

**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.

Sandhya A Kulkarni 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

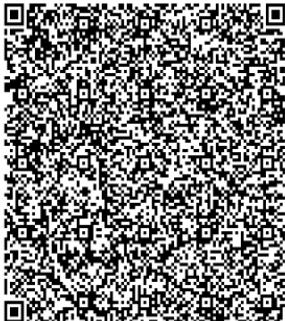
The certificate is awarded to

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

TICTACTOE PROGRAM

PSEUDO Code / Algorithm

```

TicTacToe(board) {
    # Point the board
    for i in range(3):
        for j in range(3):
            point(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
        point("Player 1 enter position :")
        if board[input] != '-':
            # take input again
            board[input] = "X"
            point("Player 2 enter the position :")
        if board[input] != "-":
            # take input again
            board[input] = "O"
}

```

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

The possibilities

$$W = \begin{bmatrix} [1, 2, 3], [4, 5, 6], [7, 8, 9], [1, 4, 7], \\ [2, 5, 8], [3, 6, 9], [1, 5, 9], [3, 5, 7] \end{bmatrix}$$

```

if (player == "Player 1")
    point("Player 1 wins")
if (player == "Player 2")
    point("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 mixed up Then there is a tie breaker!!!

O/P:

X X	X X	X X	X X
X X	X X	X X	X X
X X	X X	X X	X X

$\begin{array}{|c|c|} \hline X & X \\ \hline X & X \\ \hline \end{array}$
 $\begin{array}{|c|c|} \hline X & X \\ \hline X & X \\ \hline \end{array}$
 $\begin{array}{|c|c|} \hline X & X \\ \hline X & X \\ \hline \end{array}$
 $\begin{array}{|c|c|} \hline X & X \\ \hline X & X \\ \hline \end{array}$
 $\begin{array}{|c|c|} \hline X & X \\ \hline X & X \\ \hline \end{array}$
 X-1 O-7 X-5 O-8 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', ' ']
```

```
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

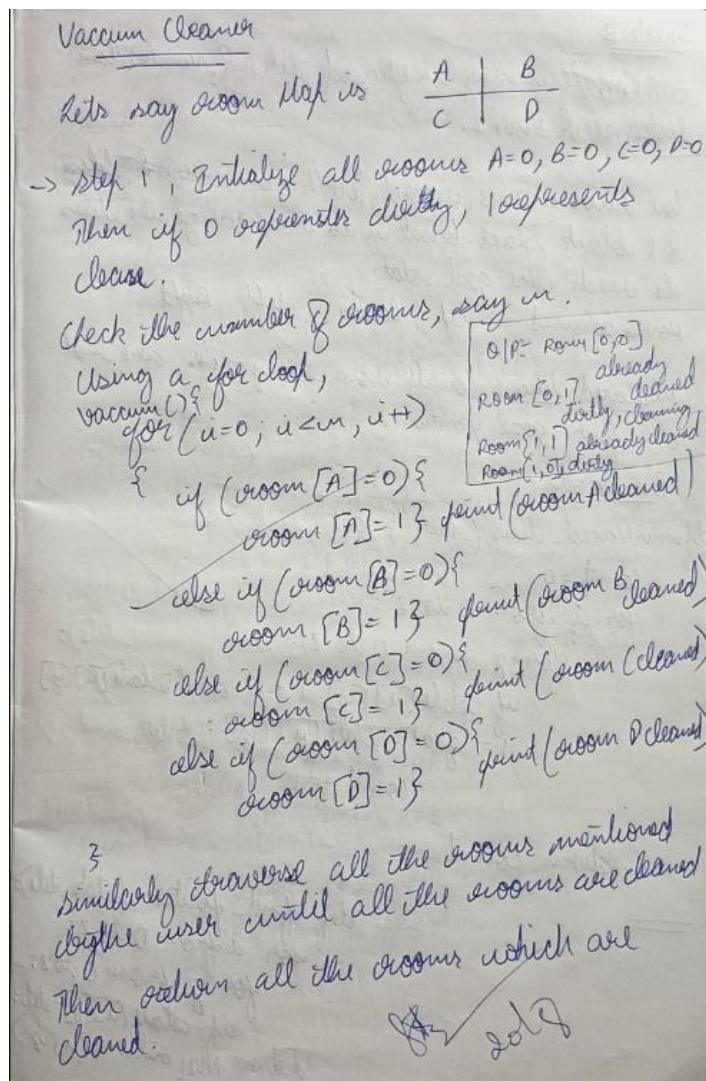
Current Board:

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

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

Implement vacuum cleaner agent

Algorithm:



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 3×3 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

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 distance 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(tile, goal_state)

 distance = distance +

 abs(i - goal_i) +
 abs(j - goal_j)

solution 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 = DLS(initial_state, goal_state)
        if result == SUCCESS:
            return result
        depth += 1

def DLS(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 = DLS(child, goal_state, limit - 1)
            if result == SUCCESS:
                return SUCCESS
            else if result == CUTOFF:
                return CUTOFF

```

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

For 5 :
 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 blank 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

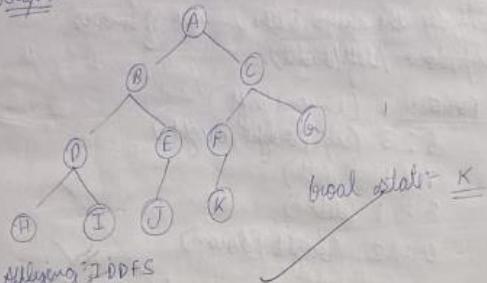
```

```

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)
    return NULL

```

Graph



Applying IDDFS

$A \rightarrow B \rightarrow C$
 $A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$
 $A \rightarrow B \rightarrow D \rightarrow H \rightarrow I \rightarrow E \rightarrow J \rightarrow C \rightarrow F \rightarrow K \rightarrow G$

for 8-Puzzle using 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

def apply_move(state, move):

~~new-blank-pos = 0~~
 blank_pos = find_blank_position
 new_state = state
 if move == 'U':
~~new-blank-pos = blank_pos + 0~~
 new-blank-pos = blank_pos - 3
 if move == 'D':
~~new-blank-pos = blank_pos + 3~~
 new-blank-pos = blank_pos + 3
 if move == 'L':
~~new-blank-pos = blank_pos - 1~~
 new-blank-pos = blank_pos - 1
 if move == 'R':
~~new-blank-pos = blank_pos + 1~~
 new-blank-pos = blank_pos + 1

O/P:

method : IDDFS

solved in 3 moves

$$\begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1, 2, 3 \\ 4, 0, 6 \\ 7, 5, 8 \end{bmatrix} \rightarrow \begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 0 \end{bmatrix}$$

$$\begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1, 2, 3 \\ 4, 0, 6 \\ 7, 5, 8 \end{bmatrix} \rightarrow \begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 0 \end{bmatrix}$$

$$\begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1, 2, 3 \\ 4, 0, 6 \\ 7, 5, 8 \end{bmatrix} \rightarrow \begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 0 \end{bmatrix}$$

$$\begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1, 2, 3 \\ 4, 0, 6 \\ 7, 5, 8 \end{bmatrix} \rightarrow \begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 0 \end{bmatrix}$$

$$\begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1, 2, 3 \\ 4, 0, 6 \\ 7, 5, 8 \end{bmatrix} \rightarrow \begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 0 \end{bmatrix}$$

$$\begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1, 2, 3 \\ 4, 0, 6 \\ 7, 5, 8 \end{bmatrix} \rightarrow \begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 0 \end{bmatrix}$$

$$\begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1, 2, 3 \\ 4, 0, 6 \\ 7, 5, 8 \end{bmatrix} \rightarrow \begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 0 \end{bmatrix}$$

$$\begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1, 2, 3 \\ 4, 0, 6 \\ 7, 5, 8 \end{bmatrix} \rightarrow \begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 0 \end{bmatrix}$$

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:])
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: 765

Enter the goal state (enter 3 digits per row, separated by spaces, 0 for empty):
Row 1: 123
Row 2: 864
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
765

283
164
765

283
164
765

023
184
765

123
084
765

123
864
765

123
864
765

```

Program3

Implement A* search algorithm

Algorithm:

Lab 3

8-puzzle using A* star algorithm

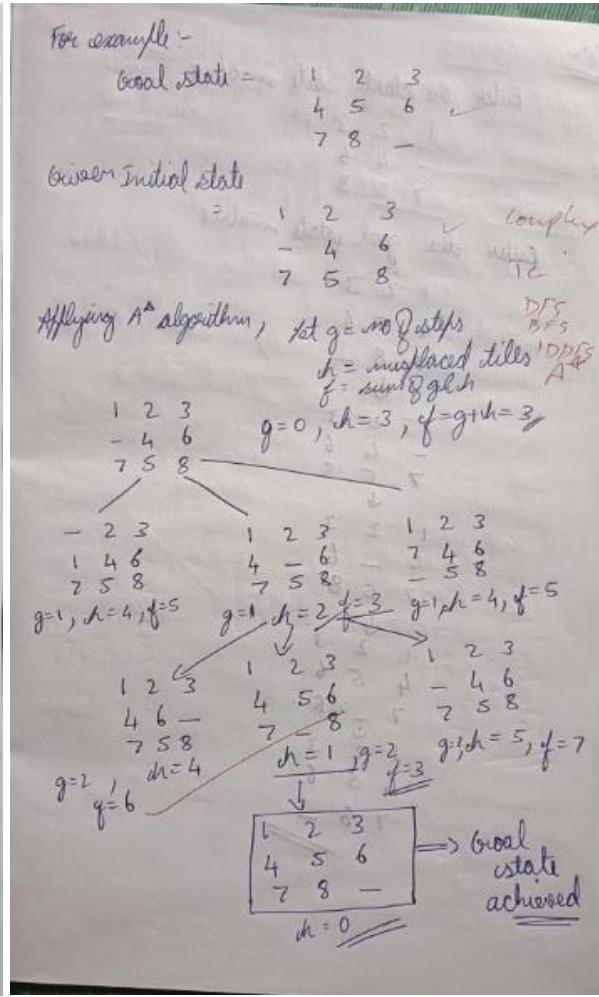
path search & graph search finding algorithm

Pseudocode

```

function start(start, goal):
    if 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)
    came_from = empty map
    while open_list is not empty:
        current = state in open_list with smallest f
        if current == goal:
            return path from came_from
        remove current from open_list
        move current to closed_list
        for neighbor in neighbors(current):
            if neighbor not in open_list:
                g(neighbor) = g(current) + 1
                if neighbor not in closed_list:
                    g(neighbor) = misplaced_tiles(neighbor)
                    h(neighbor) = manhattan(neighbor)
                    f(neighbor) = g(neighbor) + h(neighbor)
                    add neighbor to open_list
            else:
                if g(neighbor) < g(open_list[neighbor]):
                    g(neighbor) = g(current) + 1
                    h(neighbor) = manhattan(neighbor)
                    f(neighbor) = g(neighbor) + h(neighbor)
                    add neighbor to open_list
    return "No Path Found"

```



B/P -

Enter the start state matrix

$$\begin{matrix} 1 & 2 & 3 \\ - & 4 & 6 \\ 7 & 5 & 8 \end{matrix}$$

Enter the goal state matrix

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & - \end{matrix}$$

Initial state = $\begin{matrix} 1 & 2 & 3 \\ - & 4 & 6 \\ 7 & 5 & 8 \end{matrix}$

Goal state = $\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & - \end{matrix}$

Manhattan distance = $\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & - \end{matrix}$

A* search = $\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & - \end{matrix}$

Time Complexity

Misplaced Tiles	$O(d!)$
Manhattan distance	$O(d^{d-E})$
IDDFS	$O(d^d)$
A*	$O(d^d)$

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 6 4
7 0 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 <= nx < n and 0 <= ny < n:
            b = [list(row) for row in board]
            b[x][y], b[nx][ny] = b[nx][ny], b[x][y]
            res.append(tuple(tuple(row) for row in b))
    return res
```

```

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

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

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

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

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

    inv = inversion_count(start_perm)

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

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

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

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

```

```

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

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

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

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

expansions = 0

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

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

    expansions += 1

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

return None, None, expansions, len(closed)

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

```

```

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

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

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

        result = a_star_manhattan(initial, goal)
        path, cost, expansions, explored = result
        if path is None:
            print("No solution (unsolvable with given start/goal).")
            return
        print("\nSolution path (each state shows g, h, f):\n")
        for idx, state in enumerate(path):
            g = idx
            h = manhattan_heuristic(state, tuple(tuple(r) for r in goal))
            f = g + h
            print(f"Step {idx}: g={g}, h={h}, f={f}")
            print_board(state)
            print()
        print(f"Total cost (number of moves): {cost}")
        print(f"Nodes expanded: {expansions}")
        print(f"Nodes explored (unique): {explored}")
    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
1 4
7 0 5
Enter goal state row by row (use 0 for blank):
1 2 3
8 0 4
7 6 5

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

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

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

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

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

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

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

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

```

Program4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

Lab 4 £10/25
 Hill Climbing search algorithm to solve
 N -Queens problem.
Algorithm / Pseudocode
 def hill-climbing(N):
 repeat until solution found or max restarts
 reached.
 current-state ← randomly generated board of
 size N .
 current-h ← heuristic(current-state)
 loop:
 neighbours ← all neighbour states of current-state
 best-neigh ← state in neighbours with minimum
 heuristic value
 best-h ← heuristic(best-neigh)
 if best-h > current-h:
 break
 end if
 current-state ← best-neigh
 current-h ← best-h
 if current-h = 0 then
 return current-state
 end if
 end loop
 return Failed

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

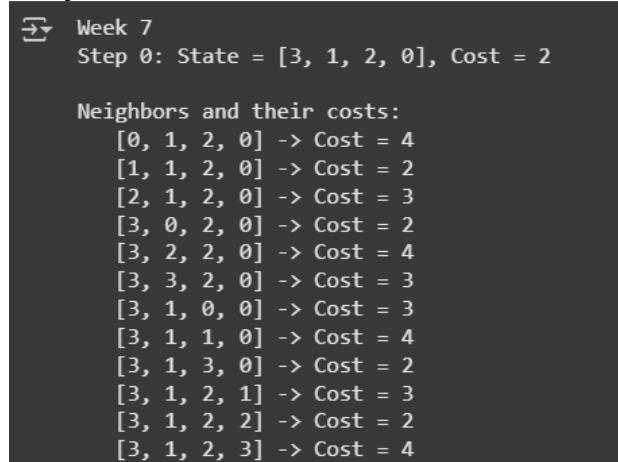
        step += 1
        current, current_cost = best_neighbor, best_cost
        print(f"\nStep {step}: Move to {current}, Cost = {current_cost}")

    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

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

K = cooling factor
T = Temperature
 ΔE = Change cost
 P = Probability of change

def simulated_anneal(initial_state):
 current = initial_state
 T = initial_temp (for eg: 100)
 while T > 0:
 neighbours = getNeighbours(current)
 if (neighbours == empty):
 return current
 for nl in neighbours:
 DE = cost(neighbour)
 if $\Delta E < 0$ or random()
 random() < $e^{-\Delta E / KT}$
 current = neighbour
 T = T * K

~~slights~~ ~~Process~~

O/P: solution found in 652 iterations

```

    . . Q . .
    . . . 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

Proposition logic

A knowledge base (KB) :- set of propositional sentences

A query represented by α

We check all possible combinations of truth values for all proposition symbols

For each possible model

- if model makes KB true, then we need check if α is also true
- if there is any model where KB is true but α is false, then KB does not entail α
- If KB is true, α is true, KB entails α

Algorithm steps

- Extract all propositional symbols from $KB \wedge \alpha$
- Generate all possible truth assignments
- Check entailment condition
if $KB = \text{true}$ & there should also be true.
- Return result

True $\rightarrow KB \text{ entails } \alpha$

False $\rightarrow KB \text{ does not entail } \alpha$

Pseudocode

$TT\text{-ENTAILS}(KB, \alpha)$:

- symbols \leftarrow all unique propositional symbols in $KB \cup \alpha$
- return $TT\text{-CHECK-ALL}(KB, \alpha, symbols, \{\})$

$TT\text{-CHECKALL}(KB, \alpha, symbols, model)$:

- if symbols is empty :
- if $PL\text{-TRUE}(KB, model)$:
 return $PL\text{-TRUE}(\alpha, model)$
- else :
 return TRUE
- else :
- $p \leftarrow$ first symbol in symbols
- $rest \leftarrow$ symbols - $\{p\}$
- return $(TT\text{-CHECK-ALL}(KB, \alpha, rest, model) \vee PL\text{-TRUE}(\alpha, model))$
- and
- $PL\text{-TRUE}(\text{sentence}, model)$:
- Evaluate the propositional sentence using truth value from model.

Q). Consider a knowledge base KB that contains the following propositional logic sentences:

$Q \rightarrow P$

$P \rightarrow \neg Q$

$Q \vee R$

- i) Construct a truth table that shows the truth value of each sentence in KB to 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 $\neg R \rightarrow R$?

Ans

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

- (ii) KB entails $\neg R \rightarrow R$
 P & R R \rightarrow KB entails $\neg R \rightarrow R$
 (F, F, T) T
 (T, F, T) T
 (iii) KB entails $R \rightarrow P$ ($\neg R \vee P$)
 P & R R $\rightarrow P$ ($\neg R \vee P$)
 (F, F, T) F \rightarrow KB does not entail $R \rightarrow P$
 (T, F, T)

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

~~Sticks~~
~~1 shot~~

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.
e.g. Unify $P(x, f(y)), P(a, f(z))$
fails because symbols P & f 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.
e.g.: $f(f(a), g(x)) = f(x, x)$ would
assign $f(a) \rightarrow x$, $g(x) \rightarrow x$ but f , g are
distinct, so it fails.

Algorithm / Pseudocode

Unify (ψ_1, ψ_2)

Step 1: If ψ_1 or ψ_2 is a variable or constant, then

- If ψ_1 or ψ_2 are identical, then return Nil
- Else if ψ_1 is a variable,

$f(x) = f(g(z))$
 $x = g(z)$
 $g(y) = g(f(a))$
 $y = f(a)$

The last two expression holds
 $x = y$

2) $\delta(x, f(y))$
 $\delta(f(y), y)$
 $x/f(y)$
 $y/f(x)$

x appears in both
 y appears in both
→ Not unifiable

3) $H(x, g(x))$
 $H(g(y), g(g(z)))$
 $x/g(y)$
 $H(g(y), g(g(z)))$
 $H(g(y), g(g(z)))$
 $\Rightarrow y/z$ 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:

```

```

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 proving a conclusion S given premises:

Premises, ... Premise_n

Call expressed in FOL

Algorithm \rightarrow

- (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.
 - (c) If the resultant is the empty clause, a contradiction has been found.

(d) if not, add resolution 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 $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$

$\wedge (\beta \Rightarrow \alpha)$

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

Move \neg univars:

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

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

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

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

$$\neg \neg \alpha = \alpha$$

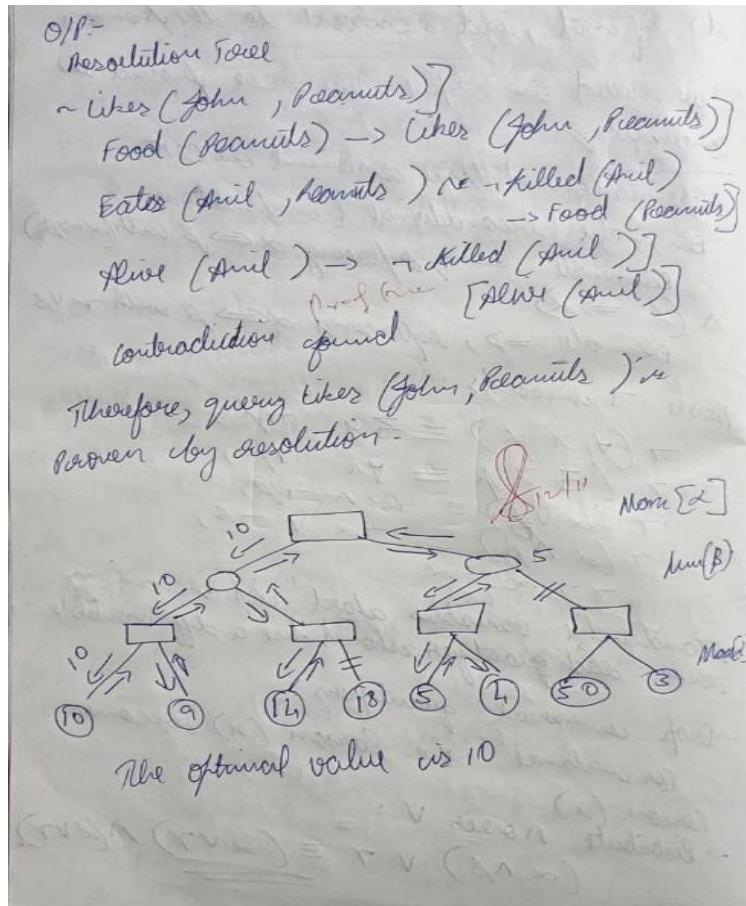
Standardize variables after by renaming them: each quantifier should use a diff variable.

Drop universal quantifiers

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

Distribute \neg over \vee :

$$(\alpha \wedge \beta) \vee \gamma \equiv \underline{\underline{(\alpha \vee \gamma) \wedge (\beta \vee \gamma)}}$$



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.")
  
```

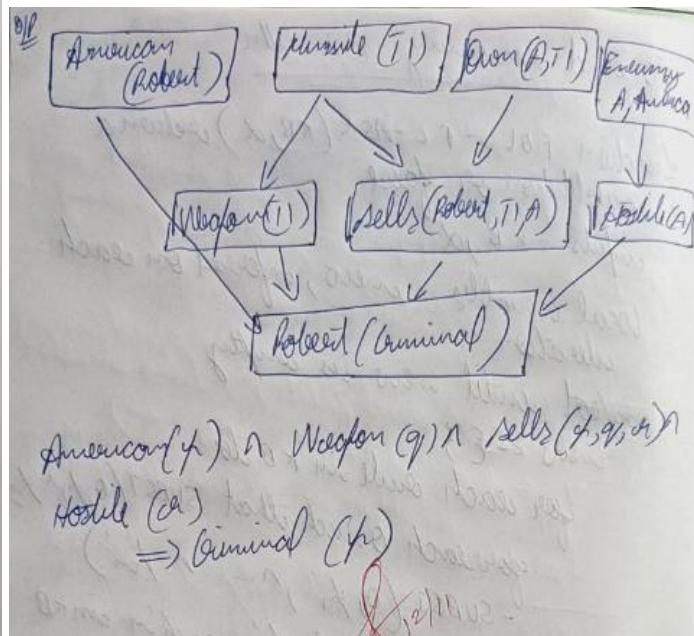
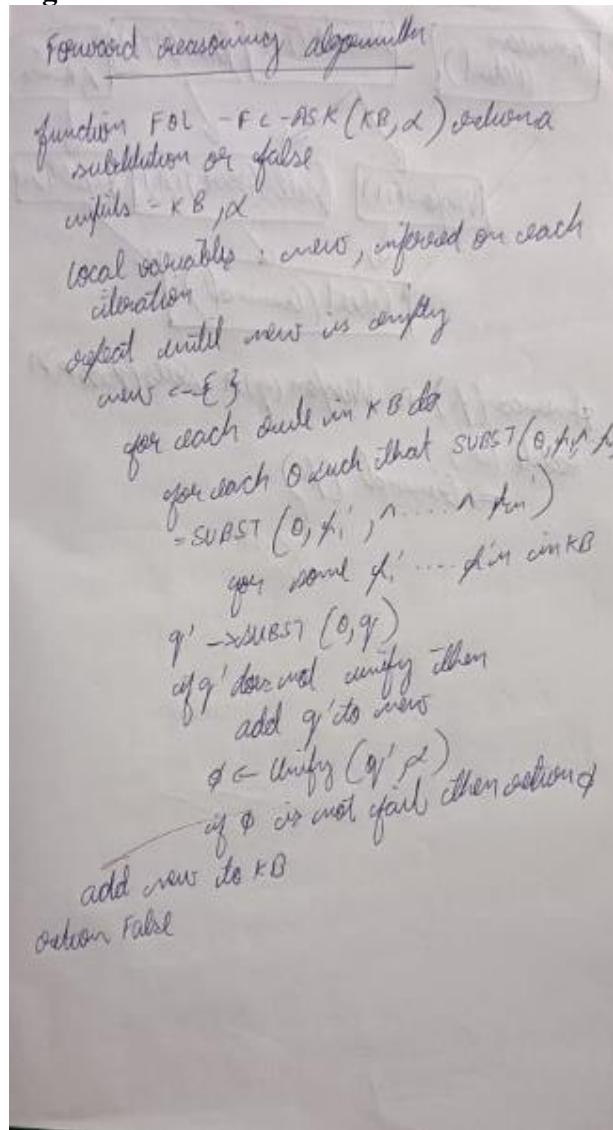
Output:

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:



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"
        for i in range(1, len(x)):
            theta = unify(x[i], y[i], theta)
            if theta == "FAIL":
                return "FAIL"
        return theta
    else:
        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(" → [] (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:

The image contains handwritten notes on the Alpha-Beta Pruning algorithm. At the top left, there is a diagram of a game tree with three levels. The root node is labeled 'Max' and has two children, each labeled 'Min'. Each 'Min' node has two children, each labeled 'Max'. This structure represents a game between two players, with alternating levels of maximization and minimization.

Lab 9

Advanced Search

Implement Alpha-Beta Pruning

A modified variant of common method as $\alpha-\beta$ pruning. It's a way for improving the common algorithm.

Algorithm

function $\alpha-\beta$ search(state) returns an action
 $v \leftarrow \text{Min-Value}(\text{state}, -\infty, +\infty)$
 foreach the action in $\text{ACTIONS}(\text{state})$ with value v
 function $\text{MAX-Value}(\text{state } \alpha, \beta)$ returns a utility value
 utility value of $\text{Terminal-Test}(\text{state})$ other action
 $v \leftarrow -\infty$
 for each a in $\text{actions}(\text{state})$ do
 $v \leftarrow \max(v, \text{min-value}(\text{result}(a), \alpha, \beta))$
 if $v \geq \beta$ then action v
 $\alpha \leftarrow \max(\alpha, v)$
 return v

function $\text{Min-Value}(\text{state } \alpha, \beta)$ returns a utility value
 utility value of $\text{Terminal-Test}(\text{state})$ other action
 $v \leftarrow +\infty$
 for each a in $\text{actions}(\text{state})$ do
 $v \leftarrow \min(v, \text{max-value}(\text{result}(a), \alpha, \beta))$
 if $v \leq \alpha$ then action v
 $\beta \leftarrow \min(\beta, v)$
 return v

D/P Initial : $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 0 & 8 \end{bmatrix}$

Max Player

Best value for $\alpha-\beta$: $9 // 11$

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"))
    else:
        best = math.inf
        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, "Left" if i == 0 else "Right"))

    return best, best_path
```

```

        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')]

```