

# **VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

“Jnana Sangama”, Belgaum -590014, Karnataka.



## **LAB REPORT on**

## **Artificial Intelligence (23CS5PCAIN)**

*Submitted by*

**ARNAV DINESH (1BM23CS052)**

*in partial fulfilment 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 **ARNAV DINESH (1BM23CS052)**, who is a Bonafide student of **B.M.S. College of Engineering**. It is in partial fulfilment 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.

<b>Sandhya A Kulkarni</b> Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE
--	--

## Index

Sl. No.	Experiment Title	Page No.
1	Implement Tic –Tac –Toe Game Implement vacuum cleaner agent	4-14
2	Implement 8 puzzle problems using Iterative deepening search algorithm	15-18
3	Implement A* search algorithm	19-27
4	Implement Hill Climbing search algorithm to solve N-Queens problem	28-29
5	Simulated Annealing to Solve 8-Queens problem	30-31
6	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	32-34
7	Implement unification in first order logic	35-37
8	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	38-39
9	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	40-45
10	Implement Alpha-Beta Pruning.	46-47

GitHub Link:  
<https://github.com/ArnavRD/AI-LAB>

## Certification Of The Assignment given:



# CERTIFICATE OF ACHIEVEMENT



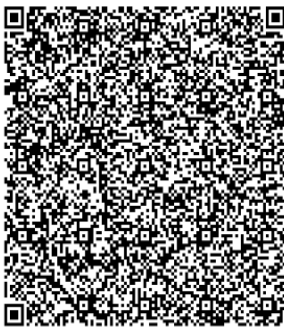
The certificate is awarded to

**Arnav Dinesh**

for successfully completing

**Artificial Intelligence Foundation Certification**

on November 15, 2025



*Congratulations! You make us proud!*

Issued on: Saturday, November 15, 2025  
To verify, scan the QR code at <https://verify.onwingspan.com>

*Satheesha B.N.*  
Satheesha B. Nanjappa  
Senior Vice President and Head  
Education, Training and Assessment  
Infosys Limited

## Program 1

Implement Tic - Tac - Toe Game

### Algorithm:

LAB-1  
TIC TAC TOE PROGRAM  
PSEUDOCODE / Algorithm

```
TicTacToe(board) {  
    # Print the board  
    for i in range(3):  
        for j in range(3):  
            print(board[i][j])  
  
    # Initialize the board  
    for i from 0 to 3  
        for j from 0 to 3  
            board[i][j] = '-'  
  
    while(1):  
        # Take input from user  
        # Player 1 - X  
        # Player 2 - O  
        print("Player 1 enter position:")  
        # Input 1  
        if board[input] != '-':  
            # Ask for input again  
            board[input] = 'X'  
            print("Player 2 enter the position:")  
            # Input 2  
            if board[input] != '-':  
                # Ask for input again  
                board[input] = 'O'
```

Now checking for the winning conditions of both Player 1 and Player 2.

The possibilities

$w = [1, 2, 3], [4, 5, 6], [7, 8, 9], [1, 4, 7],$   
 $[2, 5, 8], [3, 6, 9]$   
 $[1, 5, 9], [3, 5, 7]$

if (player = "Player 1")  
 print("Player 1 wins")

if (player = "Player 2")  
 print("Player 2 wins")

Now checking Tie breaker situations  
First after taking input  
checking if the Player 1 / Player 2 wins or  
not using the condition mentioned above.

If the full board is occupied  
with X and O having the configurations  
minced up  
Then there is a tie breaker!!!

O/P:-

X		

X-1

X		
O		

O-7

X		
	X	
O		

X-5

X	X	
O	O	

O-8

X	X	
O	O	X

X-9

Player X wins

**Code:**

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

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

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

    return False

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

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

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

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

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

```

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

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

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

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

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

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

tic_tac_toe()

```

### Output case1:

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

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

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

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

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

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

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

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

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

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

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

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

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

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

Player X wins!
Total moves (cost): 9

```

---

## Output case2:

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

Current Board:

```
[' ', ' ', ' ', ' ', ' ']  
[' ', ' ', ' ', ' ', ' ']  
[' ', ' ', ' ', ' ', ' ']
```

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

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

Current Board:

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

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

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

Current Board:

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

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

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

Current Board:

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

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

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

Current Board:

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

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

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

Current Board:

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

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

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

Current Board:

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

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

Player O wins!

Total moves (cost): 6

---

### Output case3:

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

Current Board:

```
[' ', ' ', ' ']  
[' ', ' ', ' ']  
[' ', ' ', ' ']
```

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

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

Current Board:

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

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

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

Current Board:

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

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

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

Current Board:

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

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

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

Current Board:

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

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

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

Current Board:

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

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

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

Current Board:

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

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

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

Current Board:

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

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

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

Current Board:

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

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

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

Current Board:

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

Game is a draw.

Total moves (cost): 9

## Implement vacuum cleaner agent

### Algorithm:

Vacuum Cleaner

Let's say room map is

A	B
C	D

→ Step 1, Initialize all rooms  $A=0, B=0, C=0, D=0$   
 Then if 0 represents dirty, 1 represents clean.

Check the number of rooms, say  $n$ .

Using a for loop,  
 $\text{vacuum}() \{$   
 $\text{for } (i=0; i < n; i++) \{$

$0/1 = \text{Room}[0,0]$  already  
 $\text{Room}[0,1]$  already  
 $\text{Room}[1,1]$  dirty, cleaning  
 $\text{Room}[1,0]$  already cleaned  
 $\text{Room}[0,0]$  dirty

$\{ \text{if } (\text{room}[A]=0) \{$   
 $\text{room}[A]=1 \} \text{ print}(\text{room A cleaned})$   
 $\text{else if } (\text{room}[B]=0) \{$   
 $\text{room}[B]=1 \} \text{ print}(\text{room B cleaned})$   
 $\text{else if } (\text{room}[C]=0) \{$   
 $\text{room}[C]=1 \} \text{ print}(\text{room C cleaned})$   
 $\text{else if } (\text{room}[D]=0) \{$   
 $\text{room}[D]=1 \} \text{ print}(\text{room D cleaned})$   
 $\}$

Similarly traverse all the rooms mentioned  
 by the user until all the rooms are cleaned.  
 Then return all the rooms which are  
 cleaned.

2019

### Code:

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

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

    if location == 'A':
        if state['A'] == 1: # If A is dirty
```

```

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

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

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

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

print("Cost:", cost)
print(state)

vacuum_cleaner()

```

## OUTPUT Case1:

```

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

```

### OUTPUT Case2:

```

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

```

### OUTPUT Case3:

```

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

```

## Program2

Implement 8 puzzle problem using Iterative deepening search algorithm

### Algorithm:

Week 2

8 Puzzle using misplaced tiles & Manhattan distance & IDDFS.

You have a  $3 \times 3$  board with 8 numbered tiles & 1 blank space. Goal is to rearrange the tiles to reach the goal state.  
moves allowed - Up, Down, left, right

Misplaced tiles Counts how many tiles are not in their goal position.

Pseudocode

```
def misplaced_tiles(state, goal_state):  
    count = 0  
    for i from 0 to 2:  
        for j from 0 to 2:  
            if state[i][j] != 0 and state[i][j] != goal_state[i][j]:  
                count += 1  
    return count
```

return count

```
def find_position(state, tile):  
    for i from 0 to 2:  
        for j from 0 to 2:  
            if state[i][j] == tile:  
                return (i, j)
```

Manhattan distance

Sum of the distances of each tile from its goal position, using grid distance.

Pseudocode

```
def manhattan_distance(state, goal_state):  
    distance = 0  
    for i from 0 to 2:  
        for j from 0 to 2:  
            tile = state[i][j]  
            if tile != 0:  
                (goal_i, goal_j) = find_position(goal_state, tile)  
                distance = distance + abs(i - goal_i) + abs(j - goal_j)
```

Manhattan distance

IDDFS Iterative deepening DFS

It combines both DFS & BFS

It repeatedly runs DFS with increasing depth limits until solution is found.

### Pseudocode

```
def IDDFS(initial-state, goal-state):
    depth = 0
    while True:
        result = DFS(initial-state, goal-state, depth):
        if result == SUCCESS:
            return result
        depth += 1
```

```
def DFS(state, goal-state, limit):
    if state == goal-state:
        return SUCCESS
```

```
    else if limit == 0:
        return CUTOFF
```

```
    else:
```

```
        for each move in possible-moves(state):
```

```
            child = apply-move(state, move)
```

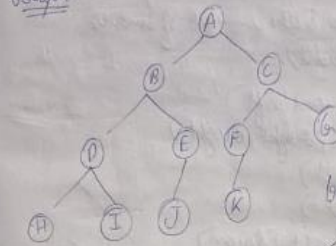
```
            result = DFS(child, goal-state, limit-1)
```

```
            if result == SUCCESS:
```

```
                return SUCCESS
```

```
            return CUTOFF
```

### Graph



### Applying IDDFS

A

A → B → C

A → B → D → E → C → F → G

A → B → D → H → I → E → J → C → F → K → G

### For 8-Puzzle using Displace & Manhattan

Set the initial state =

1	2	3
4	0	5
6	7	8

Goal state =

1	2	3
4	5	6
7	8	0

The no of tiles not in its goal state = 5  
which are 5, 6, 7, 8, 0

Manhattan distance for each misplaced tiles will be the sum of the number of moves (up/down/left/right).

For 5: 1 (1 move left)  
6: 3 (up, right, right)  
7: 1 (left)  
8: 1 (left)  
0: 2 (right, down)

```
def possible-moves(state):
```

```
    moves = [] find-blank-position(state)
```

```
    blank-pos = position of our state
```

```
    if blank-pos is not in top row:
```

```
        moves.append('U')
```

```
    if blank-pos is not in bottom row:
```

```
        moves.append('D')
```

```
    if blank-pos is not in left column:
```

```
        moves.append('L')
```

```
    if blank-pos is not in right column:
```

```
        moves.append('R')
```

```
    return moves
```

```
def find-blank-position(state):
```

```
    for i from 0 to 2:
```

```
        for j from 0 to 2:
```

```
            if state[i][j] == 0:
```

```
                return (i, j)
```

```
def apply-move(state, move):
```

```
    new-blank-pos = 0
```

```
    blank-pos = find-blank-position(state)
```

```
    new-state = state
```

```
    if move == 'U':
```

```
        new-blank-pos = blank-pos - 3
```

```
    if move == 'D':
```

```
        new-blank-pos = blank-pos + 3
```

```
    if move == 'L':
```

```
        new-blank-pos = blank-pos - 1
```

```
    if move == 'R':
```

```
        new-blank-pos = blank-pos + 1
```

O/P:-

Method: IDDFS

Initial state:

[1, 2, 3]

[4, 0, 5]

[7, 8, 0]

Goal state:

[1, 2, 3]

[4, 5, 6]

[7, 8, 0]

## Code:

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

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

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

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

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

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

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

```

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

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

max_depth = 50

solution = iddfs(initial_state, goal_state, max_depth)

if solution:
    print("\nIDDFS solution path:")
    for s in solution:
        print(s[:3])
        print(s[3:6])
        print(s[6:])
        print()
else:
    print(f"\nNo solution found within the maximum depth of {max_depth}.")

```

## Output:

```

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

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

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

IDDFS solution path:
283
164
705

283
104
765

203
184
765

023
184
765

123
804
765

123
804
765

```

### Program3

Implement A\* search algorithm

#### Algorithm:

Lab 3  
8 puzzle using A\* search algorithm

Pseudocode

function find\_start(start, goal):  
 if ~~start~~ start is not solvable:  
 return no solution

open\_list ← [start]  
 closed\_list ← []  
 g(start) = 0  
 h(start) = misplaced\_tiles(start)  
 f(start) = g(start) + h(start)  
 while open\_list is not empty:  
 current ← state in open\_list with smallest f  
 if current == goal:  
 return path from start to current  
 remove current from open\_list  
 move current to closed\_list  
 for each neighbour of current:  
 if neighbour not in open\_list:  
 g(neighbour) = g(current) + 1  
 h(neighbour) = misplaced\_tiles(neighbour)  
 f(neighbour) = g(neighbour) + h(neighbour)  
 add neighbour to open\_list  
 return No Path Found

For example:-  
 Goal state =  $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & - \end{bmatrix}$

Given Initial state =  $\begin{bmatrix} 1 & 2 & 3 \\ - & 4 & 6 \\ 7 & 5 & 8 \end{bmatrix}$  (Complex)

Applying A\* algorithm, let g = no of steps  
 h = misplaced tiles  
 f = sum of g & h

$g=0, h=3, f=g+h=3$

Search tree diagram showing states and their g, h, f values. The goal state is achieved at the bottom.

O/P:-

Enter the start state matrix

1	2	3
-	4	6
7	5	8

Enter the goal state matrix

1	2	3
4	5	6
7	8	-

Output shows the sequence of states and the final goal state achieved.

Time Complexity

Misplaced Tiles  $O(b^d)$

Manhattan distance  $O(b^{(d-E)})$

IDDFS  $O(b^d)$

A\*  $O(b^d)$

**Code:**

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

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

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

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

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

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

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

```

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

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

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

    inv = inversion_count(start_perm)

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

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

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

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

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

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

    counter = count() # tie-breaker

    h0 = misplaced_heuristic(start, goal)

```

```

g_score = {start: 0}
f0 = h0

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

expansions = 0

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

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

    expansions += 1

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

    return None, None, expansions, len(closed)

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

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

def main():

```

```

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

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

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

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

    print(f'Total cost (number of moves): {cost}')
    print(f'Nodes expanded: {expansions}')
    print(f'Nodes explored (unique): {explored}')
except Exception as e:
    print("Error:", e)

if __name__ == "__main__":
    main()

```

## Output:

```

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

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

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

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

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

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

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

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

Total cost (number of moves): 5
Nodes expanded: 6
Nodes explored (unique): 7

```

**Code:**

#MANHATTAN DISTANCE

import heapq

from itertools import count

def misplaced\_heuristic(board, goal):

misplaced = 0

n = len(board)

for i in range(n):

for j in range(n):

if board[i][j] != 0 and board[i][j] != goal[i][j]:

misplaced += 1

return misplaced

def manhattan\_heuristic(board, goal):

n = len(board)

# Map goal positions for each tile

goal\_pos = {}

for i in range(n):

for j in range(n):

goal\_pos[goal[i][j]] = (i, j)

dist = 0

for i in range(n):

for j in range(n):

val = board[i][j]

if val != 0:

gi, gj = goal\_pos[val]

dist += abs(i - gi) + abs(j - gj)

return dist

def find\_blank(board):

n = len(board)

for i in range(n):

for j in range(n):

if board[i][j] == 0:

return i, j

raise ValueError("Board does not contain a blank tile (0)")

def neighbors(board):

n = len(board)

x, y = find\_blank(board)

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

res = []

for dx, dy in dirs:

nx, ny = x + dx, y + dy

if 0 &lt;= nx &lt; n and 0 &lt;= ny &lt; n:

b = [list(row) for row in board]

b[x][y], b[nx][ny] = b[nx][ny], b[x][y]

res.append(tuple(tuple(row) for row in b))

return res

```

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

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

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

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

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

    inv = inversion_count(start_perm)

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

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

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

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

```

```

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

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

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

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

expansions = 0

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

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

    expansions += 1

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

return None, None, expansions, len(closed)

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

```

```

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

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

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

        result = a_star_manhattan(initial, goal)
        path, cost, expansions, explored = result
        if path is None:
            print("No solution (unsolvable with given start/goal).")
            return
        print("\nSolution path (each state shows g, h, f):\n")
        for idx, state in enumerate(path):
            g = idx
            h = manhattan_heuristic(state, tuple(tuple(r) for r in goal))
            f = g + h
            print(f'Step {idx}: g={g}, h={h}, f={f}')
            print_board(state)
            print()
        print(f'Total cost (number of moves): {cost}')
        print(f'Nodes expanded: {expansions}')
        print(f'Nodes explored (unique): {explored}')
    except Exception as e:
        print("Error:", e)
if __name__ == "__main__":
    main()

```

## OUTPUT:

```

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

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

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

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

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

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

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

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

Total cost (number of moves): 5
Nodes expanded: 5
Nodes explored (unique): 6

```

## Program4

### Implement Hill Climbing search algorithm to solve N-Queens problem

### Algorithm:

Lab 4 E/10/25

hill climbing search algorithm to solve  
N-Queens problem.

Algorithm / Pseudocode

def hill-climbing( $N$ ):  
  expect until solution found or max restraints  
  reached.  
  currentstate  $\leftarrow$  randomly generated board of  
  size  $N$ .  
  current-h  $\leftarrow$  heuristic(current-state)  
  loop:  
    neighbours  $\leftarrow$  all neighbour states of current-state  
    best-neigh  $\leftarrow$  state in neighbours with minimum  
    heuristic value  
    best-h  $\leftarrow$  heuristic(best-neigh)  
    if best-h  $>$  current-h:  
      break  
  end if  
  current-state  $\leftarrow$  best-neigh  
  current-h  $\leftarrow$  best-h  
  if current-h = 0 then  
    return currentstate  
  end if  
end loop  
return Failed

def Acoustic (rattle):

$hc = 0$

for i from 1 to N:

for j from i+1 to N:

if (queue i and j are in same row) OR  
((row[i] - row[j]) = (d-j)) then

$hc = hc + 1$

solution h

O/P-  
solution found after 1 constant!

a . . . . .  
. . a . . . . .  
a . . . . .  
. . a . . . . .  
a . . . . .  
. . a . . . . .  
a . . . . .  
. . a . . . . .  
a . . . . .  
. . a . . . . .

**Code:**

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

def generate_neighbors(state):
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if state[col] != row: # move queen
```

```

        new_state = list(state)
        new_state[col] = row
        neighbors.append(new_state)
    return neighbors

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

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

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

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

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

        if best_cost >= current_cost:
            break

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

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

initial_state = [3, 1, 2, 0]
hill_climbing(initial_state)

```

### Output:

```

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

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

```

## Program 5

Simulated Annealing to Solve 8-Queens problem

### Algorithm:

lab 5  
8/10/25

Simulated Annealing for N-Queens  
Pseudocode

Formula  
where

$$P = e^{-\Delta E / KT}$$

$K$  = cooling factor  
 $T$  = Temperature  
 $\Delta E$  = Change cost  
 $P$  = Probability of change

```
def simulated_ann(initial_state):  
    current = initial_state  
    T = initial temp (for eg: 100)  
    while T > 0:  
        neighbours = get_neighbours(current)  
        if (neighbours == empty):  
            return current  
        for ne in neighbours:  
             $\Delta E$  = cost(neighbours)  
            if  $\Delta E < 0$  or  $\text{random}() < e^{-\Delta E / KT}$ :  
                current = neighbour  
        T = T * K
```

O/P:- solution found in 652 iterations

.	.	Q	.	.
.	.	.	Q	.
Q	.	.	.	.
.	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] or abs(state[i] - state[j]) == abs(i - j):  
                cost += 1
```

```

    return cost

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

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

    current = [random.randint(0, n - 1) for _ in range(n)]
    current_cost = calculate_cost(current)
    best = current
    best_cost = current_cost
    temperature = initial_temp

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

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

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

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

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

    return best, best_cost

best_state, best_cost = simulated_annealing()

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

```

### Output:

```

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

```

## Program 6

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

### **Algorithm:**

Lab 6  
Propositional Logic  
A knowledge base (KB) :- set of propositional sentences  
A query represented by  $\alpha$   
We check all possible combinations of truth values for all propositional symbols  
For each possible model  
→ if model makes KB true, then we need check if  $\alpha$  is also true  
→ if there is any model where KB is true but  $\alpha$  is false, then KB does not entail  $\alpha$   
→ if KB is true,  $\alpha$  is true, KB entails  $\alpha$   
Algorithm steps  
→ Extract all propositional symbols from KB &  $\alpha$   
→ Generate all possible truth assignments  
→ Check entailment condition  
if KB = true &  $\alpha$  should also be true.  
→ Return result  
True → KB entails  $\alpha$   
False → KB does not entail  $\alpha$

Pseudocode  
TT-ENTAILS(KB,  $\alpha$ ):  
symbols  $\leftarrow$  all unique propositional symbols in KB &  $\alpha$   
return TT-CHECK-ALL(KB,  $\alpha$ , symbols, {})  
TT-CHECK-ALL(KB,  $\alpha$ , symbols, model):  
if symbols is empty:  
if PL-TRUE(KB, model):  
return PL-TRUE( $\alpha$ , model)  
else:  
return TRUE  
else:  
P  $\leftarrow$  first symbol in symbols  
rest  $\leftarrow$  symbols - {P}  
return (TT-CHECK-ALL(KB,  $\alpha$ , rest, model  $\cup$  {P=TRUE})  
and  
TT-CHECK-ALL(KB,  $\alpha$ , rest, model  $\cup$  {P=FALSE}))  
PL-TRUE(sentence, model):  
Evaluate the propositional sentence using truth value from model.  
Q. Consider a knowledge base KB that contains the following propositional logic sentences:  
 $A \rightarrow P$   
 $P \rightarrow \neg A$   
 $A \vee R$

i) Construct a truth table that shows the truth value of each sentence in KB & indicate the models in which the KB is true

(ii) Does KB entail R

(iii) Does KB entail  $R \rightarrow P$ ?

(iv) Does KB entail  $A \rightarrow R$ ?

Ans

(i)

P	Q	R	$(\neg Q \vee P)$ $A \rightarrow P$	$(\neg P \vee A)$ $P \rightarrow \neg A$	$A \vee R$	KB
F	F	F	T	T	F	F
F	F	T	T	T	T	F
F	T	F	F	T	T	F
F	T	T	F	T	F	F
T	F	F	T	T	F	T
T	F	T	T	T	T	F
T	T	F	T	F	T	F
T	T	T	T	F	T	F

(ii) KB entails R

P & Q	R	$\rightarrow$ KB entails R
(F, F, T)	T	
(T, F, T)	T	

(iii) KB entails  $R \rightarrow P$

P & Q	R	$\rightarrow$ KB does not entail $R \rightarrow P$
(F, F, T)	F	
(T, F, T)	T	

(iv) Does KB entail  $A \rightarrow R$  ( $\neg A \vee R$ )

P	Q	R	$A \rightarrow R$
(F, F, T)	T		
(T, F, T)	T		

KB entails  $A \rightarrow R$

*Shah*

### Code:

```
import itertools
```

```
def eval_expr(expr, model):
```

```
    try:
```

```
        return eval(expr, {}, model)
```

```
    except:
```

```
        return False
```

```
def tt_entails(KB, query):
```

```
    symbols = sorted(set([ch for ch in KB + query if ch.isalpha()]))
```

```
    print("\nTruth Table:")
```

```
    print("| ".join(symbols) + " | KB | Query")
```

```
    print("-" * (6 * len(symbols) + 20))
```

```
    entails = True
```

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

```
        model = dict(zip(symbols, values))
```

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

```

query_val = eval_expr(query, model)

row = " | ".join(["T" if model[s] else "F" for s in symbols])
print(f'{row} | {kb_val} | {query_val}')

if kb_val and not query_val:
    entails = False

return entails

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

result = tt_entails(KB, query)

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

```

## Output:

```

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

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

Result:
KB entails Query (True in all cases).

```

## Program 7

Implement unification in first order logic

### Algorithm:

Week 7 First Order Logic  
Implement unification in first order logic  
Unification - Process of finding a common substitution for variables in different terms to make them match.  
Goal is to make 2 expressions identical by assigning values to variables in a way that preserves their meanings  
Conditions -  
→ The predicate symbol must be the same.  
eg: Unify  $(P(x, f(y)), P(a, f(z)))$   
fails because symbols  $x$  &  $a$  are different  
→ The number of arguments in both expressions must be identical  
→ If two similar variables are found in the same expression, then unification fails.  
eg:  $f(f(a), g(x)) = f(x, x)$  would require  $f(a) = x$  &  $g(x) = x$  but if  $g$  and  $x$  are distinct, so it fails.  
Algorithm / Pseudocode  
Unify  $(\varphi_1, \varphi_2)$   
step 1: If  $\varphi_1$  or  $\varphi_2$  is a variable or constant, then  
a) If  $\varphi_1$  or  $\varphi_2$  are identical, then return  $\varphi_1$   
b) Else if  $\varphi_1$  is a variable,

$f(x) = f(g(z))$   
 $x = g(z)$   
 $g(y) = g(f(a))$   
 $cy = f(a)$   
The last two expressions holds  
 $x = y$   
2)  $\mathcal{U}(x, f(y))$   
 $\mathcal{U}(f(y), cy)$   
 $x / f(y)$   
 $cy / f(x)$   
 $x$  appears in both  
 $cy$  appears in both  
→ Not unifiable  
3)  $\mathcal{U}(x, g(x))$   
 $\mathcal{U}(g(y), g(g(z)))$   
 $x / g(y)$   
 $\mathcal{U}(g(y), g(g(y)))$   
 $\mathcal{U}(g(y), g(g(z)))$   
 $\Rightarrow y = z$   
Unifiable

### Code:

```
def occurs_check(var, term, subst):  
    if var == term:
```

```

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

def unify(x, y, subst):
    if subst is None:
        return None
    elif x == y:
        return subst
    elif isinstance(x, str) and x.isupper():
        return unify_var(x, y, subst)
    elif isinstance(y, str) and y.isupper():
        return unify_var(y, x, subst)
    elif isinstance(x, tuple) and isinstance(y, tuple):
        if x[0] != y[0] or len(x) != len(y):
            return None
        for a, b in zip(x[1:], y[1:]):
            subst = unify(a, b, subst)
            if subst is None:
                return None
        return subst
    else:
        return None

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

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

```

```

        if c == '(':
            depth += 1
        elif c == ')':
            depth -= 1
        current += c
    if current:
        args.append(parse_expr(current))
    return tuple([name] + args)

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

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

expr1 = parse_expr(expr1_input)
expr2 = parse_expr(expr2_input)

subst = unify(expr1, expr2, {})

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

print("Most General Unifier (MGU):", formatted_subst)

```

### Output:

```

➡ Enter first expression: p(b,X,f(g(Z)))
Enter second expression: p(Z,f(Y),f(Y))
Most General Unifier (MGU): {'Z': 'b', 'X': 'f(Y)', 'Y': 'g(Z)'}

```

## Program 8

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

### Algorithm:

Lab 8 First Order Logic

Create a knowledge base consisting of FOL statements and prove the given query using resolution.

Basic steps for providing a conclusion S given premises:

Premises, ..., Premise<sub>n</sub>

Call expressed in FOL

Algorithm →

- (i) Convert all sentences to CNF.
- (ii) Negate conclusion S and convert result to CNF.
- (iii) Add negated contradiction S to the premise clauses.
- (iv) Repeat until contradiction or no progress is made:
  - (a) select 2 clauses
  - (b) resolve them together, performing all required unifications.
- (v) If the resolvent is the empty clause, a contradiction has been found.

(b) If not, add resolvent to the premises.

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

Steps to convert logic statement to CNF

Eliminate biconditional & implications:

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

Eliminate  $\Rightarrow$ , replacing  $\alpha \Rightarrow \beta$  with  $\neg \alpha \vee \beta$

Move  $\neg$  inwards:

$\neg(\forall x \phi) \equiv \exists x \neg \phi$ ,  
 $\neg(\exists x \phi) \equiv \forall x \neg \phi$ ,  
 $\neg(\alpha \vee \beta) \equiv \neg \alpha \wedge \neg \beta$ ,  
 $\neg(\alpha \wedge \beta) \equiv \neg \alpha \vee \neg \beta$ ,  
 $\neg \neg \alpha \equiv \alpha$

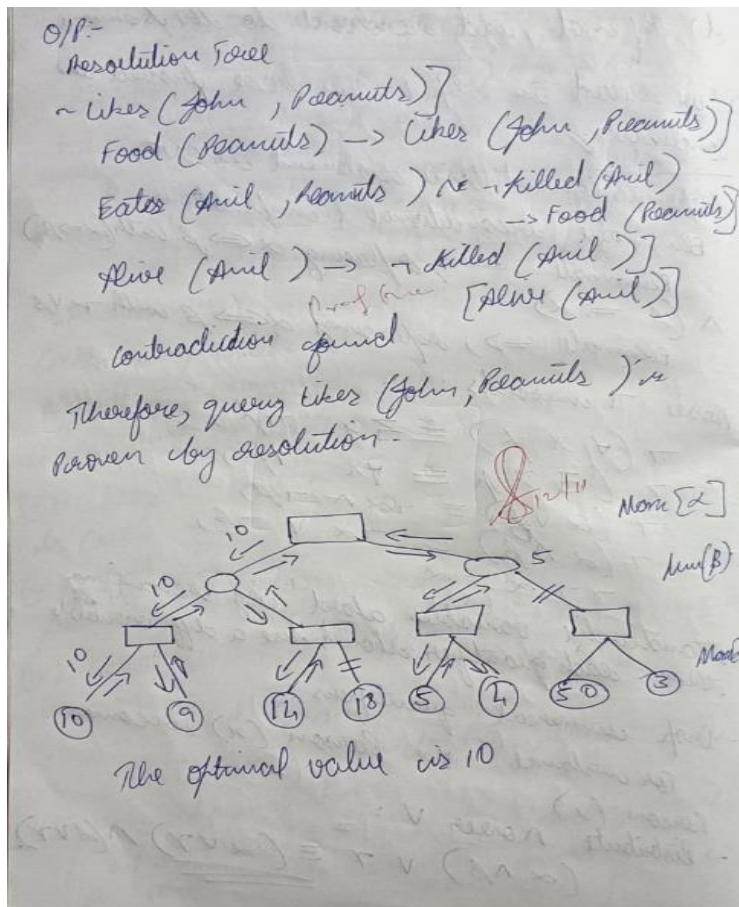
Standardize variables apart by renaming them: each quantifier should use a diff variable.

→ Drop universal quantifiers

For instance  $\forall x \text{ Person}(x)$  becomes  $\text{Person}(x)$

→ Distribute  $\neg$  over  $\vee$ :

$(\alpha \wedge \beta) \vee \gamma \equiv (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$



### Code:

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

If American(X) and Hostile(Y) and Sells\_Weapons(X, Y), then Crime(X)  
def forward\_reasoning(facts):

```
    If American(X) and Hostile(Y) and Sells_Weapons(X, Y), then Crime(X)
    if facts.get('American(Robert)', False) and facts.get('Hostile(A)', False) and facts.get('Sells_Weapons(Robert, A)', False):
        facts['Crime(Robert)'] = True
```

```
forward_reasoning(facts)
```

```
if facts.get('Crime(Robert)', False):
    print("Robert is a criminal.")
else:
    print("Robert is not a criminal.")
```

### Output:

Robert is a criminal.

### Program 9

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

#### Algorithm:

Forward reasoning algorithm

function FOL-FC-ASK( $KB, \alpha$ ) returns  
substitution or false

inputs -  $KB, \alpha$

local variables : news, inferred on each iteration

repeat until news is empty

news  $\leftarrow \{ \}$

for each rule in  $KB$  do

for each  $\theta$  such that  $SUBST(\theta, \text{rule})$

$= SUBST(\theta, \text{rule})$

you have  $\theta, \dots$  in  $KB$

$q' \rightarrow SUBST(\theta, q)$

if  $q'$  does not unify then

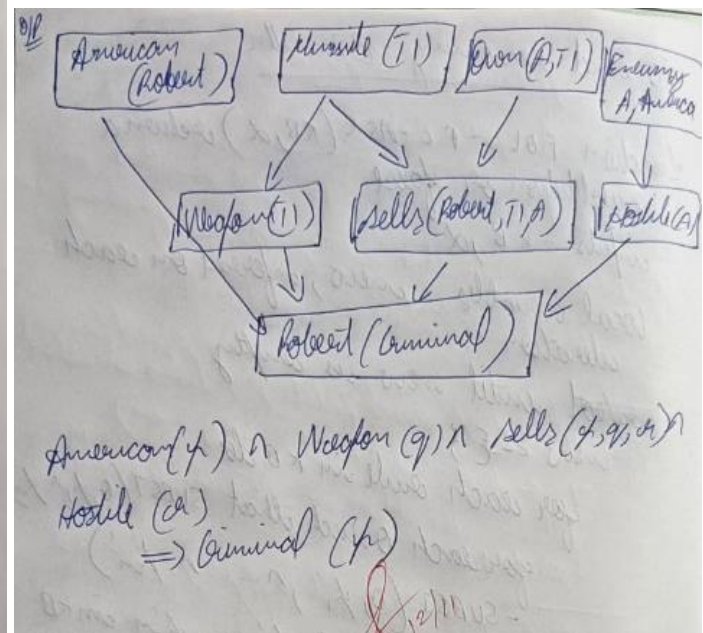
add  $q'$  to news

$\phi \leftarrow \text{unify}(q', \phi)$

if  $\phi$  is not false then action  $\phi$

add news to  $KB$

return False



#### Code:

import copy

```
# -----  
# Predicate Structure  
# -----
```

class Predicate:

def \_\_init\_\_(self, name, args, negated=False):

self.name = name

self.args = args if isinstance(args, tuple) else tuple(args)

self.negated = negated

```

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

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

def __repr__(self):
    neg = "~" if self.negated else ""
    args_str = ",".join(str(a) for a in self.args)
    return f"{neg}{self.name}({args_str})"

def negate(self):
    return Predicate(self.name, self.args, not self.negated)

def substitute(self, theta):
    """Apply substitution theta to this predicate"""
    new_args = tuple(substitute_term(arg, theta) for arg in self.args)
    return Predicate(self.name, new_args, self.negated)

def substitute_term(term, theta):
    """Apply substitution to a term"""
    if isinstance(term, str) and term.islower(): # variable
        if term in theta:
            return substitute_term(theta[term], theta)
        return term
    elif isinstance(term, tuple):
        return tuple(substitute_term(t, theta) for t in term)
    return term

# -----
# Unification Algorithm
# -----
def unify(x, y, theta=None):
    if theta is None:
        theta = {}
    if theta == "FAIL":
        return "FAIL"
    elif x == y:
        return theta
    elif isinstance(x, str) and x.islower(): # variable
        return unify_var(x, y, theta)
    elif isinstance(y, str) and y.islower(): # variable
        return unify_var(y, x, theta)
    elif isinstance(x, tuple) and isinstance(y, tuple):
        if len(x) != len(y):
            return "FAIL"
        theta = unify(x[0], y[0], theta)
        if theta == "FAIL":
            return "FAIL"

```

```

        return unify(x[1:], y[1:], theta)
    else:
        return "FAIL"

def unify_var(var, x, theta):
    if var in theta:
        return unify(theta[var], x, theta)
    elif isinstance(x, str) and x.islower() and x in theta:
        return unify(var, theta[x], theta)
    elif occurs_check(var, x, theta):
        return "FAIL"
    else:
        new_theta = copy.deepcopy(theta)
        new_theta[var] = x
        return new_theta

def occurs_check(var, x, theta):
    if var == x:
        return True
    elif isinstance(x, str) and x.islower() and x in theta:
        return occurs_check(var, theta[x], theta)
    elif isinstance(x, tuple):
        return any(occurs_check(var, xi, theta) for xi in x)
    return False

# -----
# Variable Standardization
# -----
var_counter = 0

def standardize_variables(clause):
    """Rename all variables in a clause to unique names"""
    global var_counter
    mapping = {}
    new_clause = []

    for pred in clause:
        new_args = []
        for arg in pred.args:
            if isinstance(arg, str) and arg.islower(): # variable
                if arg not in mapping:
                    mapping[arg] = f"{arg}{var_counter}"
                    var_counter += 1
                new_args.append(mapping[arg])
            else:
                new_args.append(arg)
        new_clause.append(Predicate(pred.name, new_args, pred.negated))

    return new_clause

# -----

```

```

# Resolution Algorithm
# -----
def resolve(ci, cj):
    """Resolve two clauses using FOL resolution"""
    ci = standardize_variables(ci)
    cj = standardize_variables(cj)

    resolvents = []

    for i, pi in enumerate(ci):
        for j, pj in enumerate(cj):
            # Check if predicates can be resolved (opposite signs, same name)
            if pi.negated != pj.negated and pi.name == pj.name:
                # Try to unify the arguments
                theta = unify(pi.args, pj.args)

                if theta != "FAIL":
                    # Create resolvent by removing resolved predicates and applying substitution
                    new_clause = []

                    # Add literals from ci except pi
                    for k, pred in enumerate(ci):
                        if k != i:
                            new_clause.append(pred.substitute(theta))

                    # Add literals from cj except pj
                    for k, pred in enumerate(cj):
                        if k != j:
                            new_clause.append(pred.substitute(theta))

                    # Remove duplicates
                    new_clause = list(set(new_clause))
                    resolvents.append(new_clause)

    return resolvents

def fol_resolution(kb, query):
    """FOL resolution algorithm"""
    # Negate query and add to KB
    clauses = [clause[:] for clause in kb] # deep copy
    clauses.append([query.negate()])

    print(f"\nKnowledge Base + Negated Query:")
    for i, clause in enumerate(clauses):
        print(f" {i+1}. {clause}")
    print()

    iteration = 0
    while True:
        iteration += 1
        n = len(clauses)
        pairs = [(clauses[i], clauses[j]) for i in range(n) for j in range(i + 1, n)]

```

```

new_clauses = []
for (ci, cj) in pairs:
    resolvents = resolve(ci, cj)

    for resolvent in resolvents:
        if len(resolvent) == 0:
            print(f"Iteration {iteration}: Derived empty clause from:")
            print(f"  {ci}")
            print(f"  {cj}")
            print(f"  → [] (Contradiction found!)")
            return True

        # Check if this is a new clause
        if resolvent not in clauses and resolvent not in new_clauses:
            new_clauses.append(resolvent)

if not new_clauses:
    print(f"Iteration {iteration}: No new clauses derived. Query cannot be proved.")
    return False

print(f"Iteration {iteration}: Generated {len(new_clauses)} new clause(s)")
for clause in new_clauses:
    clauses.append(clause)

# -----
# Example Usage
# -----
if __name__ == "__main__":
    # Define knowledge base
    kb = [
        # John likes all food: Food(x) => Likes(John, x)
        [Predicate("Food", ("x",), negated=True), Predicate("Likes", ("John", "x"))],

        # Food(Apple)
        [Predicate("Food", ("Apple",))],

        # Food(Vegetables)
        [Predicate("Food", ("Vegetables",))],

        # Eats(Anil, Peanuts)
        [Predicate("Eats", ("Anil", "Peanuts"))],

        # Alive(Anil)
        [Predicate("Alive", ("Anil",))],

        # If alive and eats something, that thing is food: Alive(x) ∧ Eats(x,y) => Food(y)
        [Predicate("Alive", ("x",), negated=True),
         Predicate("Eats", ("x", "y"), negated=True),
         Predicate("Food", ("y",))],

```

```

    # Harry eats everything Anil eats: Eats(Anil,y) => Eats(Harry,y)
    [Predicate("Eats", ("Anil", "y"), negated=True),
     Predicate("Eats", ("Harry", "y"))]
]

# Query: Does John like Peanuts?
query = Predicate("Likes", ("John", "Peanuts"))

print("=" * 60)
print("FIRST-ORDER LOGIC RESOLUTION THEOREM PROVER")
print("=" * 60)
print(f"\nQuery: {query}")
print("-" * 60)

result = fol_resolution(kb, query)

print("\n" + "=" * 60)
if result:
    print("Query is PROVED using resolution!")
else:
    print("Query CANNOT be proved.")
print("=" * 60)

```

### Output:

```

Query: Likes(John,Peanuts)
-----

Knowledge Base + Negated Query:
1. [~Food(x), Likes(John,x)]
2. [Food(Apple)]
3. [Food(Vegetables)]
4. [Eats(Anil,Peanuts)]
5. [Alive(Anil)]
6. [~Alive(x), ~Eats(x,y), Food(y)]
7. [~Eats(Anil,y), Eats(Harry,y)]
8. [~Likes(John,Peanuts)]

Iteration 1: Generated 8 new clause(s)
Iteration 2: Generated 16 new clause(s)
Iteration 3: Derived empty clause from:
[Eats(Anil,Peanuts)]
[~Eats(Anil,Peanuts)]
→ [] (Contradiction found!)

```

## Program 10

Implement Alpha-Beta Pruning.

### Algorithm:

Lab 9 Adversarial Search  
Implement Alpha Beta Pruning  
 A modified variant of minimax method as  $\alpha$ - $\beta$  pruning. It's a way for improving the minimax algorithm.

Algorithm  
 function  $\alpha$ - $\beta$  search (state) returns an action  
 $v \leftarrow \text{Min-Value}(\text{state}, -\infty, +\infty)$   
 return the action in  $\text{ACTIONS}(\text{state})$  with value  $v$

function  $\text{MAX-Value}(\text{state}, \alpha, \beta)$  returns a utility value  
 if  $\text{Terminal-Test}(\text{state})$  then return  $\text{Utility}(\text{state})$   
 $v \leftarrow -\infty$   
 for each  $a$  in  $\text{Actions}(\text{state})$  do  
 $v \leftarrow \text{Max}(v, \text{Min-Value}(\text{Result}(a, \text{state}), \alpha, \beta))$   
 if  $v \geq \beta$  then return  $v$   
 $\alpha \leftarrow \text{Max}(\alpha, v)$   
 return  $v$

function  $\text{Min-Value}(\text{state}, \alpha, \beta)$  returns a utility value  
 if  $\text{Terminal-Test}(\text{state})$  then return  $\text{Utility}(\text{state})$   
 $v \leftarrow +\infty$   
 for each  $a$  in  $\text{Actions}(\text{state})$  do  
 $v \leftarrow \text{Min}(v, \text{Max-Value}(\text{Result}(a, \text{state}), \alpha, \beta))$   
 if  $v \leq \alpha$  then return  $v$   
 $\beta \leftarrow \text{Min}(\beta, v)$   
 return  $v$

O/P Initial : 

1	2	3
4	5	6
7	0	8

Min Max Algorithm

Best value for  $\alpha$ - $\beta$  : 9 & 2nd

### Code:

```
import math

# Alpha-Beta Pruning Algorithm
def alpha_beta(depth, node_index, maximizing_player, values, alpha, beta, max_depth, path, pruned):
    # Base case: leaf node
    if depth == max_depth:
        return values[node_index], [node_index]

    if maximizing_player:
        best = -math.inf
        best_path = []
        for i in range(2): # two children per node
            val, child_path = alpha_beta(depth + 1, node_index * 2 + i, False, values, alpha, beta, max_depth, path, pruned)
            if val > best:
                best = val
                best_path = [node_index] + child_path
            alpha = max(alpha, best)
            if beta <= alpha:
                pruned.append((node_index, "Right" if i == 0 else "Left"))
```

```

        break
    return best, best_path
else:
    best = math.inf
    best_path = []
    for i in range(2):
        val, child_path = alpha_beta(depth + 1, node_index * 2 + i, True, values, alpha, beta, max_depth, path, pruned)
        if val < best:
            best = val
            best_path = [node_index] + child_path
        beta = min(beta, best)
        if beta <= alpha:
            pruned.append((node_index, "Right" if i == 0 else "Left"))
            break
    return best, best_path

# Example usage
if __name__ == "__main__":
    # Example game tree (leaf node values)
    values = [3, 5, 6, 9, 1, 2, 0, -1]

    print("Leaf Node Values:", values)
    path = []
    pruned = []

    max_depth = 3
    result, best_path = alpha_beta(0, 0, True, values, -math.inf, math.inf, max_depth, path, pruned)

    print("\nOptimal Value at Root Node:", result)
    print("Best Path (Node Indices):", best_path)
    print("Pruned Nodes:", pruned)

```

### Output:

```

... Leaf Node Values: [3, 5, 6, 9, 1, 2, 0, -1]

Optimal Value at Root Node: 5
Best Path (Node Indices): [0, 0, 0, 1]
Pruned Nodes: [(1, 'Right'), (1, 'Right')]

```