

**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**  
“JnanaSangama”, Belgaum -590014, Karnataka.



**LAB REPORT**  
on

# **Artificial Intelligence (23CS5PCAIN)**

*Submitted by*

**Pratham Ganapathy (1BM22CS206)**

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

**Prof.Swati Sridharan**  
Assistant Professor  
Department of Computer Science and Engineering



**B.M.S. COLLEGE OF ENGINEERING**  
(Autonomous Institution under VTU)  
**BENGALURU-560019**  
**Sep-2024 to Jan-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 Pratham Ganapathy (**1BM22CS206**), who is bonafide student of **B.M.S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Radhika A D Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE
--	--

# Index

<b>Sl. No.</b>	<b>Date</b>	<b>Experiment Title</b>	<b>Page No.</b>
1	30-9-2024	Implement Tic – Tac – Toe Game Implement vacuum cleaner agent	4-12
2	7-10-2024	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	13-16
3	14-10-2024	Implement A* search algorithm	17-21
4	21-10-2024	Implement Hill Climbing search algorithm to solve N-Queens problem	22-34
5	28-10-2024	Simulated Annealing to Solve 8-Queens problem	35-40
6	11-11-2024	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	41-47
7	2-12-2024	Implement unification in first order logic	48-54
8	2-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	55-58
9	16-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	59-62
10	16-12-2024	Implement Alpha-Beta Pruning.	63-66

Github Link: [GitHub - PrathamGanapathy/AI-CS206](#)

# Program 1

Implement Tic - Tac - Toe Game

Implement vacuum cleaner agent

Tic-Tac-Toe

Algorithm:

24/9/24

PAGE NO :  
DATE :

Lab 1 : Tic - Tac - Toe bot

→ Pseudocode

Function to determine winner of a given board

def winner(board):

    d1 = set([board[i][i] for i in range(3)])  
    d2 = set([board[2-i][i] for i in range(3)])  
    rows = [set([board[i][j] for j in range(3)])  
           for i in range(3)]  
    cols = [set([board[j][i] for i in range(3)])  
           for i in range(3)]  
    if d1 == {'x'} or d2 == {'x'} or  
        {'x'} in rows or {'x'} in cols:  
        winner = 'x'  
  
    elif d1 == {'o'} or d2 == {'o'} or  
        {'o'} in rows or {'o'} in cols:  
        winner = 'o'  
  
    else:  
        return None

- To determine next move of bot.

def checkMove(board, bot):

    for move in possibleMoves(board): (board, player)

        boardCopy = applyMove(board)

        if winner(boardCopy) == bot:

            return move

        elif winner(boardCopy) == player:

            return

    else:

        return checkMove(boardCopy, bot)

if terminal(board):

    return None

- Initial Condition : An Empty  $3 \times 3$  array with all cells filled with None

- Final Condition : A final move (integer b/ $b \in 0-8$ ) that the board can play on the marking board

- Utility Function

```
def applyMove(board, player, move):
    boardCopy = board.deepcopy()
    boardCopy[move // 3][move % 3] = player
    return boardCopy
```

```
def terminalbd(board):
    if winner(board) is not None:
        return True
    elif winner(board) is None:
        for i in board:
            if None in i:
                return False
        return True
```

```
def possibleRow(board):
    for i in range(9):
        if board[i // 3][i % 3] == None:
            ans.append(i)
    return ans
```

- Initialization Code:

board - [ (None for \_ in range(3)) for \_ in range(3) ]

player = 'x'

bot = 'o'

current = 0

if current % 2 == 0:

board = applyMove(int(input("Move: ")))

current += 1

else:

board = applyMove(checkMove(board, bot))

current += 1

if winner(board) != None:

print(f"\{winner(board)} wins!")

else:

continue

O/p:

Enter

None None None Enter no. (0-8): 2

None None None X N O

None None None N N W

N N N

X N O Enter no. (0-8): 4

X N N

X N O

N N N

X O N

X wins

Code:

```
def check_win(board, r, c):
    if board[r - 1][c - 1] == 'X':
        ch = "O"
    else:
        ch = "X"
    if ch not in board[r - 1] and '-' not in board[r - 1]:
        return True
    elif ch not in (board[0][c - 1], board[1][c - 1], board[2][c - 1]) and '-' not in (board[0][c - 1], board[1][c - 1], board[2][c - 1]):
        return True
    elif ch not in (board[0][0], board[1][1], board[2][2]) and '-' not in (board[0][0], board[1][1], board[2][2]):
        return True
    elif ch not in (board[0][2], board[1][1], board[2][0]) and '-' not in (board[0][2], board[1][1], board[2][0]):
        return True
    return False

def displayb(board):
    print(board[0])
    print(board[1])
    print(board[2])

board=[['-','-','-'],['-','-','-'],['-','-','-']]
displayb(board)
xo=1
flag=0
while '-' in board[0] or '-' in board[1] or '-' in board[2]:

    if xo==1:
        print("enter position to place X:")
        x=int(input())
        y=int(input())
        if(x>3 or y>3):
            print("invalid position")
            continue
        if(board[x-1][y-1]=='-'):
            board[x-1][y-1]='X'
            xo=0
            displayb(board)
        else:
            print("invalid position")
            continue
        if(check_win(board,x,y)):
```

```

print("X wins")
flag=1
break
else :
    print("enter position to place O:")
    x=int(input())
    y=int(input())
    if(x>3 or y>3):
        print("invalid position")
        continue
    if(board[x-1][y-1]=='-'):
        board[x-1][y-1]='O'
        xo=1
        displayb(board)
    else:
        print("invalid position")
        continue
    if(check_win(board,x,y)):
        print("O wins")
        flag=1
        break
if flag==0:
    print("Draw")
print("Game Over")

```

```

[ "-", "-", "-"]
[ "-", "-", "-"]
[ "-", "-", "-"]
enter position to place X:
1
1
[ "x", "-", "-"]
[ "-", "-", "-"]
[ "-", "-", "-"]
enter position to place O:
1
2
[ "x", "o", "-"]
[ "-", "-", "-"]
[ "-", "-", "-"]
enter position to place X:
2
1
[ "x", "o", "-"]
[ "x", "-", "-"]
[ "-", "-", "-"]
enter position to place O:
2
2
[ "x", "o", "-"]
[ "x", "o", "-"]
[ "-", "-", "-"]
enter position to place X:
3
1
[ "x", "o", "-"]
[ "x", "o", "-"]
[ "x", "-", "-"]
X wins
Game Over

```

```
[', ', ', ']
[', ', ', ']
[', ', ', ']
enter position to place X:
1
1
['X', ' ', '']
[', ', ' ', '']
[', ', ' ', '']
enter position to place O:
2
2
['X', ' ', '']
[', 'O', ' ', '']
[', ', ' ', '']
enter position to place X:
3
3
['X', ' ', '']
[', 'O', ' ', '']
[', ', 'X', '']
enter position to place O:
1
2
['X', 'O', ' ']
[', 'O', ' ']
[', ', 'X']
enter position to place X:
3
2
['X', 'O', ' ']
[', 'O', ' ']
[', ', 'X']
enter position to place O:
3
1
['X', 'O', ' ']
[', 'O', ' ']
['O', 'X', 'X']
enter position to place X:
2
1
['X', 'O', ' ']
[', 'O', ' ']
['O', 'X', 'X']
enter position to place O:
2
3
['X', 'O', ' ']
['X', 'O', 'O']
['O', 'X', 'X']
enter position to place X:
1
3
['X', 'O', 'X']
['X', 'O', 'O']
['O', 'X', 'X']
Draw
Game Over
```

## Vacuum Cleaner

1/10/24

PAGE NO:  
DATE:

### Lab 2: Vacuum Cleaner

#### Step - 1:

Create 2 rooms using classes

Class Room:

```
def __init__(self, a):  
    self.state = a;  
def suck(self):  
    self.state = "clean";
```

#### Step 2:

Instantiate the class and take user input

```
a = int(input("Room A state"))  
b = int(input("Room B state"))
```

room\_list[]

```
room_list.append(room(a))  
room_list.append(room(b))
```

#### Step 3:

Percept Sequence:

```
for i in room_list:  
    if (i.state == "dirty"):  
        i.suck()
```

Code :

class room:

```
def __init__(self, a):
    self.state = a
```

```
def suck(self):
```

```
    self.state = "clean"
```

$n = 2$

roomList = []

for i in range(n):

```
a = int(input(f"Enter room {i+1} state:"))
```

```
roomList.append(room(a))
```

print("Before cleaning")

print("Room\tState")

for i in range(len(roomList)):

```
print(f"\t{i+1}\t{roomList[i].state}")
```

for i in roomList:

```
if (i.state == "dirty"):
```

```
i.suck()
```

print("\n")

print("After cleaning")

print("Room\tState")

for i in range(len(roomList)):

```
print(f"\t{i+1}\t{roomList[i].state}")
```

P.T.O →

Output :

Enter room 1 state : clean  
Enter room 2 state : dirty

Before cleaning :

Room	State
1	clean
2	dirty

After cleaning :

Room	State
1	clean
2	clean

Stop  
1/10/24

Output for 4 rooms :

Enter room 1 state : dirty  
Enter room 2 state : clean  
Enter room 3 state : dirty  
Enter room 4 state : dirty

Before cleaning :

Room	State
1	dirty
2	clean
3	dirty
4	dirty

Apt. cleaning

Room	State
1	clean
2	clean
3	clean
4	clean

Done  
10/21

Algorithm:

Code:

```
count = 0
def rec(state, loc):
    global count
    if state['A'] == 0 and state['B'] == 0:
        print("Turning vacuum off")
        return

    if state[loc] == 1:
        state[loc] = 0
        count += 1
        print(f"Cleaned {loc}.")
        next_loc = 'B' if loc == 'A' else 'A'
        state[loc] = int(input(f"Is {loc} clean now? (0 if clean, 1 if dirty): "))
        if(state[next_loc]!=1):
            state[next_loc]=int(input(f"Is {next_loc} dirty? (0 if clean, 1 if dirty): "))
        if(state[loc]==1):

rec(state,loc)else:
    next_loc = 'B' if loc == 'A' else 'A'
    dire="left" if loc=="B" else "right"
    print(loc,"is clean")
    print(f"Moving vacuum {dire}")
    if state[next_loc] == 1:
        rec(state, next_loc)

state = {}
state['A'] = int(input("Enter state of A (0 for clean, 1 for dirty): "))
state['B'] = int(input("Enter state of B (0 for clean, 1 for dirty): "))
loc = input("Enter location (A or B): ")
rec(state, loc)
print("Cost:",count)
print(state)
```

```
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
Turning vacuum off
Cost: 0
{'A': 0, 'B': 0}
```

```
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}
```

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

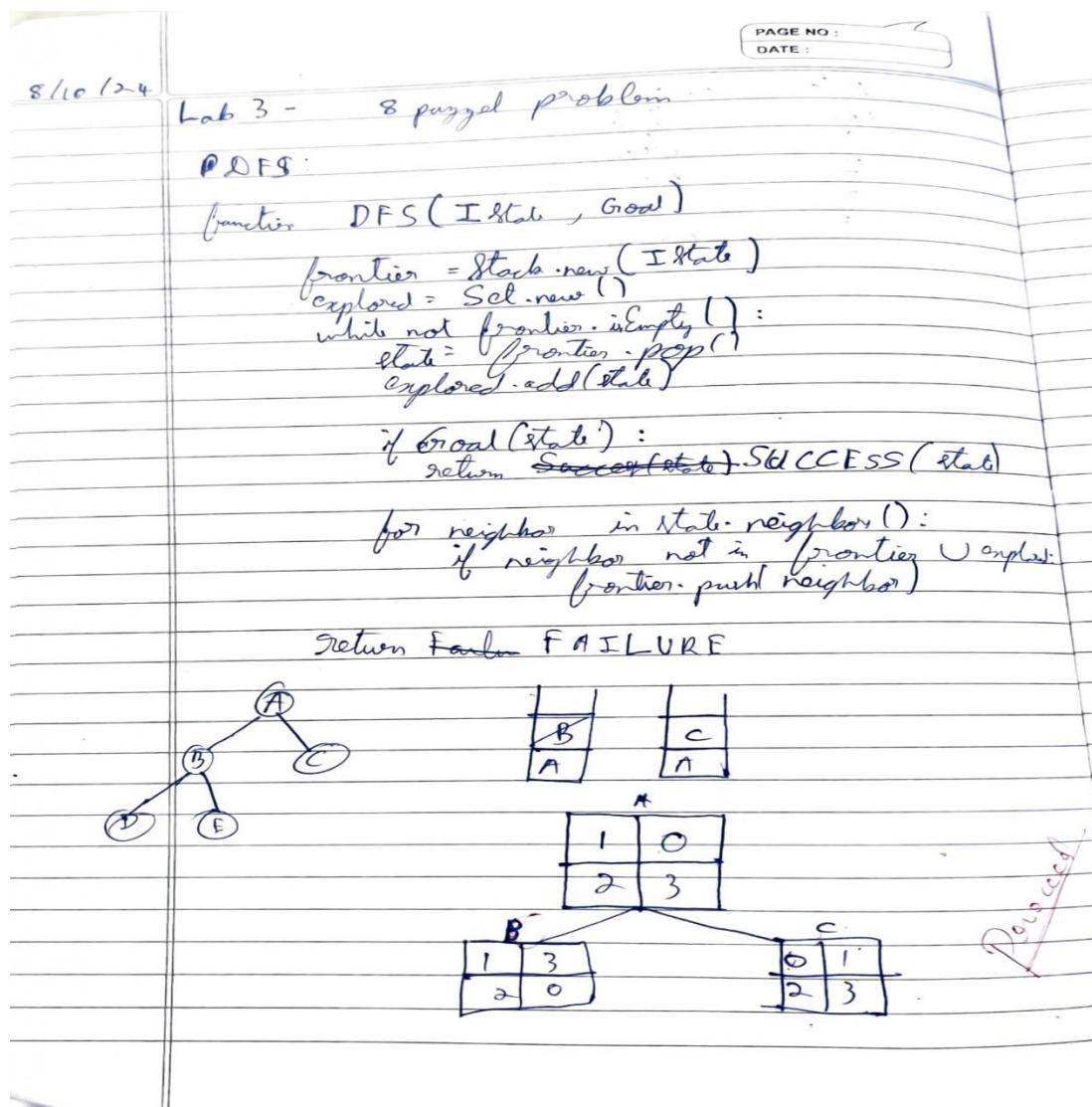
```
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): A
Cleaned A.
Is A clean now? (0 if clean, 1 if dirty): 0
A is clean
Moving vacuum right
Cleaned B.
Is B clean now? (0 if clean, 1 if dirty): 0
Is A dirty? (0 if clean, 1 if dirty):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

## Program 2

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

8 puzzle using DFS

Algorithm:



Code

```
from collections import deque
```

# Function to check if state is solvable

```
def is_solvable(state):
```

```
    inv_count = 0
```

```
    flat_state = [num for row in state for num in row if num != 0]
    for i in range(len(flat_state) - 1):
```

```
        for j in range(i + 1, len(flat_state)):
```

```
            if flat_state[i] > flat_state[j]:
```

```
                inv_count += 1
```

```
return inv_count % 2 == 0
```

# function to check if the goal state is reached

```
def is_goal(state, goal_state):
```

```
    return state == goal_state
```

# function to get the position of the blank tile (0)

```
def find_blank(state):
```

```
    for i in range(len(state)):
```

```
        for j in range(len(state[i])):
```

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

```
                return i, j
```

# function to generate possible moves from the current state

```
def get_neighbours(state):
```

```
    neighbours = []
```

```
x, y = find_blank(state)
```

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

```
for dx, dy in moves:
```

```
    nx, ny = x + dx, y + dy
```

```

if  $0 \leq nx < \text{len}(\text{state})$  and  $0 \leq ny < \text{len}(\text{state}[0])$   

    new_state = [row[:]] for row in state  

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

    new_state[x][y]  

    neighbor.append(new_state)  

return neighbor

```

```

# DFS to solve the 8-puzzle problem
def dfs(start_state, goal_state):
    if not is_solvable(start_state):
        return "The puzzle is unsolvable."

```

```

state = [(start_state, [])] + to hold DFS
visited = set()

```

```

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

```

```

    if is_goal(current_state, goal_state):
        return path

```

```

    visited.add(tuple(map(tuple, current_state)))

```

```

    for neighbor in get_neighbors(current_state):
        if tuple(map(tuple, neighbor)) not in visited:

```

```

            stack.append((neighbor, path + [neighbor]))

```

```

return "No Solution Found"

```

Code:

```
def dfs(initial_board, zero_pos):
    stack = [(initial_board, zero_pos, [])]
    visited = set()

    while stack:
        current_board, zero_pos, moves = stack.pop()

        if is_goal(current_board):
            return moves, len(moves) # Return moves and their count

        visited.add(tuple(current_board))

        for neighbor_board, neighbor_pos in get_neighbors(current_board, zero_pos):
            if tuple(neighbor_board) not in visited:
                stack.append((neighbor_board, neighbor_pos, moves + [neighbor_board]))

    return None, 0 # No solution found, return count as 0

# Initial state of the puzzle
initial_board = [1, 2, 3, 0, 4, 6, 7, 5, 8]
zero_position = (1, 0) # Position of the empty tile (0)

# Solve the puzzle using DFS
solution, move_count = dfs(initial_board, zero_position)

if solution:
    print("Solution found with moves ({ } moves):".format(move_count))
    for move in solution:
        print_board(move)
        print() # Print an empty line between moves
else:
```

```
print("No solution found."
```

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

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

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

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

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

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

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

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

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

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

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

Implement Iterative deepening search algorithm

## Algorithm

Code:

```
from collections import deque

class PuzzleState:
    def __init__(self, board, zero_pos, moves=0, previous=None):
        self.board = board
        self.zero_pos = zero_pos # Position of the zero tile
        self.moves = moves      # Number of moves taken to reach this state
        self.previous = previous # For tracking the path

    def is_goal(self, goal_state):
        return self.board == goal_state

    def get_possible_moves(self):
        moves = []
        x, y = self.zero_pos
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)] # Up, Down, Left, Right
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = [row[:] for row in self.board]

# Swap the zero tile with the adjacent tile
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
                new_board[x][y]
                moves.append((new_board, (new_x, new_y)))
        return moves

    def ids(initial_state, goal_state, max_depth):
        for depth in range(max_depth):
            visited = set()
            result = dls(initial_state, goal_state, depth, visited)
            if result:
                return result
        return None

    def dls(state, goal_state, depth, visited):
        if state.is_goal(goal_state):
```

```

        return state
    if depth == 0:
        return None

    visited.add(tuple(map(tuple, state.board))) # Mark this state as visited
    for new_board, new_zero_pos in state.get_possible_moves():
        new_state = PuzzleState(new_board, new_zero_pos, state.moves + 1, state)
        if tuple(map(tuple, new_board)) not in visited:
            result = dls(new_state, goal_state, depth - 1, visited)
            if result:
                return result
    visited.remove(tuple(map(tuple, state.board))) # Unmark this state
    return None

def print_solution(solution):
    path = []
    while solution:
        path.append(solution.board)
        solution = solution.previous
    for board in reversed(path):
        for row in board:
            print(row)
        print()

# Define the initial state and goal state
initial_state = PuzzleState(
    board=[[1, 2, 3],
           [4, 0, 5],
           [7, 8, 6]],
    zero_pos=(1, 1)
)

```

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

# Perform Iterative Deepening Search
max_depth = 20 # You can adjust this value
solution = ids(initial_state, goal_state, max_depth)
```

```
if solution:
    print("Solution found:")
    print_solution(solution)
else:
    print("No solution found.")
```

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

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

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

## Program 3

Implement A\* search algorithm

Algorithm:

15/10/24 Lab 4 PAGE NO : DATE :

IDF

```
graph TD; Y((Y)) --- P((P)); Y --- X((X)); P --- R((R)); P --- S((S)); X --- F((F)); X --- G((G)); R --- B((B)); R --- C((C)); S --- D((D)); S --- E((E)); R --- F((F))
```

Iteration 1 (depth = 0) : Y  
Iteration 2 (depth = 1) : Y → P → X  
Iteration 3 (depth = 2) : Y → P → R → S → X → F

Algorithm :-

```
adj_matrix = []
visited = []
def dfs(root, depth, curdepth):
    if (curdepth >= depth):
        return None
    if (root == target):
        return root
    else:
        for i in adj_matrix[root]:
            if adj_matrix[i] and not visited[i]:
                visited[i] = 1
                return dfs(i, depth, curdepth + 1)
```

A\* :

8	1	0
2	4	3
7	6	5

$$[4+0=4]$$

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

$$\begin{cases} 4 \times 1 = 4 \\ 4 \times 2 = 8 \end{cases}$$

8	1	3
2	4	0
7	6	5

[1+2=3]	[3+4=7]	[3+2=5]
0 8 1	8 1 0	8 4 1
2 4 3	2 4 3	2 0 3

2	8	1
0	4	3
7	6	5

$$[0+1=1]$$

20  
10/24

Code:

Misplaced Tiles

```
def mistil(state, goal):
```

```
    count = 0
```

```
    for i in range(3):
```

```
        for j in range(3):
```

```
            if state[i][j] != goal[i][j]:
```

```
                count += 1
```

```
    return count
```

```
def findmin(open_list, goal):
```

```
    minv = float('inf')
```

```
    best_state = None
```

```
    for state in open_list:
```

```
        h = mistil(state['state'], goal)
```

```
        f = state['g'] + h
```

```
        if f < minv:
```

```
            minv = f
```

```
            best_state = state
```

```
    open_list.remove(best_state)
```

```
    return best_state
```

```
def operation(state):
```

```
    next_states = []
```

```
    blank_pos = find_blank_position(state['state'])
```

```
    for move in ['up', 'down', 'left', 'right']:
```

```
        new_state = apply_move(state['state'], blank_pos, move)
```

```
        if new_state:
```

```
            next_states.append({
```

```
                'state': new_state,
```

```
                'parent': state,
```

```
                'move': move,
```

```
                'g': state['g'] + 1
```

```
            })
```

```
    return next_states
```

```
def find_blank_position(state):
```

```
    for i in range(3):
```

```
        for j in range(3):
```

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

```
                return i, j
```

```
    return None
```

```
def apply_move(state, blank_pos, move):
```

```
    i, j = blank_pos
```

```
    new_state = [row[:] for row in state]
```

```
    if move == 'up' and i > 0:
```

```

        new_state[i][j], new_state[i - 1][j] = new_state[i - 1][j], new_state[i][j]
    elif move == 'down' and i < 2:
        new_state[i][j], new_state[i + 1][j] = new_state[i + 1][j], new_state[i][j]
    elif move == 'left' and j > 0:
        new_state[i][j], new_state[i][j - 1] = new_state[i][j - 1], new_state[i][j]
    elif move == 'right' and j < 2:
        new_state[i][j], new_state[i][j + 1] = new_state[i][j + 1], new_state[i][j]
    else:
        return None
    return new_state

def print_state(state):
    for row in state:
        print(' '.join(map(str, row)))

initial_state = [[2,8,3], [1,6,4], [7,0,5]]
goal_state = [[1,2,3], [8,0,4], [7,6,5]]
open_list = [{ 'state': initial_state, 'parent': None, 'move': None, 'g': 0 }]
visited_states = []

while open_list:
    best_state = findmin(open_list, goal_state)
    print("Current state:")
    print_state(best_state['state'])
    h = mistil(best_state['state'], goal_state)
    f = best_state['g'] + h
    print(f"g(n): {best_state['g']}, h(n): {h}, f(n): {f}")
    if best_state['move'] is not None:
        print(f"Move: {best_state['move']}")
    print()
    if mistil(best_state['state'], goal_state) == 0:
        goal_state_reached = best_state
        break
    visited_states.append(best_state['state'])
    next_states = operation(best_state)
    for state in next_states:
        if state['state'] not in visited_states:
            open_list.append(state)

moves = []
while goal_state_reached['move'] is not None:
    moves.append(goal_state_reached['move'])
    goal_state_reached = goal_state_reached['parent']
moves.reverse()

print("\nMoves to reach the goal state:", moves)
print("\nGoal state reached:")

```

```
print_state(goal_state)
```

```
Current state:  
2 8 3  
1 6 4  
7 0 5  
g(n): 0, h(n): 5, f(n): 5  
  
Current state:  
2 8 3  
1 0 4  
7 6 5  
g(n): 1, h(n): 3, f(n): 4  
Move: up  
  
Current state:  
2 0 3  
1 8 4  
7 6 5  
g(n): 2, h(n): 4, f(n): 6  
Move: up  
  
Current state:  
2 8 3  
0 1 4  
7 6 5  
g(n): 2, h(n): 4, f(n): 6  
Move: left  
  
Current state:  
0 2 3  
1 8 4  
7 6 5  
g(n): 3, h(n): 3, f(n): 6  
Move: left  
  
Current state:  
1 2 3  
0 8 4  
7 6 5  
g(n): 4, h(n): 2, f(n): 6  
Move: down
```

```
Current state:  
1 2 3  
8 0 4  
7 6 5  
g(n): 5, h(n): 0, f(n): 5  
Move: right
```

```
Moves to reach the goal state: ['up', 'up', 'left', 'down', 'right']
```

```
Goal state reached:  
1 2 3  
8 0 4  
7 6 5
```

```

Manhattan Distance
def manhattan_distance(state, goal):
    distance = 0
    for i in range(3):
        for j in range(3):
            tile = state[i][j]
            if tile != 0: # Ignore the blank space (0)
                # Find the position of the tile in the goal state
                for r in range(3):
                    for c in range(3):
                        if goal[r][c] == tile:
                            target_row, target_col = r, c
                            break
                # Add the Manhattan distance (absolute difference in rows and columns)
                distance += abs(target_row - i) + abs(target_col - j)
    return distance

def findmin(open_list, goal):
    minv = float('inf')
    best_state = None
    for state in open_list:
        h = manhattan_distance(state['state'], goal) # Use Manhattan distance here
        f = state['g'] + h
        if f < minv:
            minv = f
            best_state = state
    open_list.remove(best_state)
    return best_state

def operation(state):
    next_states = []
    blank_pos = find_blank_position(state['state'])
    for move in ['up', 'down', 'left', 'right']:
        new_state = apply_move(state['state'], blank_pos, move)
        if new_state:
            next_states.append({
                'state': new_state,
                'parent': state,
                'move': move,
                'g': state['g'] + 1
            })
    return next_states

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

```

```

        return i, j
    return None

def apply_move(state, blank_pos, move):
    i, j = blank_pos
    new_state = [row[:] for row in state]
    if move == 'up' and i > 0:
        new_state[i][j], new_state[i - 1][j] = new_state[i - 1][j], new_state[i][j]
    elif move == 'down' and i < 2:
        new_state[i][j], new_state[i + 1][j] = new_state[i + 1][j], new_state[i][j]
    elif move == 'left' and j > 0:
        new_state[i][j], new_state[i][j - 1] = new_state[i][j - 1], new_state[i][j]
    elif move == 'right' and j < 2:
        new_state[i][j], new_state[i][j + 1] = new_state[i][j + 1], new_state[i][j]
    else:
        return None
    return new_state

def print_state(state):
    for row in state:
        print(''.join(map(str, row)))

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

# Open list and visited states
open_list = [{'state': initial_state, 'parent': None, 'move': None, 'g': 0}]
visited_states = []

while open_list:
    best_state = findmin(open_list, goal_state)

    print("Current state:")
    print_state(best_state['state'])

    h = manhattan_distance(best_state['state'], goal_state) # Using Manhattan distance here
    f = best_state['g'] + h
    print(f"g(n): {best_state['g']}, h(n): {h}, f(n): {f}")

    if best_state['move'] is not None:
        print(f"Move: {best_state['move']} ")
    print()
    if h == 0: # Goal is reached if h == 0
        goal_state_reached = best_state
        break

```

```

visited_states.append(best_state['state'])
next_states = operation(best_state)

for state in next_states:
    if state['state'] not in visited_states:
        open_list.append(state)

# Reconstruct the path of moves
moves = []
while goal_state_reached['move'] is not None:
    moves.append(goal_state_reached['move'])
    goal_state_reached = goal_state_reached['parent']
moves.reverse()

print("\nMoves to reach the goal state:", moves)
print("\nGoal state reached:")
print_state(goal_state)

```

```

Current state:
2 8 3
1 6 4
7 0 5
g(n): 0, h(n): 5, f(n): 5

Current state:
2 8 3
1 0 4
7 6 5
g(n): 1, h(n): 4, f(n): 5
Move: up

Current state:
2 0 3
1 8 4
7 6 5
g(n): 2, h(n): 3, f(n): 5
Move: up

Current state:
0 2 3
1 8 4
7 6 5
g(n): 3, h(n): 2, f(n): 5
Move: left

Current state:
1 2 3
0 8 4
7 6 5
g(n): 4, h(n): 1, f(n): 5
Move: down

```

```
Current state:  
1 2 3  
8 0 4  
7 6 5  
g(n): 5, h(n): 0, f(n): 5  
Move: right  
  
Moves to reach the goal state: ['up', 'up', 'left', 'down', 'right']  
  
Goal state reached:  
1 2 3  
8 0 4  
7 6 5
```

## Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

29/10/24

PAGE NO :  
DATE :

Lab 6 - 8-queens using A\* and Hill climbing algorithm

1. 8-queens using A\*:  
Taking  $f(x)$  as number of empty spaces and  $g(x)$  as number of queens left to place.

Initial Setup:  
Assuming an  $8 \times 8$  array of integers representing the chessboard. ~~1 indicates queen placed on square, 0 indicates not placed.~~

board =  $\left[ \left[ 0 \text{ for } i \text{ in range}(8) \right] \text{ for } j \text{ in range}(8) \right]$

Heuristic functions:

```
def emptySpaces(board):
    for i in range(8):
        for j in range(8):
            row_check = set(board[i])
            col_check = set([board[j][k] for k in range(8)])
            col_check = set([board[k][j] for k in range(8)])
            diag_index = [(k, k + j - 1) for k in range(8)]
            for i in diag_index:
                if min(i) < 0 or max(i) > 7:
                    diag_index.remove(i)
            diag_check = [board[i[0]][i[1]] for i in diag_index]
            diag_check = set(diag_check)
            diag_check = [board[i[0]][i[1]] for i in diag_index]
            diag_check.set(diag_check)
```

```

if len(row-check) == 1 and len(col-check) == 1
and len(dig-check) == 1:
    spaces += 1
return spaces

```

```

2 def no_of_Queens_left(board):
    count = 0
    for row in board:
        if 1 in row:
            count += 1
    return 8 - count

```

### Hill Climbing Algorithm:

Initial Setup:

Consider a mathematical expression which we have to attempt to maximise

```
def hill_climber(initial_sol=0, step=0.01, max_step=100)
```

```
current = initial_sol
```

```
best = current
```

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

```
    neighbor = [current + step, current - step]
```

```
    neighbor_best = min(neighbor, key=lambda x: cost(x))
```

```
    current = neighbor_best
```

```
    if cost(best) > cost(current):
```

```
        best = current
```

```
    else:
```

```
        break
```

```
result best
```

Initial Code:

initial\_val = random.uniform(-10, 10)

steps = random.choice(0.01, 0.001)

print("hill-climbing, (initial\_val, steps))

~~Don't do this~~

Output

Code:

import random

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

```

return conflicts

def hill_climbing(n):
    cost=0
    while True:
        # Initialize a random board
        current_board = list(range(n))
        random.shuffle(current_board)
        current_conflicts = calculate_conflicts(current_board)

    while True:
        # Generate neighbors by moving each queen to a different position
        found_better = False
        for i in range(n):
            for j in range(n):
                if j != current_board[i]: # Only consider different positions
                    neighbor_board = list(current_board)
                    neighbor_board[i] = j
                    neighbor_conflicts = calculate_conflicts(neighbor_board)
                    if neighbor_conflicts < current_conflicts:
                        print_board(current_board)
                        print(current_conflicts)
                        print_board(neighbor_board)
                        print(neighbor_conflicts)
                        current_board = neighbor_board
                        current_conflicts = neighbor_conflicts
                        cost+=1
                        found_better = True
                        break
            if found_better:
                break

        # If no better neighbor found, stop searching
        if not found_better:
            break

    # If a solution is found (zero conflicts), return the board
    if current_conflicts == 0:
        return current_board, current_conflicts, cost

def print_board(board):
    n = len(board)
    for i in range(n):
        row = ['.] * n
        row[board[i]] = 'Q' # Place a queen
        print(''.join(row))
    print()

```

```
print("====")
# Example Usage
n = 4
solution, conflicts, cost = hill_climbing(n)
print("Final Board Configuration:")
print_board(solution)
print("Number of Cost:", cost)
=====
Q . .
. . . Q
. . Q .
. Q . .

4
Q . .
Q . .
. . Q .
. Q . .

3
Q . .
Q . .
. . Q .
. Q . .

3
. . Q .
Q . .
. . Q .
. Q . .

2
. . Q .
Q . .
. . Q .
. Q . .

2
. . . Q
Q . .
. . Q .
. Q . .

1
Final Board Configuration:
. Q . .
. . . Q
Q . .
. . Q .
```

## Program 5

Simulated Annealing to Solve 8-Queens problem  
Algorithm:

Lab 5: Simulated Annealing algorithm

```
def annealing (init_sol, temp, cooling, final_temp):
    current ← init_sol
    now ← current
    while temp >= final_temp:
        new.get
        new = getNeighbor (current)
        delta E ← cost (new) - cost (current)
        if delta E < 0:
            current = new
        else if random () < e^(-delta E / temp):
            current ← new
```

Code:

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

def queens_max(position):
    # This function calculates the number of pairs of queens that are not attacking each other
    position = np.round(position).astype(int) # Round and convert to integers for queen positions
    n = len(position)
    queen_not_attacking = 0

    for i in range(n - 1):
        no_attack_on_j = 0
        for j in range(i + 1, n):
            # Check if queens are on the same row or on the same diagonal
            if position[i] != position[j] and abs(position[i] - position[j]) != (j - i):
                no_attack_on_j += 1
            if no_attack_on_j == n - 1 - i:
                queen_not_attacking += 1
        if queen_not_attacking == n - 1:
            queen_not_attacking += 1
    return -queen_not_attacking # Negative because we want to maximize this value

# Bounds for each queen's position (0 to 7 for an 8x8 chessboard)
bounds = [(0, 8) for _ in range(8)]

# Use dual_annealing for simulated annealing optimization
result = dual_annealing(queens_max, bounds)

# Display the results
best_position = np.round(result.x).astype(int)
best_objective = -result.fun # Flip sign to get the number of non-attacking queens

print('The best position found is:', best_position)
print('The number of queens that are not attacking each other is:', best_objective)
```

```
The best position found is: [0 8 5 2 6 3 7 4]
The number of queens that are not attacking each other is: 8
```

## Program 6

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

Algorithm:

PAGE NO :  
DATE :

Lab -7

Knowledge Base

1. Alice is the mother of Bob
2. Bob is the father of Charlie
3. A father is a parent
4. A mother is a parent
5. All parents have children
6. If someone is a parent, their children are siblings
7. Alice is married to David

Hypothesis:  
Charlie is sibling of Bob

Entailment:

Let  $R_3$ : Bob is parent.

$R_2, R_5 \not\in R_3$

Let  $R_9$ : Alice is parent

$R_1, R_5 \in R_9$

Let  $R_{10}$ : Bob has a child, Charlie

$R_9, R_5, R_2 \vdash R_{10}$

Let  $R_{11}$ : Bob is sibling to Charlie

$R_{10}, R_8, R_6 \vdash R_{11}$

~~Drop~~  $\therefore$  We can prove hypothesis are true

Code:

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

# Function to evaluate an expression
def evaluate_expression(a, b, c, expression):
    # Use eval() to evaluate the logical expression
    return eval(expression)

# Function to generate the truth table and evaluate a logical expression
def truth_table_and_evaluation(kb, query):
    # All possible combinations of truth values for a, b, and c
    truth_values = [True, False]
    combinations = list(itertools.product(truth_values, repeat=3))

    # Reverse the combinations to start from the bottom (False -> True)
    combinations.reverse()

    # Header for the full truth table
    print(f'{a}<5} {b}<5} {c}<5} {KB}<20}{'Query'}<20}'")

    # Evaluate the expressions for each combination
```

```

for combination in combinations:
    a, b, c = combination

    # Evaluate the knowledge base (KB) and query expressions
    kb_result = evaluate_expression(a, b, c, kb)
    query_result = evaluate_expression(a, b, c, query)

    # Replace True/False with string "True"/"False"
    kb_result_str = "True" if kb_result else "False"
    query_result_str = "True" if query_result else "False"

    # Convert boolean values of a, b, c to "True"/"False"
    a_str = "True" if a else "False"
    b_str = "True" if b else "False"
    c_str = "True" if c else "False"

    # Print the results for the knowledge base and the query
    print(f'{a_str:<5} {b_str:<5} {c_str:<5} {kb_result_str:<20} {query_result_str:<20}')

# Additional output for combinations where both KB and query are true
print("\nCombinations where both KB and Query are True:")
print(f'{a:<5} {b:<5} {c:<5} {KB:<20}{Query:<20}')


# Print only the rows where both KB and Query are True
for combination in combinations:
    a, b, c = combination

    # Evaluate the knowledge base (KB) and query expressions
    kb_result = evaluate_expression(a, b, c, kb)
    query_result = evaluate_expression(a, b, c, query)

    # If both KB and query are True, print the combination
    if kb_result and query_result:
        a_str = "True" if a else "False"
        b_str = "True" if b else "False"
        c_str = "True" if c else "False"
        kb_result_str = "True" if kb_result else "False"
        query_result_str = "True" if query_result else "False"
        print(f'{a_str:<5} {b_str:<5} {c_str:<5} {kb_result_str:<20} {query_result_str:<20}')


# Define the logical expressions as strings
kb = "(a or c) and (b or not c)" # Knowledge Base
query = "a or b" # Query to evaluate

# Generate the truth table and evaluate the knowledge base and query
truth_table_and_evaluation(kb, query)

```

a	b	c	KB	Query
False	False	False	False	False
False	False	True	False	False
False	True	False	False	True
False	True	True	True	True
True	False	False	True	True
True	False	True	False	True
True	True	False	True	True
True	True	True	True	True

Combinations where both KB and Query are True:

a	b	c	KB	Query
False	True	True	True	True
True	False	False	True	True
True	True	False	True	True
True	True	True	True	True

## Program 7

Implement unification in first order logic

Algorithm:

Code:

```
import re
```

```
def occurs_check(var, x):
    """Checks if var occurs in x (to prevent circular substitutions)."""
    if var == x:
        return True
    elif isinstance(x, list): # If x is a compound expression (like a function or predicate)
        return any(occurs_check(var, xi) for xi in x)
    return False

def unify_var(var, x, subst):
    """Handles unification of a variable with another term."""
    pass
```

```

if var in subst: # If var is already substituted
    return unify(subst[var], x, subst)
elif isinstance(x, (list, tuple)) and tuple(x) in subst: # Handle compound expressions
    return unify(var, subst[tuple(x)], subst)
elif occurs_check(var, x): # Check for circular references
    return "FAILURE"
else:
    # Add the substitution to the set (convert list to tuple for hashability)
    subst[var] = tuple(x) if isinstance(x, list) else x
return subst

def unify(x, y, subst=None):
    """
    Unifies two expressions x and y and returns the substitution set if they can be unified.
    Returns 'FAILURE' if unification is not possible.
    """
    if subst is None:
        subst = {} # Initialize an empty substitution set

    # Step 1: Handle cases where x or y is a variable or constant
    if x == y: # If x and y are identical
        return subst
    elif isinstance(x, str) and x.islower(): # If x is a variable
        return unify_var(x, y, subst)
    elif isinstance(y, str) and y.islower(): # If y is a variable
        return unify_var(y, x, subst)
    elif isinstance(x, list) and isinstance(y, list): # If x and y are compound expressions (lists)
        if len(x) != len(y): # Step 3: Different number of arguments
            return "FAILURE"

    # Step 2: Check if the predicate symbols (the first element) match
    if x[0] != y[0]: # If the predicates/functions are different
        return "FAILURE"

    # Step 5: Recursively unify each argument
    for xi, yi in zip(x[1:], y[1:]): # Skip the predicate (first element)
        subst = unify(xi, yi, subst)
        if subst == "FAILURE":
            return "FAILURE"
    return subst

else: # If x and y are different constants or non-unifiable structures
    return "FAILURE"

def unify_and_check(expr1, expr2):
    """
    Attempts to unify two expressions and returns a tuple:
    (is_unified: bool, substitutions: dict or None)
    """

```

```

"""
result = unify(expr1, expr2)
if result == "FAILURE":
    return False, None
return True, result

def display_result(expr1, expr2, is_unified, subst):
    print("Expression 1:", expr1)
    print("Expression 2:", expr2)
    if not is_unified:
        print("Result: Unification Failed")
    else:
        print("Result: Unification Successful")
        print("Substitutions:", {k: list(v) if isinstance(v, tuple) else v for k, v in subst.items()})

def parse_input(input_str):
    """Parses a string input into a structure that can be processed by the unification algorithm."""
    # Remove spaces and handle parentheses
    input_str = input_str.replace(" ", "")

    # Handle compound terms (like p(x, f(y)) -> ['p', 'x', ['f', 'y']])
    def parse_term(term):
        # Handle the compound term
        if '(' in term:
            match = re.match(r'([a-zA-Z0-9_]+)(.*)', term)
            if match:
                predicate = match.group(1)
                arguments_str = match.group(2)
                arguments = [parse_term(arg.strip()) for arg in arguments_str.split(',')]
                return [predicate] + arguments
        return term

    return parse_term(input_str)

# Main function to interact with the user
def main():
    while True:
        # Get the first and second terms from the user
        expr1_input = input("Enter the first expression (e.g., p(x, f(y))): ")
        expr2_input = input("Enter the second expression (e.g., p(a, f(z))): ")

        # Parse the input strings into the appropriate structures
        expr1 = parse_input(expr1_input)
        expr2 = parse_input(expr2_input)

        # Perform unification
        is_unified, result = unify_and_check(expr1, expr2)

```

```

# Display the results
display_result(expr1, expr2, is_unified, result)

# Ask the user if they want to run another test
another_test = input("Do you want to test another pair of expressions? (yes/no): ").strip().lower()
if another_test != 'yes':
    break

if __name__ == "__main__":
    main()

Enter the first expression (e.g., p(x, f(y))): p(b,x,f(g(z)))
Enter the second expression (e.g., p(a, f(z))): p(z,f(y),f(y))
Expression 1: ['p', '(b', 'x', ['f', '(g(z))']]]
Expression 2: ['p', '(z', ['f', '(y)', ['f', '(y)']]]
Result: Unification Successful
Substitutions: {'(b': '(z', 'x': ['f', '(y)', '(g(z))'): '(y)')}
Do you want to test another pair of expressions? (yes/no): yes
Enter the first expression (e.g., p(x, f(y))): p(x,h(y))
Enter the second expression (e.g., p(a, f(z))): p(a,f(z))
Expression 1: ['p', '(x', ['h', '(y)']]
Expression 2: ['p', '(a', ['f', '(z)']]
Result: Unification Failed
Do you want to test another pair of expressions? (yes/no): yes
Enter the first expression (e.g., p(x, f(y))): p(f(a),g(y))
Enter the second expression (e.g., p(a, f(z))): p(x,x)
Expression 1: ['p', '(f(a)', ['g', '(y)']]
Expression 2: ['p', '(x', 'x')]
Result: Unification Successful
Substitutions: {'(f(a)': '(x', 'x)': ['g', '(y)']}
Do you want to test another pair of expressions? (yes/no): no

```

## Program 8

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

Algorithm:

PAGE NO :  
DATE :

Lab - 8

Problem:

- 1 All birds can fly :  $\forall x (\text{Bird}(x) \rightarrow \text{CanFly}(x))$
- 2 All sparrow are birds :  $\forall x (\text{Sparrow}(x) \rightarrow \text{Bird}(x))$
- 3 Tweety is a sparrow :  $\text{Sparrow}(\text{Tweety})$

Hypothesis

Tweety can fly  
 $\text{CanFly}(\text{Tweety})$  ( $\text{Tweety can fly}$ )

Entailment:

- 1  $\forall x (\text{Bird}(x) \rightarrow \text{CanFly}(x))$   
 $x$ : bird
- 2  $B(\text{Tweety})$
- 3 ~~from 1 :~~  
 $B(\text{Tweety}) \rightarrow \text{CanFly}(\text{Tweety})$
- 4  ~~$\forall x (\text{Sparrow}(x) \rightarrow \text{Bird}(x))$~~
- 5  $\text{Sparrow}(\text{Tweety})$
- 6 ~~from 5 :~~  
 $\text{Sparrow}(\text{Tweety}) \rightarrow \text{Bird}(\text{Tweety})$
- 7 ~~from 6 & 3 :~~  $\text{CanFly}(\text{Tweety})$

~~Done~~  
19/11/24

Code:

import re

# Regular expression

predicates = {

"is-a-bird": "B",

"can-fly": "F",

"is-a-pigeon": "S"

}

# content

Content = {

"This": "T" }

def translate\_to\_fol(sentence):

sentence = sentence.replace("I", "I'm not")

if "is-a-bird" in sentence:

return translate\_is\_a\_bird(sentence)

if "can-fly" in sentence:

return translate\_can\_fly(sentence)

if "is-a-pigeon" in sentence:

return translate\_is\_a\_pigeon(sentence)

if "away" in sentence:

return translate\_away(sentence)

if "there exist" in sentence or "there is" in sentence:

return translate\_with(sentence)

if "not all" in sentence:

return translate\_not\_all(sentence)

if "if" in sentence and "then" in sentence:

return translate\_if\_then(sentence)

if "nobody" in sentence:

return translate\_nobody(sentence)

if "and" in sentence:

return translate\_and(sentence)

return "Translator not available for this sentence structure."

# helper function for specific cases

def translate\_is\_a\_bird(sentence):

match = re.match(r"([a-zA-Z]+) is-a-bird", sentence)

if match:

subject = match.group(1)

subject\_content = Content.get(subject, subject)

return f"B({subject\_content})"

else:

return "I need sentence structure."

def translate\_can\_fly(sentence):

match = re.match(r"([a-zA-Z]+) can-fly", sentence)

if match:

subject = match.group(1)

PAGE NO.:  
DATE:

```

subject const = constants.get(subject, subject)
return f" {subject} -const"

```

return "Invalid sentence structure."

```

def translate_girls(sentence):
    match = re.match(r"there exists ([a-zA-Z]+) who ([a-zA-Z]+)([a-zA-Z]+)", sentence)

```

if match occurs:

subject = match.group(1)

predicate = match.group(2)

object = match.group(3)

subject const = constants.get(subject, subject)

object const = constants.get(object, object)

return f" {exists} {predicate} ({subject}, {object})"

return "Invalid sentence structure."

# Function to handle "nobody" sentence

```

def translate_nobody(sentence):
    match = re.match(r"nobody is ([a-zA-Z]+) themself", sentence)

```

if match:

predicate = match.group(1)

return f" {exists} {predicate} ({x}, {x})"

return "Invalid sentence structure."

def main():

print("Enter a sentence like: ")

print("1. Tweety is a bird")

print("2. Every bird can fly.")

PAGE NO.:  
DATE:

```

print("3. Tweety is a sparrow")
print("4. There exists a bird who can fly")
print("Type 'exit' to quit")

```

while True:

sentence = input("Enter a sentence: ").strip()

if sentence.lower() == 'exit':

print("Goodbye!")

break

\* fol\_translator = translate\_to\_fol(sentence)

print("First-Order Logic Translation: ", fol\_translator)

if name == "\_\_main\_\_":

main()

## Lab-9 : First Order Logic Forward Chaining

Considering Statement

It per law, it is a crime for an American to sell weapons to hostile nation. Country A, an enemy of America, has some missiles, and all missiles were sold by Robert, who is American.

Prove Robert is a criminal

Deduced Statement:

$$\text{American}(x) \wedge \text{Weapon}(y) \wedge \text{sold}(z, y, x) \wedge \text{Hostile}(z) \\ \Rightarrow \text{Criminal}(x)$$

$\text{Enemy}(A, \text{America})$

$$\forall x \text{Missile}(x) \wedge \text{Own}(A, x) \\ \forall x \text{Missile}(x) \wedge \text{Own}(A, x) \Rightarrow \text{sold}(A, x, \text{Robert})$$

~~American~~  
 $\text{America}(\text{Robert})$

We concluding that missiles are weapon

$$\forall x \text{Missile}(x) \rightarrow \text{weapon}(x)$$

We can also deduce that an enemy of America is hostile

$$\forall x \text{Enemy}(x, \text{America}) \Rightarrow \text{Hostile}(x)$$

$\therefore$  First level of atomic statement

~~$\text{American}(\text{Robert}) \quad \text{Missile}(y) \quad \text{Own}(A, y) \quad \text{Enemy}(A, \text{Asia})$~~

Code:

```
# Define the knowledge base (KB) as a set of facts
KB = set()

# Premises based on the provided FOL problem
KB.add('American(Robert)')
KB.add('Enemy(America, A)')
KB.add('Missile(T1)')
KB.add('Owns(A, T1)")

# Define inference rules
def modus_ponens(fact1, fact2, conclusion):
    """ Apply modus ponens inference rule: if fact1 and fact2 are true, then conclude conclusion """
    if fact1 in KB and fact2 in KB:
        KB.add(conclusion)
        print(f"Inferred: {conclusion}")

def forward_chaining():
    """ Perform forward chaining to infer new facts until no more inferences can be made """
    # 1. Apply: Missile(x) → Weapon(x)
    if 'Missile(T1)' in KB:
        KB.add('Weapon(T1)')
        print(f"Inferred: Weapon(T1)")

    # 2. Apply: Sells(Robert, T1, A) from Owns(A, T1) and Weapon(T1)
    if 'Owns(A, T1)' in KB and 'Weapon(T1)' in KB:
        KB.add('Sells(Robert, T1, A)')
        print(f"Inferred: Sells(Robert, T1, A)")

    # 3. Apply: Hostile(A) from Enemy(A, America)
    if 'Enemy(America, A)' in KB:
        KB.add('Hostile(A)')
        print(f"Inferred: Hostile(A)")

    # 4. Now, check if the goal is reached (i.e., if 'Criminal(Robert)' can be inferred)
    if 'American(Robert)' in KB and 'Weapon(T1)' in KB and 'Sells(Robert, T1, A)' in KB and 'Hostile(A)' in KB:
        KB.add('Criminal(Robert)')
        print("Inferred: Criminal(Robert)")

    # Check if we've reached our goal
    if 'Criminal(Robert)' in KB:
        print("Robert is a criminal!")
    else:
        print("No more inferences can be made.")

# Run forward chaining to attempt to derive the conclusion
forward_chaining()
```

```
Inferred: Weapon(T1)
Inferred: Sells(Robert, T1, A)
Inferred: Hostile(A)
Inferred: Criminal(Robert)
Robert is a criminal!
```

## Program 9

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

Algorithm:

PAGE NO :  
DATE :

Lab - 9 : First Order Logic Forward Chaining

Considering Statement  
As per law, it is a crime for an American to sell weapons to hostile nation. Country A, an enemy of America, has some missiles, and all missiles were sold by Robert, who is American.

Prove Robert is a criminal

Deduced Statement :-  
 $\text{American}(x) \wedge \text{Weapon}(y) \wedge \text{sold}(z, y, x) \wedge \text{Hostile}(z) \Rightarrow \text{Criminal}(x)$

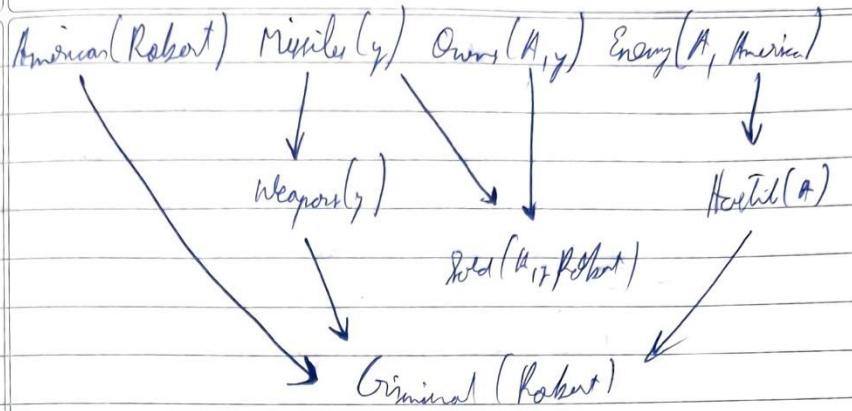
$\text{Enemy}(A, \text{America})$   
 $\neg \exists x \text{ Missile}(x) \wedge \text{Owns}(A, x)$   
 $\forall x \text{ Missile}(x) \wedge \text{Owns}(A, x) \Rightarrow \text{sold}(A, x, \text{Robert})$

~~American  
America (Robert)~~

We are deducing that missiles are weapons  
 $\forall x \text{ Missile}(x) \rightarrow \text{Weapon}(x)$

We can also deduce that an enemy of America is hostile  
 $\forall x \text{ Enemy}(x, \text{America}) \Rightarrow \text{Hostile}(x)$

$\therefore$  First level of atomic statement  
~~American(Robert) Missile(y) Owns(A, y) Enemy(A, America)~~



Algorithm:

Letting  $\text{KB}$  represent all current knowledge  
 $\alpha$  represents atomic query to be verified

while  $\text{new} \neq \{\}$ :

for role in  $\text{KB}$  do  
 $(p_1 \wedge p_2 \dots \wedge p_n \rightarrow q) \leftarrow \text{knowledge-variable}(r)$   
 $\theta \in \Theta$

$(\theta, p_1 \wedge p_2 \dots \wedge p_n) \rightarrow (\theta, q')$   
 for  $p_1, p_2, \dots \in \text{KB}$

$q' = \text{subst}(\theta, q)$

If  $q'$  does not unify some entries in  
 $\text{KB}$  or new

$\text{new} + q'$   
 $\phi = \text{unify}(q', \alpha)$

If not  $\phi$  return False  
 Else return  $\phi$

$\text{KB} + \text{new}$

return  $\phi$

### General Algorithm

- Consider all atomic sentence in KB
- Add sentence to KB s.t all can be inferred from layer from current level of sentence and requires atomic sentence already in KB
- If find sentence not in KB, return  
else else return yes

→ Resolution using FOL

Contains the following sentences:

1. food( $\eta$ )  $\rightarrow$  likes(John,  $\eta$ )
2. food(Apple)
3. food(vegetable)
4. eat( $y_1 z$ )  $\wedge$  killed( $y$ )  $\rightarrow$  food( $z$ )
5. eat( $x_1$ , Peanuts)
6. alive( $x_1$ )
7. killed( $x$ )  $\vee$  alive( $x$ )
8.  $\neg$  killed( $\eta$ )  $\vee$   $\neg$  alive( $\eta$ )

To prove : likes(John, Peanuts)

```

# Define the knowledge base (KB)
KB = {
    "food(Apple)": True,
    "food(vegetables)": True,
    "eats(Anil, Peanuts)": True,
    "alive(Anil)": True,
    "likes(John, X)": "food(X)", # Rule: John likes all food
    "food(X)": "eats(Y, X) and not killed(Y)", # Rule: Anything eaten and not killed is food
    "eats(Harry, X)": "eats(Anil, X)", # Rule: Harry eats what Anil eats
    "alive(X)": "not killed(X)", # Rule: Alive implies not killed
    "not killed(X)": "alive(X)", # Rule: Not killed implies alive
}

# Function to evaluate if a predicate is true based on the KB
def resolve(predicate):
    # If it's a direct fact in KB
    if predicate in KB and isinstance(KB[predicate], bool):
        return KB[predicate]

    # If it's a derived rule
    if predicate in KB:
        rule = KB[predicate]
        if " and " in rule: # Handle conjunction
            sub_preds = rule.split(" and ")
            return all(resolve(sub.strip()) for sub in sub_preds)
        elif " or " in rule: # Handle disjunction
            sub_preds = rule.split(" or ")
            return any(resolve(sub.strip()) for sub in sub_preds)
        elif "not " in rule: # Handle negation
            sub_pred = rule[4:] # Remove "not "
            return not resolve(sub_pred.strip())
        else: # Handle single predicate
            return resolve(rule.strip())

    # If the predicate is a specific query (e.g., likes(John, Peanuts))
    if "(" in predicate:
        func, args = predicate.split("(")
        args = args.strip(")").split(", ")
        if func == "food" and args[0] == "Peanuts":
            return resolve("eats(Anil, Peanuts)") and not resolve("killed(Anil)")
        if func == "likes" and args[0] == "John" and args[1] == "Peanuts":
            return resolve("food(Peanuts)")

    # Default to False if no rule or fact applies
    return False

```

```
# Query to prove: John likes Peanuts
query = "likes(John, Peanuts)"
result = resolve(query)

# Print the result
print(f'Does John like peanuts? {"Yes" if result else "No"}')
Does John like peanuts? Yes
```

## Program 10

Implement Alpha-Beta Pruning.

Algorithm:

PAGE NO :  
DATE :

Alpha - Beta Pruning :

```
def abprune (board, a - INT.MAX, b - INT.MAX)
    if row == N
        return board
    for col in range 0 to N :
        if safe (board, row, col) :
            board [row] [col] = 1
            result = abprune (board, result, b)
        if result > a :
            return result
    board [row] [col] = 0
    a = max (a, result)
    if a > b :
        break
    return NULL
```

Using the algorithm to solve we can get the result  
output list given as  
~~(0, 4, 7, 5, 2, 6, 1, 3)~~

From  
2nd Pgm

Code:

```
# Alpha-Beta Pruning Implementation
def alpha_beta_pruning(node, alpha, beta, maximizing_player):
    # Base case: If it's a leaf node, return its value (simulating evaluation of the node)
    if type(node) is int:
        return node

    # If not a leaf node, explore the children
    if maximizing_player:
        max_eval = -float('inf')
        for child in node: # Iterate over children of the maximizer node
            eval = alpha_beta_pruning(child, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval) # Maximize alpha
            if beta <= alpha: # Prune the branch
                break
        return max_eval
    else:
        min_eval = float('inf')
        for child in node: # Iterate over children of the minimizer node

            eval = alpha_beta_pruning(child, alpha, beta, True)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval) # Minimize beta
            if beta <= alpha: # Prune the branch
                break
        return min_eval

# Function to build the tree from a list of numbers
def build_tree(numbers):
    # We need to build a tree with alternating levels of maximizers and minimizers
    # Start from the leaf nodes and work up
    current_level = [[n] for n in numbers]

    while len(current_level) > 1:
        next_level = []
        for i in range(0, len(current_level), 2):
            if i + 1 < len(current_level):
                next_level.append(current_level[i] + current_level[i + 1]) # Combine two nodes
            else:
                next_level.append(current_level[i]) # Odd number of elements, just carry forward
        current_level = next_level

    return current_level[0] # Return the root node, which is a maximizer
```

```

# Main function to run alpha-beta pruning
def main():
    # Input: User provides a list of numbers
    numbers = list(map(int, input("Enter numbers for the game tree (space-separated): ").split()))

    # Build the tree with the given numbers
    tree = build_tree(numbers)

    # Parameters: Tree, initial alpha, beta, and the root node is a maximizing player
    alpha = -float('inf')
    beta = float('inf')
    maximizing_player = True # The root node is a maximizing player

    # Perform alpha-beta pruning and get the final result
    result = alpha_beta_pruning(tree, alpha, beta, maximizing_player)

    print("Final Result of Alpha-Beta Pruning:", result)

if __name__ == "__main__":
    main()

```

```

Enter numbers for the game tree (space-separated): 10 9 14 18 5 4 50 3
Final Result of Alpha-Beta Pruning: 50

```