Intelligence (23CS5PCAIN)

Submitted by

Siddharth Arya (1BM23CS328)

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 **Siddharth Arya (1BM23CS328)**, 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

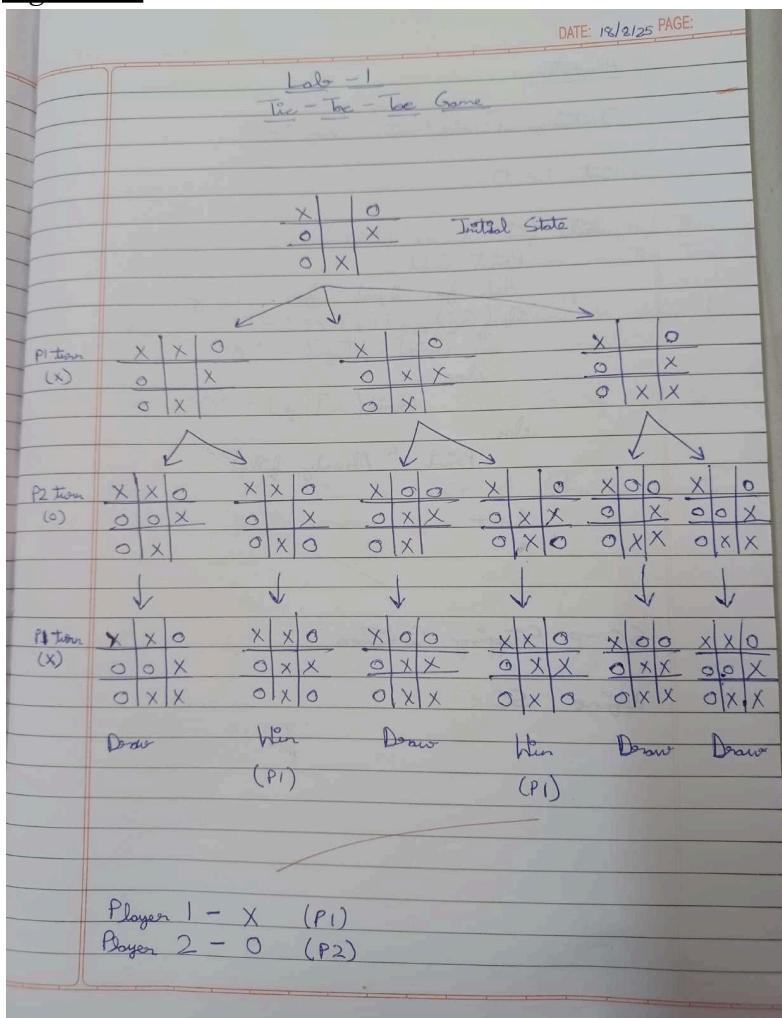
Sl. No.	Date	Experiment Title	Page No.
1	25-08-2025	Implement Tic –Tac –Toe Game Implement vacuum cleaner agent	4
2	01-09-2025	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	10
3	08-09-2025	Implement A* search algorithm	17
4	15-09-2025	Implement Hill Climbing search algorithm to solve N-Queens problem	21
5	15-09-2025	Simulated Annealing to Solve 8-Queens problem	25
6	22-09-2025	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	28
7	13-10-2025	Implement unification in first order logic	34
8	13-10-2025	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	38
9	27-10-2025	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	44
10	27-10-2025	Implement Alpha-Beta Pruning.	51

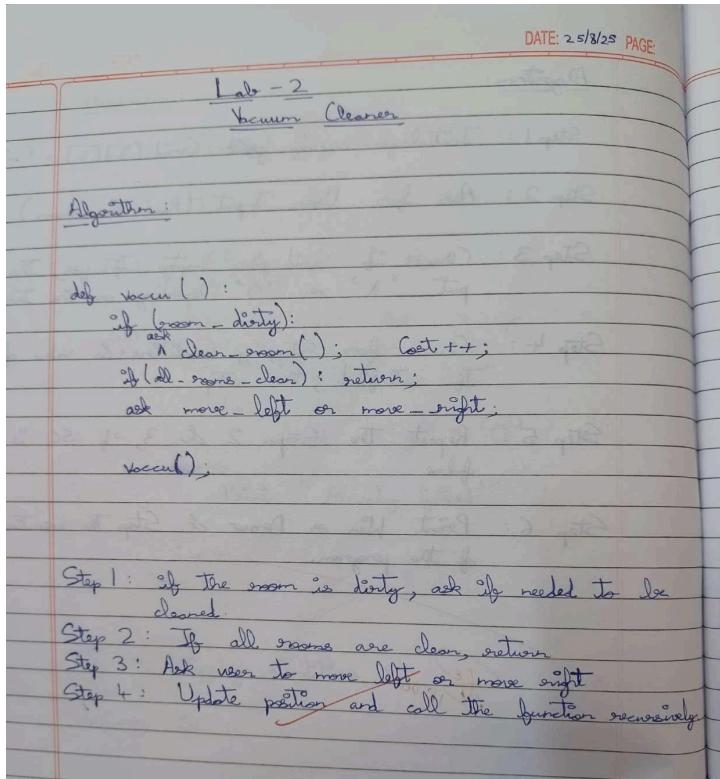
Github Link:
https://github.com/Sid-CS328/1BM23CS328_AI_Lab

Program 1

Implement Tic – Tac – Toe Game

Algorithm:





Code:

Tic-Tac-Toe game

```
import random
```

```
board = [' ' for _ in range(9)]
```

```
current_winner = None
```

```
def print_board():
```

```

    print(f" {board[0]} | {board[1]} | {board[2]} ")
    print(" ---+---+---")
    print(f" {board[3]} | {board[4]} | {board[5]} ")
    print(" ---+---+---")
    print(f" {board[6]} | {board[7]} | {board[8]} ")

```

```

def check_winner(player):
    win_conditions = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],
        [0, 3, 6], [1, 4, 7], [2, 5, 8],
        [0, 4, 8], [2, 4, 6]
    ]

    for condition in win_conditions:
        if all(board[i] == player for i in condition):
            return True
    return False

def check_draw():
    return ' ' not in board

def available_moves():
    return [i for i, spot in enumerate(board) if spot == ' ']

def make_move(position, player):
    if board[position] == ' ':
        board[position] = player
        return True
    return False

def computer_move():
    return random.choice(available_moves())

def play_game():
    global current_winner

    print_board()

    user_player = 'X'
    computer_player = 'O'

    while True:
        user_move = int(input("Enter your move (0-8): "))
        if make_move(user_move, user_player):
            print("Player (X) moves:")

```

```

print_board()

if check_winner(user_player):
    current_winner = user_player
    print("You win!")
    break

if check_draw():
    print("It's a draw!")
    break

computer_move_position = computer_move()
make_move(computer_move_position, computer_player)
print("Computer's move:")
print_board()

if check_winner(computer_player):
    current_winner = computer_player
    print("Computer wins!")
    break

if check_draw():
    print("It's a draw!")
    break
else:
    print("Invalid move. Try again.")

if __name__ == "__main__":
    play_game()

# Vacuum Cleaner

def vacuum_cleaner(rooms, position, cost):
    # Base case: If the vacuum is at either end, stop
    if position < 0 or position >= N:
        return cost

    print("At", position);

```

```

if not rooms[position]:
    rooms[position] = True
    cost += 1
    print("Room Cleaned")

c = input("Move Right?");
if(c == 'y'):
    return vacuum_cleaner(rooms, position + 1, cost)

c = input("Move Left?");
if(c == 'y'):
    return vacuum_cleaner(rooms, position - 1, cost)

#cost = vacuum_cleaner(rooms, position - 1, cost)
#cost = vacuum_cleaner(rooms, position + 1, cost)
return cost

N = int(input())
rooms = [bool(int(x)) for x in input().split()] # Read rooms as booleans
initial_pos = int(input())

total_cost = vacuum_cleaner(rooms, initial_pos, 0)
print("Cost to clean:", total_cost)
print("Siddharth Arya, 1BM23CS328")

```

Output:

```
Name: Siddharth Arya
```

```
USN: 1BM23CS328
```

```
| |
```

```
--+---+--
```

```
| |
```

```
--+---+--
```

```
| |
```

```
Enter your move (0-8): 0
```

```
Player (X) moves:
```

```
X | |
```

```
--+---+--
```

```
| |
```

```
--+---+--
```

```
| |
```

```
Computer's move:
```

```
X | |
```

```
--+---+--
```

```
| |
```

```
--+---+--
```

```
| | 0
```

```
Enter your move (0-8): 1
```

```
Player (X) moves:
```

```
X | X |
```

```
--+---+--
```

```
| |
```

```
--+---+--
```

```
| | 0
```

```
Computer's move:
```

```
X | X |
```

```
--+---+--
```

```
| | 0
```

```
--+---+--
```

```
| | 0
```

```
Enter your move (0-8): 2
```

```
Player (X) moves:
```

```
X | X | X
```

```
--+---+--
```

```
| | 0
```

```
--+---+--
```

```
| | 0
```

```
You win!
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm

Algorithm:

DATE: PAGE:

3(b)

DFS without heuristic approach

Algorithm :

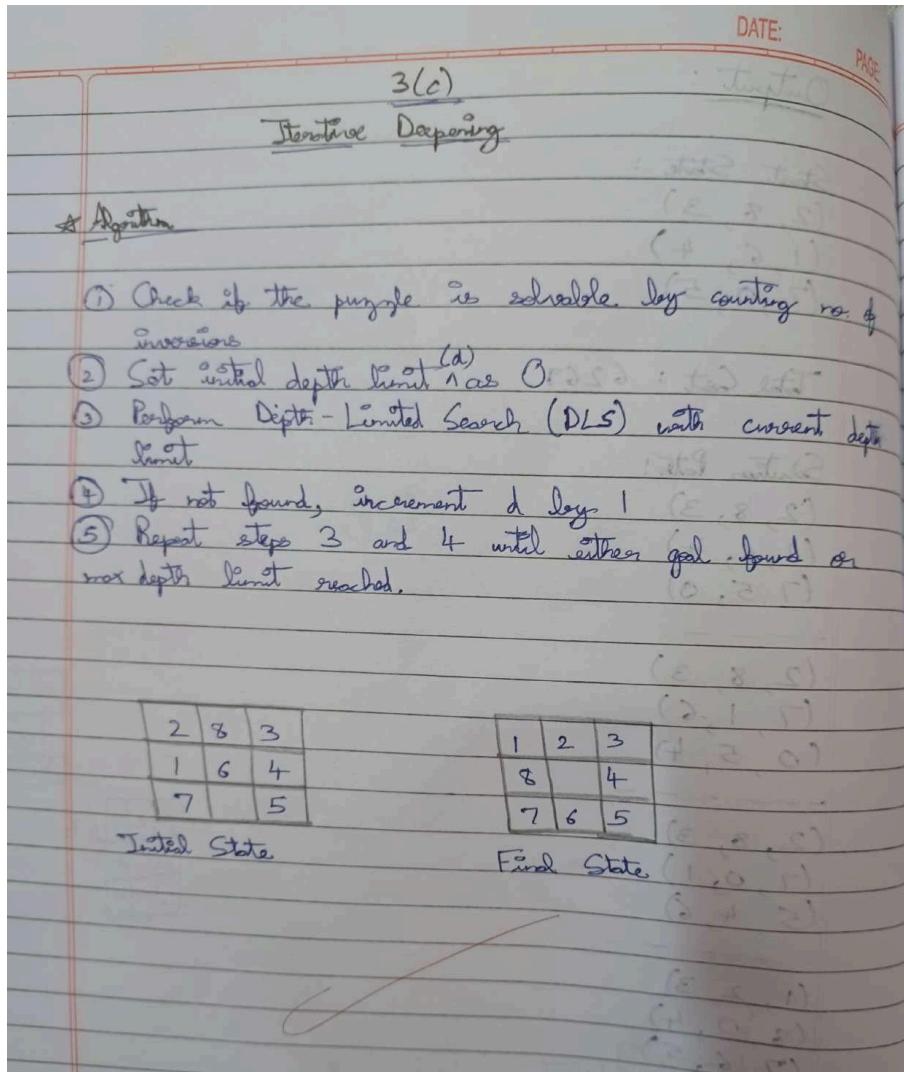
- ① Start by initializing a grid $[3][3]$ array based on the initial state of the 3-puzzle grid.
- ② Initialise a visited array, which is 3D, and stores all the visited grids using memoization.
- ③ Use DFS & stack to solve the problem.
- ④ Use Recursion & backtracking to move the blocks all in 4 directions: Up, Down, Left, Right.
- ⑤ Once goal state reached, stop the program and return.

2	8	3
1	6	4
7	5	

Initial

1	2	3
8		4
7	6	5

Goal



Code:

8-puzzle problem using DFS

```
from collections import deque
```

```
goal_state = (1, 2, 3, 8, 0, 4, 7, 6, 5)
```

```
valid_moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
```

```
def is_valid_move(position, move, size=3):
```

```

row, col = position
new_row, new_col = row + move[0], col + move[1]
return 0 <= new_row < size and 0 <= new_col < size

def get_new_state(state, position, move):
    row, col = position
    new_row, new_col = row + move[0], col + move[1]
    new_state = list(state)
    new_pos = new_row * 3 + new_col
    blank_pos = row * 3 + col
    new_state[blank_pos], new_state[new_pos] = new_state[new_pos], new_state[blank_pos]
    return tuple(new_state)

def dfs(start_state):
    stack = [(start_state, start_state.index(0))]
    visited = set()
    visited.add(start_state)
    parent_map = {start_state: None}
    move_map = {start_state: None}

    total_cost = 1

    visited_states = []

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

        visited_states.append(current_state)

        if current_state == goal_state:
            path = []
            while current_state != start_state:
                path.append(move_map[current_state])
                current_state = parent_map[current_state]
            return path[::-1], visited_states, total_cost
        row, col = divmod(blank_pos, 3)

        for move in valid_moves:
            if is_valid_move((row, col), move):
                new_state = get_new_state(current_state, (row, col), move)

```

```

        if new_state not in visited:
            visited.add(new_state)
            stack.append((new_state, new_state.index(0)))
            parent_map[new_state] = current_state
            move_map[new_state] = move
            total_cost += 1

    return None, visited_states, total_cost

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

start_state = (2, 8, 3, 1, 6, 4, 7, 0, 5)
print("Start State:")
print_state(start_state)

path, visited_states, total_cost = dfs(start_state)

if path:
    print("\nSolution Path:")
    current_state = start_state
    for move in path:
        print(f'Move blank {move} from {current_state}')
        row, col = divmod(current_state.index(0), 3)
        current_state = get_new_state(current_state, (row, col), move)
        print_state(current_state)
else:
    print("\nNo solution found.")

print("\nVisited States:")
for state in visited_states:
    print_state(state)
    print("----")

print(f"\nTotal Cost (Visited States): {total_cost}")

print("Name: Siddharth Arya")
print("USN: 1BM23CS328")

```

```

#8-puzzle using IDS

from collections import deque

goal_state = (1, 2, 3, 8, 0, 4, 7, 6, 5)

valid_moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

def is_valid_move(position, move, size=3):
    row, col = position
    new_row, new_col = row + move[0], col + move[1]
    return 0 <= new_row < size and 0 <= new_col < size

def get_new_state(state, position, move):
    row, col = position
    new_row, new_col = row + move[0], col + move[1]
    new_state = list(state)
    new_pos = new_row * 3 + new_col
    blank_pos = row * 3 + col
    new_state[blank_pos], new_state[new_pos] = new_state[new_pos], new_state[blank_pos]
    return tuple(new_state)

def depth_limited_dfs(start_state, depth_limit, visited_states):
    stack = [(start_state, start_state.index(0), 0)]
    visited = set()
    visited.add(start_state)

    while stack:
        current_state, blank_pos, current_depth = stack.pop()

        visited_states.append(current_state)

        if current_state == goal_state:
            return True

        if current_depth < depth_limit:
            row, col = divmod(blank_pos, 3)

```

```

for move in valid_moves:
    if is_valid_move((row, col), move):
        new_state = get_new_state(current_state, (row, col), move)
        if new_state not in visited:
            visited.add(new_state)
            stack.append((new_state, new_state.index(0), current_depth + 1))

return False

def iterative_deepening_search(start_state):
    depth = 0
    visited_states = []
    total_cost = 0

    while True:
        print(f"Searching with depth limit {depth}...")
        if depth_limited_dfs(start_state, depth, visited_states):
            print("Goal found!")
            break
        depth += 1

    total_cost = len(visited_states)

    return visited_states, total_cost

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

start_state = (2, 8, 3, 1, 6, 4, 7, 0, 5)
print("Start State:")
print_state(start_state)

visited_states, total_cost = iterative_deepening_search(start_state)

print(f"\nTotal Cost (Visited States): {total_cost}")
print("Name: Siddharth Arya")
print("USN: 1BM23CS328")

```

```

print("\nAll visited states:")
for state in visited_states:
    print_state(state)
    print("----")

```

Output:

<pre> Start State: (2, 8, 3) (1, 6, 4) (7, 0, 5) Total Cost (Visited States): 6269 Name: Siddharth Arya USN: 1BM23CS328 Solution Path: Move blank (0, 1) from (2, 8, 3, 1, 6, 4, 7, 0, 5) (2, 8, 3) (1, 6, 4) (7, 5, 0) Move blank (-1, 0) from (2, 8, 3, 1, 6, 4, 7, 5, 0) (2, 8, 3) (1, 6, 0) (7, 5, 4) Move blank (0, -1) from (2, 8, 3, 1, 6, 0, 7, 5, 4) (2, 8, 3) (1, 0, 6) (7, 5, 4) Move blank (0, -1) from (2, 8, 3, 1, 0, 6, 7, 5, 4) (2, 8, 3) (0, 1, 6) (7, 5, 4) Move blank (1, 0) from (2, 8, 3, 0, 1, 6, 7, 5, 4) (2, 8, 3) (7, 1, 6) (0, 5, 4) </pre>	<pre> Start State: (2, 8, 3) (1, 6, 4) (7, 0, 5) Searching with depth limit 0... Searching with depth limit 1... Searching with depth limit 2... Searching with depth limit 3... Searching with depth limit 4... Searching with depth limit 5... Goal found! Total Cost (Visited States): 126 Name: Siddharth Arya USN: 1BM23CS328 All visited states: (2, 8, 3) (1, 6, 4) (7, 0, 5) ---- (2, 8, 3) (1, 6, 4) (7, 0, 5) ---- (2, 8, 3) (1, 6, 4) (7, 5, 0) ---- (2, 8, 3) (1, 6, 4) (7, 5, 4) ---- (2, 8, 3) (1, 6, 4) (0, 7, 5) ---- (2, 8, 3) (1, 0, 4) (7, 6, 5) ---- (2, 8, 3) (1, 6, 4) (7, 0, 5) </pre>
---	---

Program 3

Implement A* search algorithm

Algorithm:

DATE: PAGE:

5 (b)

A* algorithm using Manhattan Distance

1	5	8
3	2	.
4	6	7

Initial State

1	2	3
4	5	6
7	8	.

Final State

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

depth value → heuristic value

* Algorithm

- ① Set initial depth (g) as 0.
- ② Perform the possible iterations by moving the blank tile (U, D, L, R).
- ③ Use the formula given for $f(n)$.
- ④ Perform Steps 2 and 3 to the iteration with the lowest $f(n)$ value.
- ⑤ Point out the final set of goal state reached, otherwise return -1.

* Here, the heuristic used is Manhattan Distance

Code:

A* algorithm using Manhattan distance

```
import heapq
```

```
GOAL_STATE = (1, 2, 3, 8, 0, 4, 7, 6, 5)
```

```
NEIGHBORS = {
    0: [1, 3], 1: [0, 2, 4], 2: [1, 5],
```

```

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

def manhattan_distance(state):
    distance = 0
    for i, tile in enumerate(state):
        if tile == 0:
            continue
        goal_pos = GOAL_STATE.index(tile)
        x1, y1 = divmod(i, 3)
        x2, y2 = divmod(goal_pos, 3)
        distance += abs(x1 - x2) + abs(y1 - y2)
    return distance

def get_neighbors(state):
    zero_index = state.index(0)
    neighbors = []
    for swap_index in NEIGHBORS[zero_index]:
        new_state = list(state)
        new_state[zero_index], new_state[swap_index] = new_state[swap_index], new_state[zero_index]
        neighbors.append(tuple(new_state))
    return neighbors

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

def a_star_manhattan(start_state):
    open_set = []
    heapq.heappush(open_set, (manhattan_distance(start_state), 0, start_state))
    came_from = {}
    g_score = {start_state: 0}
    closed_set = set()

```

```

while open_set:
    f, g, current = heapq.heappop(open_set)

    if current == GOAL_STATE:
        return reconstruct_path(came_from, current)

    closed_set.add(current)

    for neighbor in get_neighbors(current):
        if neighbor in closed_set:
            continue

        tentative_g = g + 1

        if tentative_g < g_score.get(neighbor, float('inf')):
            came_from[neighbor] = current
            g_score[neighbor] = tentative_g
            f_score = tentative_g + manhattan_distance(neighbor)
            heapq.heappush(open_set, (f_score, tentative_g, neighbor))

return None

if __name__ == "__main__":
    start = (2, 8, 3, 1, 6, 4, 7, 0, 5)
    path = a_star_manhattan(start)

    if path:
        print("Solution using Manhattan Distance heuristic:")
        for state in path:
            print(state[:3], state[3:6], state[6:], sep="\n", end="\n\n")
        print(f"Total cost: {len(path) - 1}")
    else:
        print("No solution found.")

print("Name: Siddharth Arya")
print("USN: 1BM23CS328")

```

Output:

Solution using Manhattan Distance heuristic:

(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)

Total cost: 5

Name: Siddharth Arya

USN: 1BM23CS328

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

5(c)
Hill Climbing Algorithm

Algorithm

- ① Initialize board with a random configuration of N queens.
- ② Calculate the initial cost (number of attacking pairs).
- ③ While the cost is greater than 0:
 - ④ Generate all neighbors
 - ⑤ Evaluate the cost of each neighbor
 - ⑥ If lower cost neighbor found, move to that neighbor
- ⑦ Repeat until a solution is found (cost = 0)
- ⑧ Return the final configuration

① | 0 | 1 | 2 | 3 |
 Q | | | |
Q		
	Q	
		Q
 Initial state: { $x_0: 0, x_1: 1, x_2: 2, x_3: 3\}$
 Cost = 6

② | | | |
Q		
	Q	
		Q
 Initial state: { $x_0: 3, x_1: 2, x_2: 1, x_3: 0\}$
 Cost = 6

Code:

All steps for Hill Climbing Algorithm

```
import random
```

```
def calculate_cost(board):
```

```
    cost = 0
```

```
    n = len(board)
```

```
    for i in range(n):
```

```

for j in range(i + 1, n):
    if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
        cost += 1
return cost

def get_neighbors(board):
    neighbors = []
    n = len(board)
    for row in range(n):
        for new_col in range(n):
            if new_col != board[row]:
                new_board = board[:]
                new_board[row] = new_col
                neighbors.append(new_board)
    return neighbors

def hill_climbing(n, max_restarts=100):
    solutions = set()
    all_iterations = []

    for restart in range(max_restarts):
        current_board = [random.randint(0, n-1) for _ in range(n)]
        current_cost = calculate_cost(current_board)
        iterations = []

        iterations.append((list(current_board), current_cost))

        while current_cost > 0:
            neighbors = get_neighbors(current_board)
            next_board = None
            next_cost = current_cost

            for neighbor in neighbors:
                cost = calculate_cost(neighbor)
                if cost < next_cost:
                    next_board = neighbor
                    next_cost = cost

            if next_board is None:
                break
            current_board = next_board
            current_cost = next_cost

    all_iterations.append(iterations)
    return solutions

```

```

else:
    current_board = next_board
    current_cost = next_cost

iterations.append((list(current_board), current_cost))

if current_cost == 0:
    solutions.add(tuple(current_board))

all_iterations.append(iterations)

return list(solutions), all_iterations

def print_solution(board):
    print(" ".join(map(str, board)))

n = 4
solutions, all_iterations = hill_climbing(n)

print("Name: Siddharth Arya")
print("USN: 1BM23CS328")

for restart_idx, iterations in enumerate(all_iterations):
    print(f'Restart {restart_idx + 1}:')
    for step_idx, (board_state, cost) in enumerate(iterations):
        print(f'Iteration {step_idx + 1}:')
        print_solution(board_state)
        print(f'Cost: {cost}')
        print()

for i, solution in enumerate(solutions):
    print(f'Solution {i + 1}:')
    print_solution(solution)
    print()

```

Output:

```
Iteration 2:
```

```
3 2 0 1
```

```
Cost: 2
```

```
Restart 91:
```

```
Iteration 1:
```

```
1 2 0 3
```

```
Cost: 1
```

```
Restart 92:
```

```
Iteration 1:
```

```
0 2 2 3
```

```
Cost: 4
```

```
Iteration 2:
```

```
0 2 0 3
```

```
Cost: 2
```

```
Iteration 3:
```

```
1 2 0 3
```

```
Cost: 1
```

```
Restart 93:
```

```
Iteration 1:
```

```
3 2 3 0
```

```
Cost: 5
```

```
Iteration 2:
```

```
3 1 3 0
```

```
Cost: 2
```

```
Solution 1:
```

```
Final Configuration: (2, 0, 3, 1)
```

```
Solution 2:
```

```
Final Configuration: (1, 3, 0, 2)
```

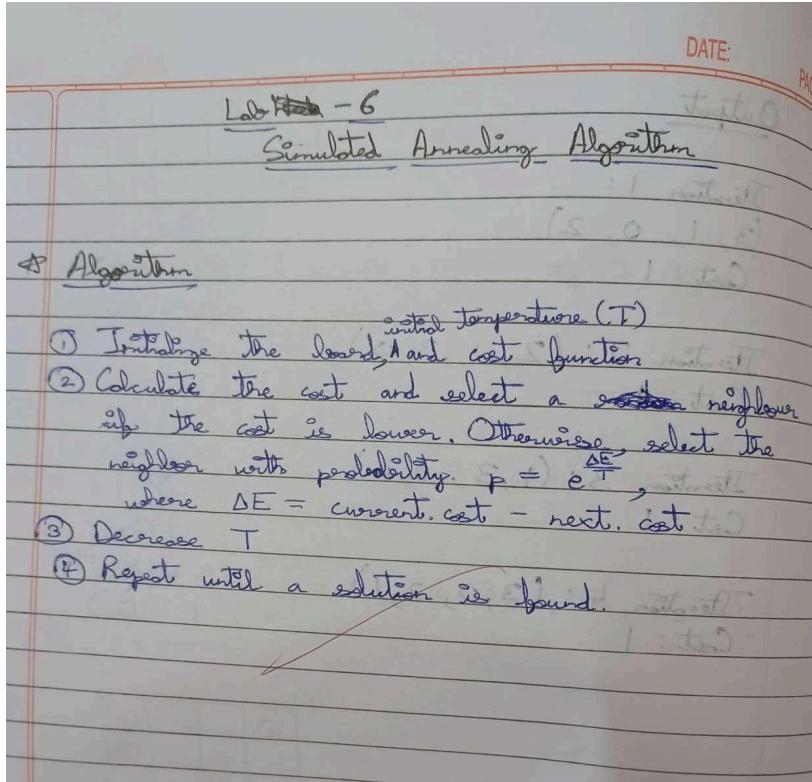
```
Name: Siddharth Arya
```

```
USN: 1BM23CS328
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



Code:

```
# Simualted Annealing  
# All steps
```

```
import numpy as np
from scipy.optimize import dual_annealing

def calculate_cost(board):
    cost = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                cost += 1
    return cost
```

```

def cost_function(board):
    board = np.round(board).astype(int)
    return calculate_cost(board)

def log_intermediate_results(x, f, context):
    iteration_data.append((np.round(x).astype(int), f))

def solve_8_queens(n=8, max_restarts=100):
    bounds = [(0, n-1)] * n
    unique_solutions = set()

    all_iterations = []
    for _ in range(max_restarts):
        global iteration_data
        iteration_data = []

        result = dual_annealing(cost_function, bounds, callback=log_intermediate_results)

        solution = np.round(result.x).astype(int)
        cost = result.fun

        all_iterations.append(iteration_data)

        if cost == 0:
            unique_solutions.add(tuple(solution))

    return list(unique_solutions), all_iterations

def print_solution(board):
    print(" ".join(map(str, board)))

solutions, all_iterations = solve_8_queens()

print("All Iterations for Each Restart:")
for restart_idx, iterations in enumerate(all_iterations):
    print(f'Restart {restart_idx + 1}:')
    for step_idx, (board_state, cost) in enumerate(iterations):
        print(f'Iteration {step_idx + 1}:')
        print("State (Queen positions):", board_state)
        print("Cost:", cost)

```

```
print()  
  
print("Unique Solutions Found:")  
for idx, solution in enumerate(solutions):  
    print(f"Solution {idx + 1}:")  
    print_solution(solution)  
    print()
```

```
print("Name: Siddharth Arya")  
print("USN: 1BM23CS328")
```

Output:

Unique Solutions Found: Solution 1: 5 2 0 6 4 7 1 3 Solution 2: 3 1 4 7 5 0 2 6 Solution 3: 5 2 6 1 7 4 0 3 Solution 4: 4 6 0 2 7 5 3 1 Solution 5: 5 3 1 7 4 6 0 2 Solution 6: 2 5 7 0 3 6 4 1 Solution 7: 5 2 6 3 0 7 1 4 Solution 8: 5 3 6 0 2 4 1 7 Solution 9: 2 5 1 6 4 0 7 3 Solution 10: 4 2 0 5 7 1 3 6 Solution 11: 3 7 0 2 5 1 6 4	Solution 50: 6 3 1 7 5 0 2 4 Solution 51: 4 6 0 3 1 7 5 2 Solution 52: 5 3 0 4 7 1 6 2 Solution 53: 3 1 7 5 0 2 4 6 Solution 54: 6 1 3 0 7 4 2 5 Solution 55: 3 0 4 7 1 6 2 5 Solution 56: 2 6 1 7 5 3 0 4 Solution 57: 4 6 1 3 7 0 2 5 Name: Siddharth Arya USN: 1BM23CS328
---	---

Program 6

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

Algorithm:

DATE: 22/1/25 PAGE: 1

Lab - 7
Knowledge Base using Propositional Logic

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

* Example:

$$\alpha = A \vee B$$

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

A	B	C	$A \vee C$	$B \vee \neg C$	KB	α
false	false	false	false	true	false	false
false	false	true	true	false	false	false
false	true	false	true	false	false	false
false	true	true	true	true	false	true
true	false	false	true	true	true	true
true	false	true	true	true	true	true
true	true	false	true	false	false	true
true	true	true	true	true	true	true

Code:

Knowledge Base using Propositional Logic

Symbols: AND, OR, Negation/NOT, Implies/Implication, If and only If

```
from itertools import product
```

```

def AND(p, q):
    return p and q

def OR(p, q):
    return p or q

def NOT(p):
    return not p

def IMPLIES(p, q):
    return (not p) or q

def IFF(p, q):
    return p == q

class Formula:
    def __init__(self, op, *args):
        self.op = op
        self.args = args

    def evaluate(self, assignment):
        if isinstance(self.op, str) and len(self.args) == 0:
            return assignment[self.op]

        if self.op == 'AND':
            return AND(self.args[0].evaluate(assignment), self.args[1].evaluate(assignment))
        elif self.op == 'OR':
            return OR(self.args[0].evaluate(assignment), self.args[1].evaluate(assignment))
        elif self.op == 'NOT':
            return NOT(self.args[0].evaluate(assignment))
        elif self.op == 'IMPLIES':
            return IMPLIES(self.args[0].evaluate(assignment), self.args[1].evaluate(assignment))
        elif self.op == 'IFF':
            return IFF(self.args[0].evaluate(assignment), self.args[1].evaluate(assignment))
        else:
            raise ValueError(f"Unknown operator: {self.op}")

    def __str__(self):
        if isinstance(self.op, str) and len(self.args) == 0:
            return self.op


```

```

        elif self.op == 'NOT':
            return f"¬{self.args[0]}"
        elif self.op == 'AND':
            return f"({self.args[0]} ∧ {self.args[1]})"
        elif self.op == 'OR':
            return f"({self.args[0]} ∨ {self.args[1]})"
        elif self.op == 'IMPLIES':
            return f"({self.args[0]} → {self.args[1]})"
        elif self.op == 'IFF':
            return f"({self.args[0]} ↔ {self.args[1]})"
        else:
            return f"UnknownOp({self.op})"

def extract_symbols(formula):
    symbols = set()
    if isinstance(formula.op, str) and len(formula.args) == 0:
        symbols.add(formula.op)
    else:
        for arg in formula.args:
            symbols |= extract_symbols(arg)
    return symbols

def extract_main_subformulas(formula):
    subs = set()
    subs.add(formula)
    for arg in formula.args:
        subs.add(arg)
    return subs

def entails(KB, alpha):
    base_symbols = {'A', 'B', 'C'}
    symbols_in_formulas = extract_symbols(KB) | extract_symbols(alpha)
    all_symbols = sorted(list(base_symbols | symbols_in_formulas))

    subformulas_KB = extract_subformulas(KB)
    subformulas_alpha = extract_subformulas(alpha)
    all_subformulas = list(subformulas_KB | subformulas_alpha)

    all_subformulas = [f for f in all_subformulas if not (isinstance(f.op, str) and len(f.args) == 0)]

```

```

all_subformulas.sort(key=lambda f: str(f))

print("Knowledge Base (KB):", KB)
print("Alpha (Query):", alpha)
print()

headers = all_symbols + [str(f) for f in all_subformulas] + ["KB", "Alpha"]

truth_table = []
true_assignments = []

for values in product([False, True], repeat=len(all_symbols)):
    assignment = dict(zip(all_symbols, values))

    subformula_values = []
    for f in all_subformulas:
        val = f.evaluate(assignment)
        subformula_values.append(val)

    kb_val = KB.evaluate(assignment)
    alpha_val = alpha.evaluate(assignment)

    truth_table.append((assignment, subformula_values, kb_val, alpha_val))
    if kb_val and alpha_val:
        true_assignments.append(assignment)

header_str = " | ".join(f"{{h:^12}}" for h in headers)
print(header_str)
print("-" * len(header_str))

for assignment, sub_vals, kb_val, alpha_val in truth_table:
    vals = ['T' if assignment[s] else 'F' for s in all_symbols]
    vals += ['T' if v else 'F' for v in sub_vals]
    vals += ['T' if kb_val else 'F', 'T' if alpha_val else 'F']

    row_str = " | ".join(f"{{v:^12}}" for v in vals)
    print(row_str)

print("\nAssignments where both KB and Alpha are TRUE:")
if true_assignments:

```

```

for a in true_assignments:
    print({k: ('T' if v else 'F') for k, v in a.items()})
else:
    print("None")

for assignment, sub_vals, kb_val, alpha_val in truth_table:
    if kb_val and not alpha_val:
        print("\nResult: KB does NOT entail Alpha.")
        return False

    print("\nResult: KB entails Alpha.")
    return True

```

```

A = Formula('A')
B = Formula('B')
C = Formula('C')

```

```

KB = Formula('IFF',
             Formula('IMPLIES',
                     Formula('AND', A, Formula('NOT', B)),
                     Formula('OR', C, A)),
             Formula('OR', B, Formula('NOT', C)))

```

```
alpha = Formula('IMPLIES', A, C)
```

```

result = entails(KB, alpha)
print(f"\nDoes KB entail alpha? {result}")

```

```

print("Name: Siddharth Arya")
print("USN: 1BM23CS328")

```

Output:

Knowledge Base (KB): $((A \wedge \neg B) \rightarrow (C \vee A)) \vee (B \vee \neg C)$													Alpha (Query): $(A \rightarrow C)$	
A	B	C	$((A \wedge \neg B) \rightarrow (C \vee A)) \vee (B \vee \neg C)$	$((A \wedge \neg B) \rightarrow (C \vee A))$	$(B \vee \neg C)$	$(A \rightarrow C)$	$(A \wedge \neg B)$	$(B \vee \neg C)$	$(C \vee A)$	$\neg B$	$\neg C$	KB	Alpha	
F	F	F	T	T	T	F	T	F	T	T	T	T		
F	F	T	F	T	T	F	F	T	T	F	T	T		
F	T	F	T	T	T	F	F	F	T	T	T	T		
F	T	T	T	T	T	F	T	T	F	F	T	T		
T	F	F	T	T	F	T	T	T	T	T	F	F		
T	F	T	F	T	T	T	F	T	F	F	T	T		
T	T	F	T	T	F	T	T	F	T	T	F	F		
T	T	T	T	T	T	F	T	T	F	F	T	T		

Assessments where both KB and Alpha are TRUE:

```
{"A": "F", "B": "F", "C": "F"}, {"A": "F", "B": "T", "C": "F"}, {"A": "T", "B": "F", "C": "F"}, {"A": "F", "B": "T", "C": "T"}, {"A": "T", "B": "T", "C": "T"}
```

Result: KB does NOT entail Alpha.

Does KB entail alpha? False

Name: Siddharth Arya
USN: 1BM23CS328

Program 7

Implement unification in first order logic

Algorithm:

DATE: _____ PAGE: _____

Lab - 9
Implement unification in First Order Logic.

* Algorithm

UNIFY(x, y, θ):

- ① If $\theta = \text{failure}$, return failure
- ② Else if $x = y$, return θ
- ③ Else if x is a variable, return UNIFY VAR(x, y, θ)
- ④ Else if y is a variable, return UNIFY VAR(y, x, θ)
- ⑤ Else if x & y are both compound expressions:
 - If predicate/function symbols differ → return failure
 - If their arity (no. of arguments) differ → return failure
 - else recursively UNIFY their argument lists
- ⑥ Else return failure

UNIFY VAR(var, x, θ):

- ① If var is already in θ :
 - return UNIFY($\theta[\text{var}], x, \theta$)
- ② Else if x is already in θ :
 - return UNIFY($var, \theta[x], \theta$)
- ③ Else if var occurs in x (occurred-check):
 - return failure
- ④ Else extend θ with $\{var \rightarrow x\}$
 - return θ

Code:

Implement unification in first order logic

```
def is_variable(x):
    return isinstance(x, str) and x[0].islower() and x.isalpha()
```

```
def is_compound(x):
    return isinstance(x, tuple) and len(x) > 0
```

```
def get_functor(x):
    return x[0] if is_compound(x) else None
```

```

def get_args(x):
    return list(x[1:]) if is_compound(x) else []

def occur_check(var, x, theta):
    if var == x:
        return True
    elif is_variable(x) and x in theta:
        return occur_check(var, theta[x], theta)
    elif is_compound(x):
        return any(occur_check(var, arg, theta) for arg in get_args(x))
    else:
        return False

def unify(x, y, theta=None):
    if theta is None:
        theta = {}

    if theta == "failure":
        return "failure"
    elif x == y:
        return theta
    elif is_variable(x):
        return unify_var(x, y, theta)
    elif is_variable(y):
        return unify_var(y, x, theta)
    elif is_compound(x) and is_compound(y):
        if get_functor(x) != get_functor(y):
            return "failure"
        if len(get_args(x)) != len(get_args(y)):
            return "failure"
        return unify(get_args(x), get_args(y), theta)
    elif isinstance(x, list) and isinstance(y, list):
        if len(x) != len(y):
            return "failure"
        if not x:
            return theta
        first_unify = unify(x[0], y[0], theta)
        return unify(x[1:], y[1:], first_unify)
    else:

```

```

        return "failure"

def unify_var(var, x, theta):
    if var in theta:
        return unify(theta[var], x, theta)
    elif is_variable(x) and x in theta:
        return unify(var, theta[x], theta)
    elif occur_check(var, x, theta):
        return "failure"
    else:
        theta[var] = x
    return theta

```

```

expr1 = ("P", "x", ("h", "y"))
expr2 = ("P", "a", ("h", "b"))
print(unify(expr1, expr2))

```

```

expr3 = ("P", "x", ("h", "y"))
expr4 = ("P", "a", ("f", "z"))
print(unify(expr3, expr4))

```

```

expr5 = ("Knows", "John", "x")
expr6 = ("Knows", "x", "Elisabeth")
print(unify(expr5, expr6))

```

```

print()
print("Name: Siddharth Arya")
print("USN: 1BM23CS328")

```

Output:

```
{'x': 'a', 'y': 'b'}  
failure  
failure
```

Name: Siddharth Arya
USN: 1BM23CS328

Program 8

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

Algorithm:

DATE: _____ PAGE: _____

★ Forward Chaining FOL Example:

The law says that it is a crime for an American to sell weapons to hostile nations. The country Narnia, an enemy of America, has some missiles, and all its missiles were sold to it by Colonel West, who is an American citizen. An enemy of America counts as hostile.

Prove that West is a criminal.

Rule 1: $\forall x, y, z \quad \text{American}(x) \wedge \text{Weapon}(y) \wedge \text{Sells}(x, y, z) \Rightarrow \text{Hostile}(z)$

Rule 2: $\forall x \quad \text{Missile}(x) \wedge \text{Owes}(\text{Narnia}, x) \Rightarrow \text{Sells}(\text{West}, x, \text{Narnia})$

Rule 3: $\forall x \quad \text{Enemy}(x, \text{America}) \Rightarrow \text{Hostile}(x)$

Rule 4: $\forall x \quad \text{Missile}(x) \Rightarrow \text{Weapon}(x)$

American (West)
 Enemy (Narnia, America)
 Owes (Narnia, M1) and
 Missile (M1)

Goal:

```

    graph TD
      C[Criminal(West)] --> W[Weapon(M1)]
      C --> S[Sells(West, M1, Narnia)]
      C --> H[Hostile(Narnia)]
      W --> A[American(West)]
      M[Missile(M1)] --> W
      O[Owes(Narnia, M1)] --> S
      E[Enemy(Narnia, America)] --> H
  
```

Code:

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

```
# Forward reasoning in First Order Logic
```

```
from itertools import product
```

```
def is_variable(x):  
    return isinstance(x, str) and x[0].islower()
```

```
def substitute(theta, x):  
    if isinstance(x, tuple):  
        return (x[0], [substitute(theta, arg) for arg in x[1]])  
    elif isinstance(x, list):  
        return [substitute(theta, xi) for xi in x]  
    else:  
        return theta.get(x, x)
```

```
def unify(x, y, theta=None):  
    if theta is None:  
        theta = {}  
    if theta == "fail":  
        return "fail"  
    elif x == y:  
        return theta  
    elif is_variable(x):  
        return unify_var(x, y, theta)  
    elif is_variable(y):  
        return unify_var(y, x, theta)  
    elif isinstance(x, tuple) and isinstance(y, tuple):  
        if x[0] != y[0] or len(x[1]) != len(y[1]):  
            return "fail"  
        return unify(x[1], y[1], theta)  
    elif isinstance(x, list) and isinstance(y, list):  
        if not x and not y:  
            return theta  
        return unify(x[1:], y[1:], unify(x[0], y[0], theta))
```

```

else:
    return "fail"

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)
    elif occurs_check(var, x, theta):
        return "fail"
    else:
        new_theta = theta.copy()
        new_theta[var] = x
        return new_theta

def occurs_check(var, x, theta):
    if var == x:
        return True
    elif isinstance(x, list):
        return any(occurs_check(var, xi, theta) for xi in x)
    elif isinstance(x, tuple):
        return occurs_check(var, x[1], theta)
    elif x in theta:
        return occurs_check(var, theta[x], theta)
    return False

def pred(name, *args):
    return (name, list(args))

def fol_fc_ask(KB, query, trace=True):
    inferred = set(map(str, KB["facts"]))
    new_inferred = True
    iteration = 0

    while new_inferred:
        iteration += 1
        new_inferred = False
        if trace:

```

```

print(f"\n--- Iteration {iteration} ---")

for rule in KB["rules"]:
    premises, conclusion = rule
    possible_bindings = [list(inferred) for _ in premises]

    for facts_combo in product(*possible_bindings):
        theta = {}
        ok = True
        for premise, fact in zip(premises, facts_combo):
            fact = eval(fact)
            theta = unify(premise, fact, theta)
            if theta == "fail":
                ok = False
                break

        if not ok:
            continue

        inferred_fact = substitute(theta, conclusion)
        inferred_str = str(inferred_fact)

        if inferred_str not in inferred:
            inferred.add(inferred_str)
            new_inferred = True
            if trace:
                print(f"Derived new fact: {inferred_fact} from {premises}")

        if unify(inferred_fact, query) != "fail":
            if trace:
                print(f"\nDerived goal: {query}")
            return True

return False

if __name__ == "__main__":

```

```

KB = {
    "facts": [
        pred("American", "West"),
        pred("Missile", "M1"),
        pred("Owns", "Nono", "M1"),
        pred("Enemy", "Nono", "America")
    ],
    "rules": [
        ([pred("American", "x"), pred("Weapon", "y"), pred("Sells", "x", "y", "z"),
        pred("Hostile", "z")],
        pred("Criminal", "x")),
        ([pred("Missile", "x")], pred("Weapon", "x")),
        ([pred("Missile", "x"), pred("Owns", "Nono", "x")], pred("Sells", "West", "x", "Nono")),
        ([pred("Enemy", "x", "America")], pred("Hostile", "x"))
    ]
}

```

```
query = pred("Criminal", "West")
```

```

print("Inference process:")
result = fol_fc_ask(KB, query)

```

```

if result:
    print("\nConclusion: West is a Criminal.")
else:
    print("\nCould not prove the query.")

```

```

print()
print("Name: Siddharth Arya")
print("USN: 1BM23CS328")

```

Output:

```
Inference process:  
---- Iteration 1 ----  
Derived new fact: ('Weapon', ['M1']) from [('Missile', ['x'])]  
Derived new fact: ('Sells', ['West', 'M1', 'Nono']) from [('Missile', ['x']), ('Owns', ['Nono', 'x'])]  
Derived new fact: ('Hostile', ['Nono']) from [('Enemy', ['x', 'America'])]  
---- Iteration 2 ----  
Derived new fact: ('Criminal', ['West']) from [('American', ['x']), ('Weapon', ['y']), ('Sells', ['x', 'y', 'z']), ('Hostile', ['z'])]  
Derived goal: ('Criminal', ['West'])  
Conclusion: West is a Criminal.  
Name: Siddharth Arya  
USN: 1BM23CS328
```

Program 9

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

Algorithm:

DATE: _____
PAGE: _____

Lab - 8
Resolution in FOL

* Algorithm

- ① Convert all sentences to CNF
- ② Negate conclusion & convert result to CNF
- ③ Add negated conclusion S to premise clauses
- ④ Repeat until contradiction or no progress is made:
 - a) Select 2 clauses
 - b) Resolve them together
 - c) If resultant is empty clause, a contradiction has been found.
 - d) If not, add resultant to premises

* Output

Given KB

John likes all kind of food
Apple & vegetables are food
Anyone who is alive implies not killed
Anyone who is not killed implies alive
John likes peanuts

Code:

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

```
from itertools import combinations
```

```

def negate(literal):
    if literal.startswith("¬"):
        return literal[1:]
    else:
        return "¬" + literal

def is_variable(x):
    return isinstance(x, str) and len(x) == 1 and x.islower()

def get_predicate(literal):
    if literal.startswith("¬"):
        return literal[1:literal.find("(")]
    else:
        return literal[:literal.find("(")]

def get_args_from_literal(literal):
    return [arg.strip() for arg in literal[literal.find("(")+1:-1].split(",")]

def literal_to_tuple(literal):
    pred = get_predicate(literal)
    args = get_args_from_literal(literal)
    return (pred, *args)

def tuple_to_literal(tuple_form, negated=False):
    pred = tuple_form[0]
    args = ", ".join(tuple_form[1:])
    literal = f"{pred}({args})"
    if negated:
        return "¬" + literal
    else:
        return literal

def substitute_literal(literal, subs):
    lit_tuple = literal_to_tuple(literal)
    new_args = []
    for arg in lit_tuple[1:]:
        if is_variable(arg) and arg in subs:
            new_args.append(subs[arg])
        else:
            new_args.append(arg)

```

```

new_literal_tuple = (lit_tuple[0], *new_args)
return tuple_to_literal(new_literal_tuple, literal.startswith("~"))

def resolve(ci, cj):
    resolvents = set()
    for lit_i in ci:
        for lit_j in cj:
            if negate(lit_i) == lit_j or lit_i == negate(lit_j):
                new_clause = (ci - {lit_i}) | (cj - {lit_j})
                resolvents.add(frozenset(new_clause))

    for lit_i in ci:
        for lit_j in cj:
            neg_lit_i = negate(lit_i)
            if get_predicate(neg_lit_i) == get_predicate(lit_j) and
len(get_args_from_literal(neg_lit_i)) == len(get_args_from_literal(lit_j)):
                term_i = literal_to_tuple(neg_lit_i)
                term_j = literal_to_tuple(lit_j)

                unifier = unify(term_i, term_j)

                if unifier != "failure":
                    new_clause_i = {substitute_literal(l, unifier) for l in ci - {lit_i}}
                    new_clause_j = {substitute_literal(l, unifier) for l in cj - {lit_j}}
                    new_clause = frozenset(new_clause_i | new_clause_j)
                    resolvents.add(new_clause)

    return resolvents

def resolution(kb, query):
    kb_clauses = set(kb)
    kb_clauses.add(frozenset([negate(query)]))

    new_clauses = set()
    iterations = 0

    while True:
        iterations += 1
        print(f"\nIteration {iterations}: KB size = {len(kb_clauses)}")

```

```

pairs = list(combinations(kb_clauses, 2))
for (ci, cj) in pairs:
    resolvents = resolve(ci, cj)
    for res in resolvents:
        if frozenset() in resolvents:
            print("\nDerived empty clause -> Contradiction found")
            return True
        if res not in kb_clauses and res not in new_clauses:
            new_clauses.add(res)

if new_clauses.issubset(kb_clauses):
    return False
kb_clauses = kb_clauses.union(new_clauses)
new_clauses = set()

def is_variable(x):
    return isinstance(x, str) and len(x) == 1 and x.islower()

def is_compound(x):
    return isinstance(x, tuple) and len(x) > 0

def get_functor(x):
    return x[0] if is_compound(x) else None

def get_args(x):
    return list(x[1:]) if is_compound(x) else []

def occur_check(var, x, theta):
    if var == x:
        return True
    elif is_variable(x) and x in theta:
        return occur_check(var, theta[x], theta)
    elif is_compound(x):
        return any(occur_check(var, arg, theta) for arg in get_args(x))
    else:
        return False

def unify(x, y, theta=None):
    if theta is None:

```

```

theta = {}

if theta == "failure":
    return "failure"
elif x == y:
    return theta
elif is_variable(x):
    return unify_var(x, y, theta)
elif is_variable(y):
    return unify_var(y, x, theta)
elif is_compound(x) and is_compound(y):
    if get_functor(x) != get_functor(y):
        return "failure"
    if len(get_args(x)) != len(get_args(y)):
        return "failure"
    return unify(get_args(x), get_args(y), theta)
elif isinstance(x, list) and isinstance(y, list):
    if len(x) != len(y):
        return "failure"
    if not x:
        return theta
    first_unify = unify(x[0], y[0], theta)
    if first_unify == "failure":
        return "failure"
    return unify(x[1:], y[1:], first_unify)
else:
    return "failure"

def unify_var(var, x, theta):
    if var in theta:
        return unify(theta[var], x, theta)
    elif is_variable(x) and x in theta:
        return unify(var, theta[x], theta)
    elif occur_check(var, x, theta):
        return "failure"
    else:
        theta[var] = x
        return theta

```

```

kb = set()

kb.add(frozenset(["~Food(x)", "Likes(John, x)"]))
kb.add(frozenset(["Food(Peanuts)"]))
kb.add(frozenset(["Food(Apple)"]))
kb.add(frozenset(["Food(Vegetables)"]))
kb.add(frozenset(["~Eats(x, y)", "Killed(x)", "Food(y)"]))
kb.add(frozenset(["Eats(Anil, Peanuts)"]))
kb.add(frozenset(["Alive(Anil)"]))
kb.add(frozenset(["~Eats(Anil, y)", "Eats(Harry, y)"]))
kb.add(frozenset(["~Alive(x)", "~Killed(x)"]))
kb.add(frozenset(["Killed(x)", "Alive(x)"]))

```

```

print("Knowledge Base (Clauses):")
for c in kb:
    print(c)

query = "Likes(John, Peanuts)"

proved = resolution(kb, query)

print("\nResult:")
if proved:
    print(f"Query proved: {query}")
else:
    print(f"Could not prove the query: {query}")

print()
print("Name: Siddharth Arya")
print("USN: 1BM23CS328")

```

Output:

```
Knowledge Base (Clauses):
frozenset({'Food(Apple)'})
frozenset({'~Killed(x)', '~Alive(x)'})
frozenset({'Alive(x)', 'Killed(x)'})
frozenset({'Eats(Anil, Peanuts)'})
frozenset({'~Food(x)', 'Likes(John, x)'})
frozenset({'Food(Vegetables)'})
frozenset({'Food(y)', '~Eats(x, y)', 'Killed(x)'})
frozenset({'Alive(Anil)'})
frozenset({'Food(Peanuts)'})
frozenset({'~Eats(Anil, y)', 'Eats(Harry, y)'})
```

Iteration 1: KB size = 11

Iteration 2: KB size = 29

Derived empty clause → Contradiction found

Result:

Query proved: Likes(John, Peanuts)

Name: Siddharth Arya

USN: 1BM23CS328

Program 10

Implement Alpha-Beta Pruning

Algorithm:

Lab - 12
Alpha - Beta Pruning

* Algorithm:

function ALPHA-BETA-SEARCH (state) returns an action
 $v \leftarrow \text{MAX_VALUE} (\text{state}, -\infty, +\infty)$
 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 a in ACTIONS (state) do
 $v \leftarrow \text{MAX}(v, \text{MIN_VALUE} (\text{RESULT}(s, a), \alpha, \beta))$
 if $v \geq \beta$ then return v
 $\alpha \leftarrow \text{MAX}(\alpha, v)$
 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 a in ACTIONS (state) do
 $v \leftarrow \text{MIN}(v, \text{MAX_VALUE} (\text{RESULT}(s, a), \alpha, \beta))$
 if $v \leq \alpha$ then return v
 $\beta \leftarrow \text{MIN}(\beta, v)$
 return v

Code:

Implement Alpha-Beta Pruning.

```

import math

class Node:
    def __init__(self, value=None, children=None, is_max=True):
        self.value = value
        self.children = children or []
        self.is_max = is_max

    def alpha_beta(self, alpha=-math.inf, beta=math.inf):
        if not self.children:
            return self.value

        if self.is_max:
            value = -math.inf
            for child in self.children:
                value = max(value, child.alpha_beta(alpha, beta))
            alpha = max(alpha, value)
            if alpha >= beta:
                break
            self.value = value
            return value
        else:
            value = math.inf
            for child in self.children:
                value = min(value, child.alpha_beta(alpha, beta))
            beta = min(beta, value)
            if beta <= alpha:
                break
            self.value = value
            return value

    def print_tree(self, level=0):
        indent = " " * level
        if not self.children:
            print(f'{indent}Leaf Node (value={self.value})')
        else:
            role = "MAX" if self.is_max else "MIN"
            print(f'{indent} {role} Node (value={self.value})')

```

```

for child in node.children:
    print_tree(child, level + 1)

def build_tree_from_leaves(leaf_values, branching_factor, is_max=True):
    current_level_nodes = [Node(value=v, is_max=not is_max) for v in leaf_values]
    level = 1

    while len(current_level_nodes) > 1:
        next_level_nodes = []
        for i in range(0, len(current_level_nodes), branching_factor):
            children = current_level_nodes[i:i + branching_factor]
            node = Node(children=children, is_max=(level % 2 == 0))
            next_level_nodes.append(node)
        current_level_nodes = next_level_nodes
        level += 1

    root = current_level_nodes[0]
    root.is_max = True
    return root

if __name__ == "__main__":
    branching_factor = int(input("Enter number of branches per node: "))
    num_leaves = int(input("Enter total number of leaf nodes: "))

    depth = math.log(num_leaves, branching_factor) + 1
    if abs(round(depth) - depth) > 1e-9:
        print("\n Error: The number of leaves must form a full tree ( $b^{(d-1)}$ ).")
        print(" Example: With 3 branches, leaf count = 3, 9, 27, ...")
        exit()

    depth = int(round(depth))
    print(f"\nTree depth will be {depth} levels (including root).")

    leaf_values = []
    print("\nEnter the values for each leaf node (can be number, inf, -inf):")
    for i in range(num_leaves):
        val = input(f"Leaf {i+1}: ").strip()
        if val.lower() == "inf":
            val = math.inf
        elif val.lower() == "-inf":

```

```
val = -math.inf
else:
    val = float(val)
leaf_values.append(val)

root = build_tree_from_leaves(leaf_values, branching_factor, is_max=True)
best_value = alpha_beta(root)

print("\n==== Alpha-Beta Pruning Result ====")
print(f"Best value for root: {best_value}\n")

print("==== Game Tree ====")
print_tree(root)
```

```
print()
print("Name: Siddharth Arya")
print("USN: 1BM23CS328")
```

Output:

```
Enter number of branches per node: 2
Enter total number of leaf nodes: 8

Tree depth will be 4 levels (including root).

Enter the values for each leaf node (can be number, inf, -inf):
Leaf 1: 3
Leaf 2: 5
Leaf 3: 2
Leaf 4: 9
Leaf 5: 1
Leaf 6: 7
Leaf 7: 4
Leaf 8: 8

==== Alpha-Beta Pruning Result ====
Best value for root: 4.0

==== Game Tree ====
MAX Node (value=4.0)
    MAX Node (value=3.0)
        MIN Node (value=3.0)
            Leaf Node (value=3.0)
            Leaf Node (value=5.0)
        MIN Node (value=2.0)
            Leaf Node (value=2.0)
            Leaf Node (value=9.0)
    MAX Node (value=4.0)
        MIN Node (value=1.0)
            Leaf Node (value=1.0)
            Leaf Node (value=7.0)
        MIN Node (value=4.0)
            Leaf Node (value=4.0)
            Leaf Node (value=8.0)
```

Name: Siddharth Arya
USN: 1BM23CS328