VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

VIJAY J(1WA23CS040)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

Aug-2025 to Jan-2026

B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

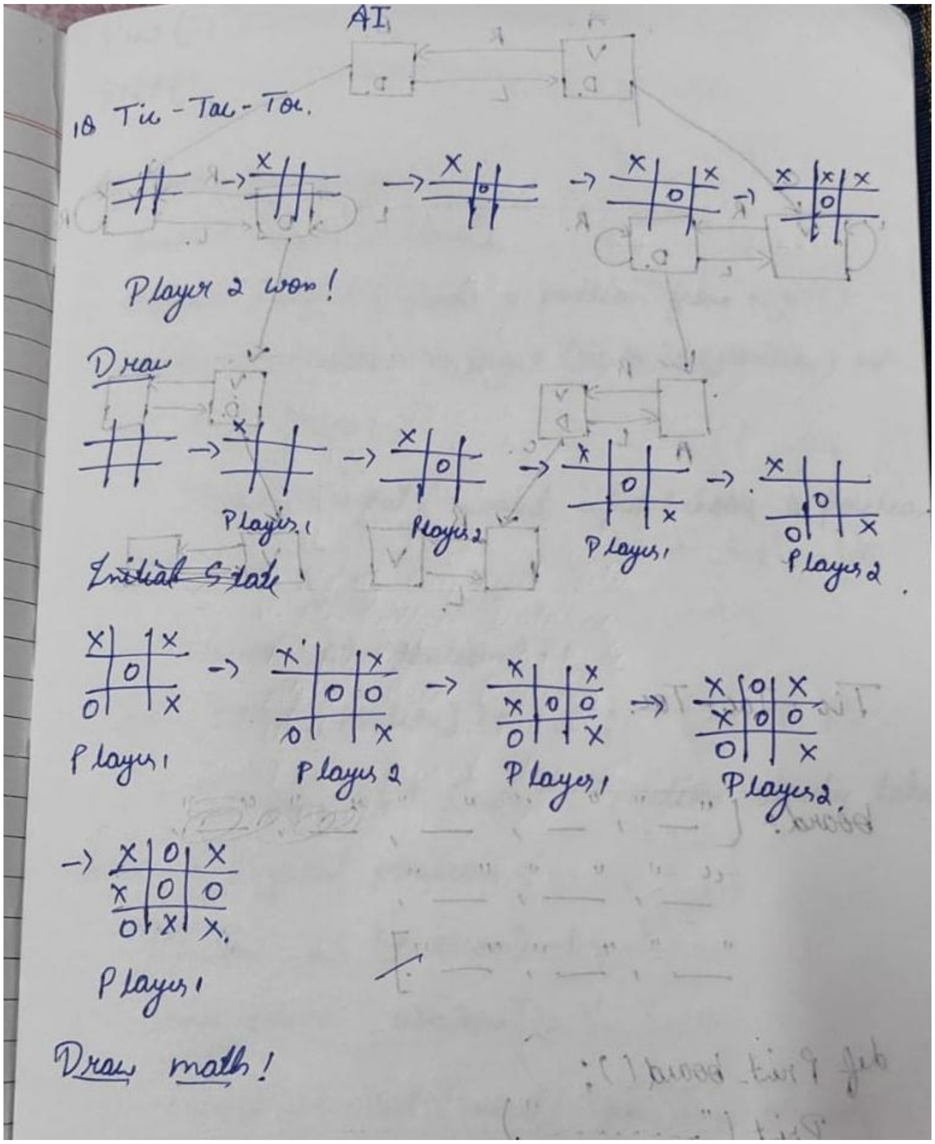
This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **VIJAY J (1WA23CS040),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of 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.

Lab faculty : SINDHOOR N Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

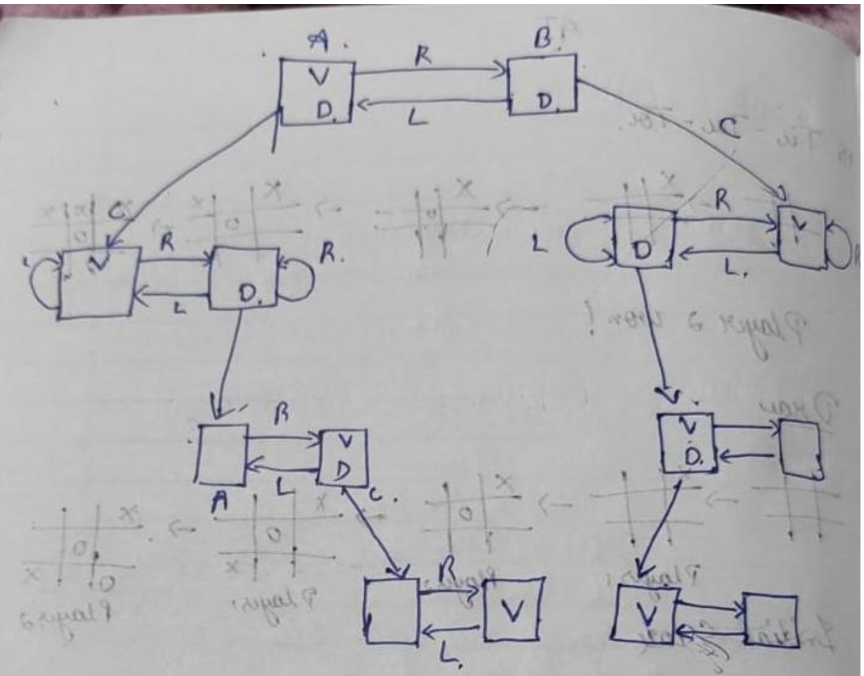
Index

Program 1 - Tic Tac toe and Vacum Cleaner

Algorithm



Code



Tic - Tac - Toe.

```
board = ["—", "—", "—",
         "—", "—", "—",
         "—", "—", "—"]

def Print_board():
    Print("---------")
    Print("|" + board[0] + "|" + board[1] + "|" + bo
-[2] + "|"]
    Print("-------")
    Print("|" + board[3] + "|" + board[4] + "|" + bo
- [5] + "|"]
```

Print ("|" + board [6] + "|" + board [7] + "|" + board [8] + "|

Print ("- - - - - - -")

def take_turn (player):

Print (f"player's turn")

Position = input ("choose a position from 1-9:")

while not position is digit (is or int (position) not

- in range (1,10):

Position = input (invalid input (choose a position from 1-9 : ").

Position = int (position) - 1

while board [position] != "_":

Position = int (input ("position already taken

- choose a different position :

Position = int (position) - 1

while board [position] != "_":

Position = int (input ("position already -

- taken . choose a diffrent position : ")) - 1

board [position] = player

Print_ board ()

```
def check_win(player):
    return ((board[0] == board[1] == board[2]
    == player) or
    (board[3] == board[4] == board[5] ==
    player) or
    (board[6] == board[7] == board[8] ==
    player)).

def check_tie():
    return "_" not in board.

def play_game():
    Print ("welcome to Tic Tac toe!").
    ~~Print (~~
    Print_board ().
    current_player = "X"
    game_over = False.
    while not game_over:
        take_turn (current player)
        if check_win(current_players)
            Print(f"current_player) wins!?").
            game_over = True.
```

```
elif check tie( ):
    Print ("its a tie!")
    game-over = True

else:
    current-player = "o" if current_player
    == "x" else "x"
Play-game().
```

2/9/25

Vacuum Cleaner.

```
rooms = ["D", "D"];
def fullclean (rooms):
    if ( rooms [0] == "ND" and rooms [1] == "ND"):
        return True
    else:
        return False.

def clean_room ():
    V=0;
    while not fullclean (rooms):
        if (rooms [V] == "D");
            room [V] = "ND")
            Print ("Room", "V", "Cleaned !")
        elif ( rooms [V] == "ND"):
            if (V==0):
                V+= 1
        else: V -= 1
    Print ("All rooms cleaned !")
clean_~~clean~~room().
```

O/P

Room 0 cleaned !
Room 1 Cleaned !
All rooms cleaned !

2/9/25

Code

```
import random
class TicTacToe:
```

```
def _init_(self):
    self.board = []
    def create_board(self):
        for i in range(3):
            row = []
            for j in range(3):
                row.append('-')
            self.board.append(row)
    def get_random_first_player(self):
        return random.randint(0, 1)
    def fix_spot(self, row, col, player):
        self.board[row][col] = player
    def is_player_win(self, player):
        win = None
        n = len(self.board)
        for i in range(n):
            win = True
            for j in range(n):
                if self.board[i][j] != player:
                    win = False
                    break
            if win:
                return win
        for i in range(n):
            win = True
            for j in range(n):
                if self.board[j][i] != player:
                    win = False
                    break
            if win:
                return win
        win = True
        for i in range(n):
            if self.board[i][i] != player:
                win = False
                break
        if win:
            return win
        win = True
```

```
for i in range(n):
                if self.board[i][n - 1 - i] != player:
                    win = False
                    break
            if win:
                return win
            return False
            for row in self.board:
                for item in row:
                    if item == '-':
                        return False
            return True
        def is_board_filled(self):
            for row in self.board:
                for item in row:
                    if item == '-':
                        return False
            return True
        def swap_player_turn(self, player):
            return 'X' if player == 'O' else 'O'
        def show_board(self):
            for row in self.board:
                for item in row:
                    print(item, end=" ")
                print()
        def start(self):
            self.create_board()
            player = 'X' if self.get_random_first_player() == 1 else 'O'
            while True:
                print(f"Player {player} turn")
                self.show_board()
                row, col = list(
                    map(int, input("Enter row and column numbers to fix spot: ").split()))
                print()
                self.fix_spot(row - 1, col - 1, player)
                if self.is_player_win(player):
                    print(f"Player {player} wins the game!")
                    break
                if self.is_board_filled():
```

```
                print("Match Draw!")
                break
            player = self.swap_player_turn(player)
        print()
        self.show_board()
tic_tac_toe = TicTacToe()
tic_tac_toe.start()
```

Output Snapshot

```
Player O turn
- - -
- - -
- - -
Enter row and column numbers to fix spot: 0 3
Player X turn
- - -
- - -
- - O
Enter row and column numbers to fix spot: 1 2
Player O turn
- X - - - -
- - O
Enter row and column numbers to fix spot: 3 0
Player X turn
- X -
- - -
- - O
Enter row and column numbers to fix spot: 3 2
Player O turn
- X -
- - -
- X O
Enter row and column numbers to fix spot: 2 1
Player X turn
- X -
O - -
- X O
Enter row and column numbers to fix spot: 2 2
Player X wins the game!
```

Program 2 - Vacuum Cleaner

Code

```
def vacuum_world():

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

    location_input = input("Enter Location of Vacuum: ")
    status_input = input("Enter status of " + location_input+ " : ")
    status_input_complement = input("Enter status of other room : ")
print("Initial Location Condition {A : " + str(status_input_complement) + ", B : " + str(status_input) + " }"
    if location_input == 'A':
        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")
            goal_state['A'] = '0'
            cost += 1 #cost for suck
            print("Cost for CLEANING A " + str(cost))
            print("Location A has been Cleaned.")

            if status_input_complement == '1':
                print("Location B is Dirty.")
                print("Moving right to the Location B. ")
                cost += 1
                print("COST for moving RIGHT " + str(cost))
                goal_state['B'] = '0'
                cost += 1
                print("COST for SUCK " + str(cost))
                print("Location B has been Cleaned. ")
            else:
                print("No action" + str(cost))
                print("Location B is already clean.")

        if status_input == '0':
            print("Location A is already clean ")
            if status_input_complement == '1':
                print("Location B is Dirty.")
                print("Moving RIGHT to the Location B. ")
                cost += 1
                print("COST for moving RIGHT " + str(cost))
                goal_state['B'] = '0'
                cost += 1
```

```
            print("Cost for SUCK" + str(cost))
            print("Location B has been Cleaned. ")
        else:
            print("No action " + str(cost))
            print(cost)
            print("Location B is already clean.")

else:
    print("Vacuum is placed in location B")
    if status_input == '1':
        print("Location B is Dirty.")
        goal_state['B'] = '0'
        cost += 1
        print("COST for CLEANING " + str(cost))



        print("Location B has been Cleaned.")

        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("Moving LEFT to the Location A. ")
            cost += 1
            print("COST for moving LEFT " + str(cost))
            goal_state['A'] = '0'
            cost += 1
            print("COST for SUCK " + str(cost))
            print("Location A has been Cleaned.")

    else:
        print(cost)
        print("Location B is already clean.")

        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("Moving LEFT to the Location A. ")
            cost += 1
            print("COST for moving LEFT " + str(cost))
            goal_state['A'] = '0'
            cost += 1
            print("Cost for SUCK " + str(cost))
            print("Location A has been Cleaned. ")
        else:
```

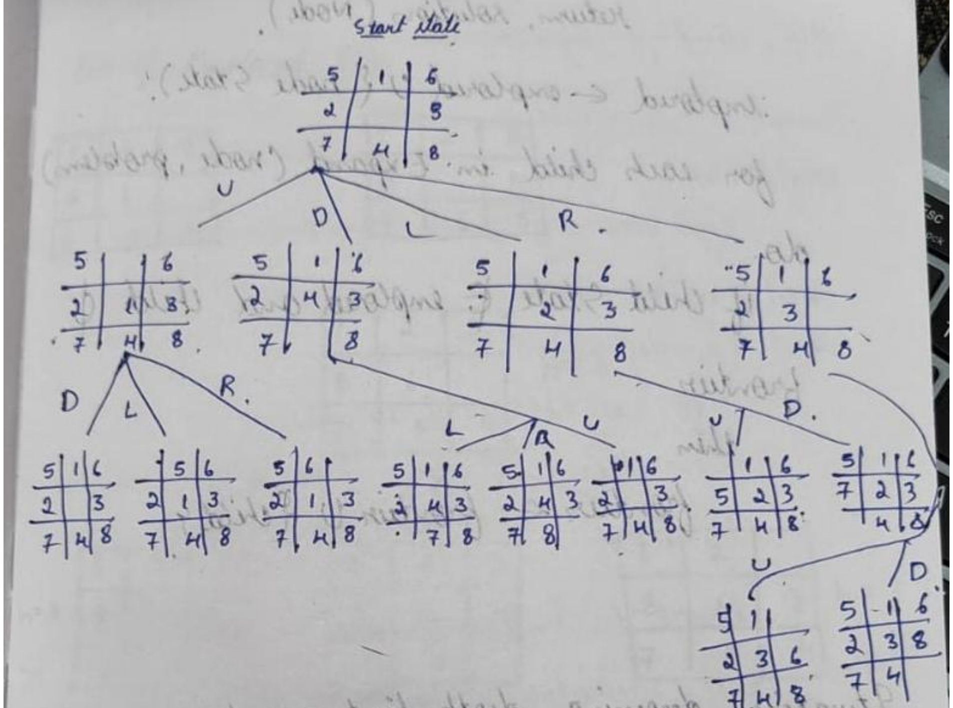
```
        print("No action " + str(cost))
        print("Location A is already clean.")

print("GOAL STATE: ")
print(goal_state)
print("Performance Measurement: " + str(cost))
vacuum_world()
```

Output Snapshot

```
Enter Location of Vacuum: A
Enter status of A : 0
Enter status of other room : 1
Initial Location Condition {A : 1, B : 0 }
Vacuum is placed in Location A
Location A is already clean
Location B is Dirty.
Moving RIGHT to the Location B.
COST for moving RIGHT 1
Cost for SUCK2
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 2
```

Program 3 - 8 Puzzle Using BFS

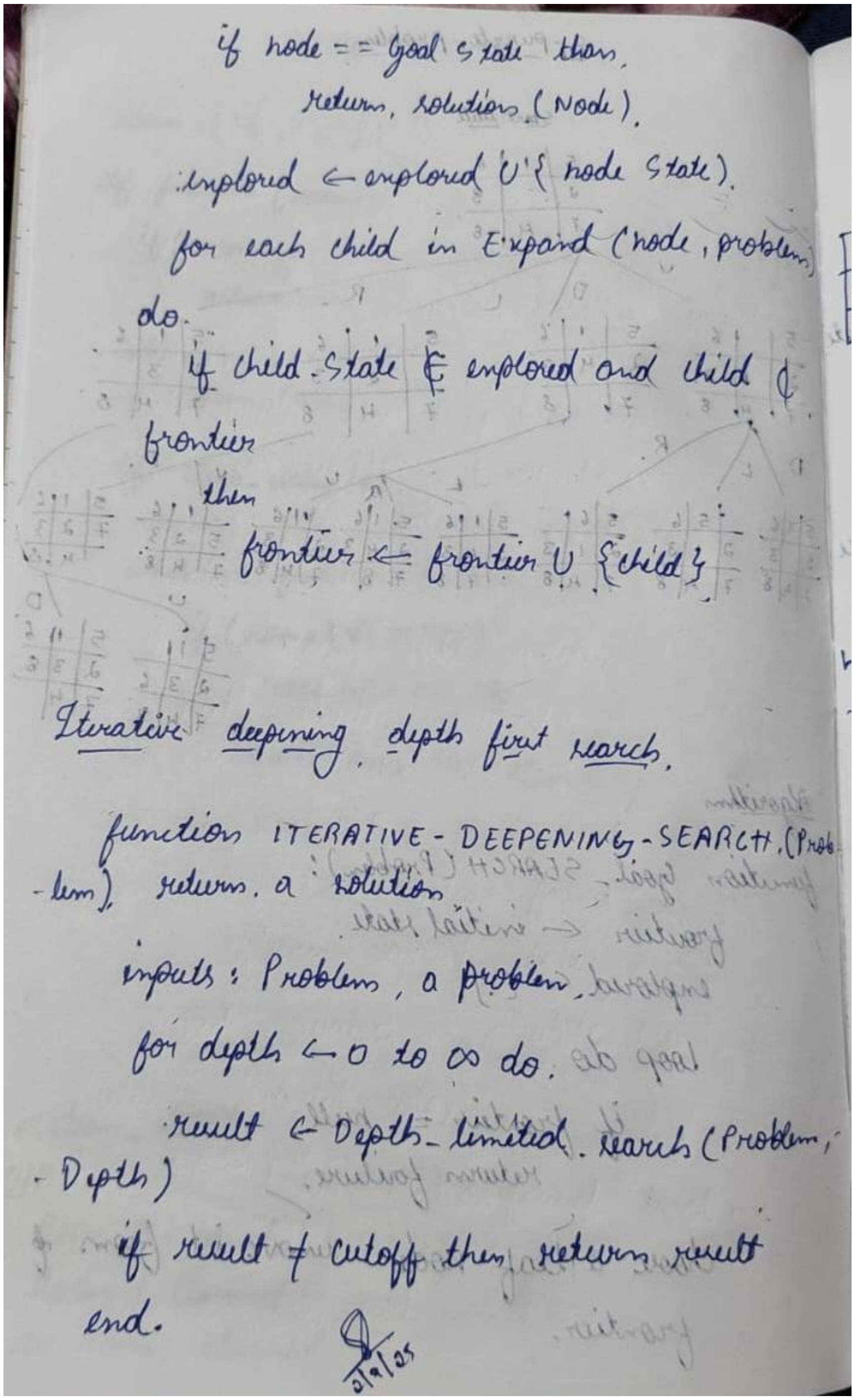
8-puzzle problem



Algorithm

function Goal_SEARCH (Problem):
frontier ← initial state.
explored ← [ ]
loop do
if frontier =; null,
return failure;
choose a leaf node remove it from ~~f~~
frontier.

Algorithm :



if node == Goal state then
return solution (Node).
explored ← explored ∪ {node State}.
for each child in Expand (node, problem)
do.
if child.State ∉ explored and child ∉ frontier
then
frontier ← frontier ∪ {child}

Iterative deepening depth first search.

function ITERATIVE-DEEPENING-SEARCH (Problem) returns a solution
inputs: Problem, a problem
for depth ← 0 to ∞ do.
result ← Depth_limited_search (Problem, Depth)
if result ≠ cutoff then return result
end.

-

Code

```
import sys
import numpy as np
class Node:
        def __init__(self, state, parent, action):
                self.state = state
                self.parent = parent
```

```
        self.action = action
class StackFrontier:
    def __init__(self):
        self.frontier = []
    def add(self, node):
        self.frontier.append(node)
    def contains_state(self, state):
        return any((node.state[0] == state[0]).all() for node in self.frontier)
    def empty(self):
        return len(self.frontier) == 0
    def remove(self):
        if self.empty():
            raise Exception("Empty Frontier")
        else:
            node = self.frontier[-1]
            self.frontier = self.frontier[:-1]
            return no
class QueueFrontier(StackFrontier):
    def remove(self):
        if self.empty():
            raise Exception("Empty Frontier")
        else:   node = self.frontier[0]
            self.frontier = self.frontier[1:]
            return node
class Puzzle:
    def _init_(self, start, startIndex, goal, goalIndex):
        self.start = [start, startIndex]
        self.goal = [goal, goalIndex]
        self.solution = None

    def neighbors(self, state):
        mat, (row, col) = state
```

```
results = [] if row > 0:
        mat1 = np.copy(mat)
        mat1[row][col] = mat1[row - 1][col]
        mat1[row - 1][col] = 0
        results.append(('up', [mat1, (row - 1, col)]))
    if col > 0:
        mat1 = np.copy(mat)
        mat1[row][col] = mat1[row][col - 1]
        mat1[row][col - 1] = 0
        results.append(('left', [mat1, (row, col - 1)]))
    if row < 2:
        mat1 = np.copy(mat)
        mat1[row][col] = mat1[row + 1][col]
        mat1[row + 1][col] = 0
        results.append(('down', [mat1, (row + 1, col)]))
    if col < 2:
        mat1 = np.copy(mat)
        mat1[row][col] = mat1[row][col + 1]
        mat1[row][col + 1] = 0
        results.append(('right', [mat1, (row, col + 1)]))
    return results
def print(self):
    solution = self.solution if self.solution is not None else None



    print("Start State:\n", self.start[0], "\n")
    print("Goal State:\n", self.goal[0], "\n")
    print("\nStates Explored: ", self.num_explored, "\n")
    print("Solution:\n ")
    for action, cell in zip(solution[0], solution[1]):
        print("action: ", action, "\n", cell[0], "\n")
    print("Goal Reached!!")
def does_not_contain_state(self, state):
    for st in self.explored:
        if (st[0] == state[0]).all():
        return False
    return True
def solve(self):
    self.num_explored = 0
    start = Node(state=self.start, parent=None, action=None)
```

```
        frontier = QueueFrontier()
        frontier.add(start)
        self.explored = []
        while True:
            if frontier.empty():
                raise Exception("No solution")
            node = frontier.remove()
            self.num_explored += 1
            if (node.state[0] == self.goal[0]).all():
                actions = []
                cells = []
                while node.parent is not None:
                    actions.append(node.action)
                    cells.append(node.state)
                    node = node.parent
                actions.reverse()
                cells.reverse()
                self.solution = (actions, cells)
                return
            self.explored.append(node.state)
            for action, state in self.neighbors(node.state):
    if not frontier.contains_state(state) and self.does_not_contain_state(state): child =
                    Node(state=state, parent=node, action=action)
                    frontier.add(child)
start = np.array([[1, 2, 3], [8, 0, 4], [7, 6, 5]])



goal = np.array([[2, 8, 1], [0, 4, 3], [7, 6, 5]])
startIndex = (1, 1)
goalIndex = (1, 0)
p = Puzzle(start, startIndex, goal, goalIndex)
p.solve() p.print()
```

Output Snapshot

```
Start State:
 [[1 2 3]
 [8 0 4]
 [7 6 5]]

Goal State:
 [[2 8 1]
 [0 4 3]
 [7 6 5]]


States Explored:  358

Solution:

action:  up
 [[1 0 3]
 [8 2 4]
 [7 6 5]]

action:  left
 [[0 1 3]
 [8 2 4]
 [7 6 5]]

action:  down
 [[8 1 3]
 [0 2 4]
 [7 6 5]]

action:  right
 [[8 1 3]
 [2 0 4]
 [7 6 5]]

action:  right
 [[8 1 3]
 [2 4 0]
 [7 6 5]]
```

```
action:  down
 [[8 1 3]
 [0 2 4]
 [7 6 5]]

action:  right
 [[8 1 3]
 [2 0 4]
 [7 6 5]]

action:  right
 [[8 1 3]
 [2 4 0]
 [7 6 5]]

action:  up
 [[8 1 0]
 [2 4 3]
 [7 6 5]]

action:  left
 [[8 0 1]
 [2 4 3]
 [7 6 5]]

action:  left
 [[0 8 1]
 [2 4 3]
 [7 6 5]]

action:  down
 [[2 8 1]
 [0 4 3]
 [7 6 5]]

Goal Reached!!
```

Program 04 - 8 Puzzle Using A*

Algorithm

A* Algorithm on 8 puzzle

Algorithm

function A* (start, goal)

open list ← priority queue ordered by $f(n) = g(n) + h(n)$

closed_list ← empty set

g (start) ← 0.

f (start) ← g (start) + h (start)

open_list . insert (start, f (start)

while open_list is not empty

current ← node in open_list with lowest f values.

if current = goal :

returns path from start to goal.

add current to closed_list

for each neighbour of current:

if neighbour. is closed_list:

continue

tentative_g ← g (current) + cost (current, neighbour) if neighbour not in open_list or.

tentative_g < g (neighbour)

g (neighbour) ← tentative_g.

f (neighbour) ← g (neighbour) + h(neighbour)

Parent (neighbour) ← current.

if neighbour not in open list.

open_list.insert (neighbour, f(neighbour))

returns failure.

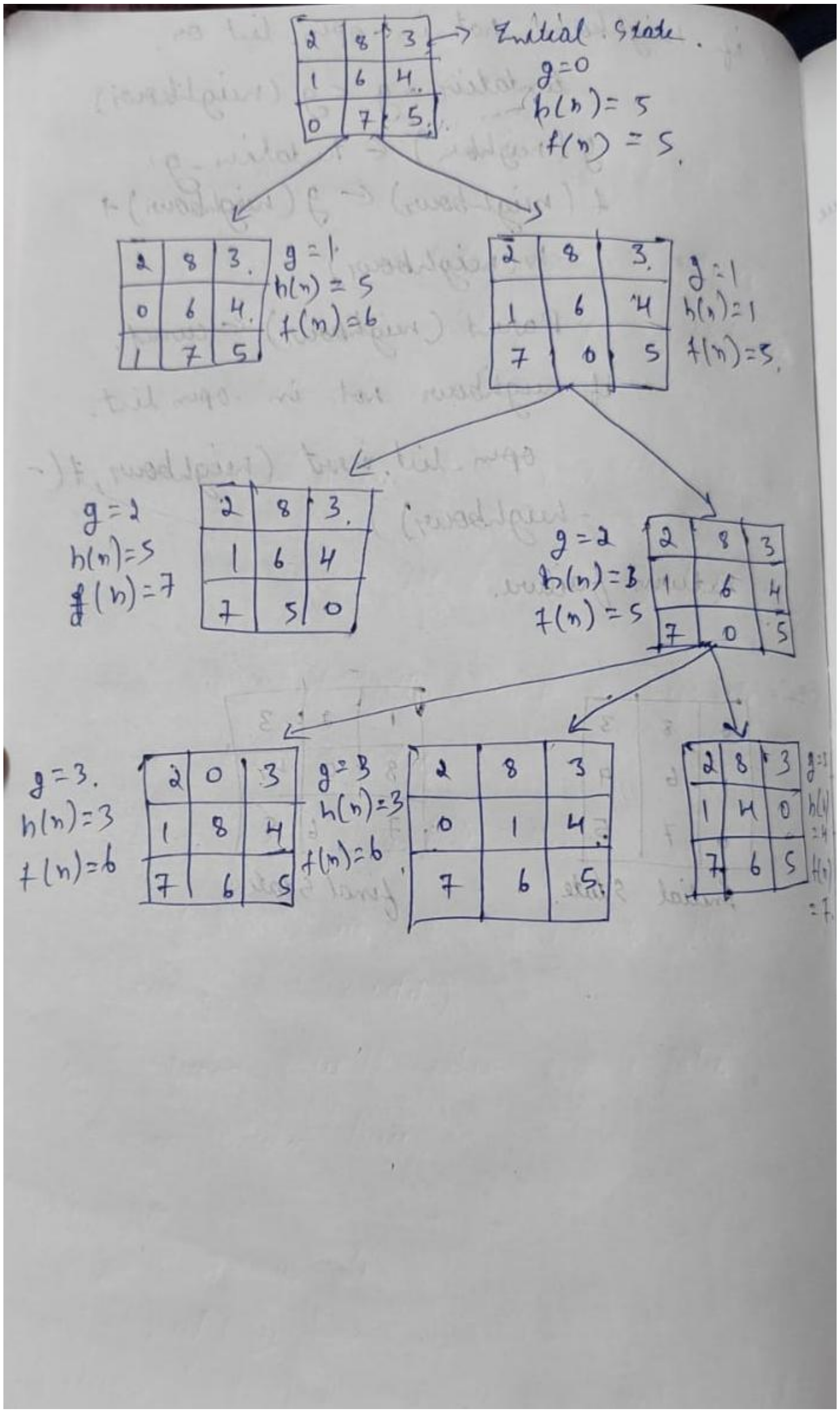
Ex:-

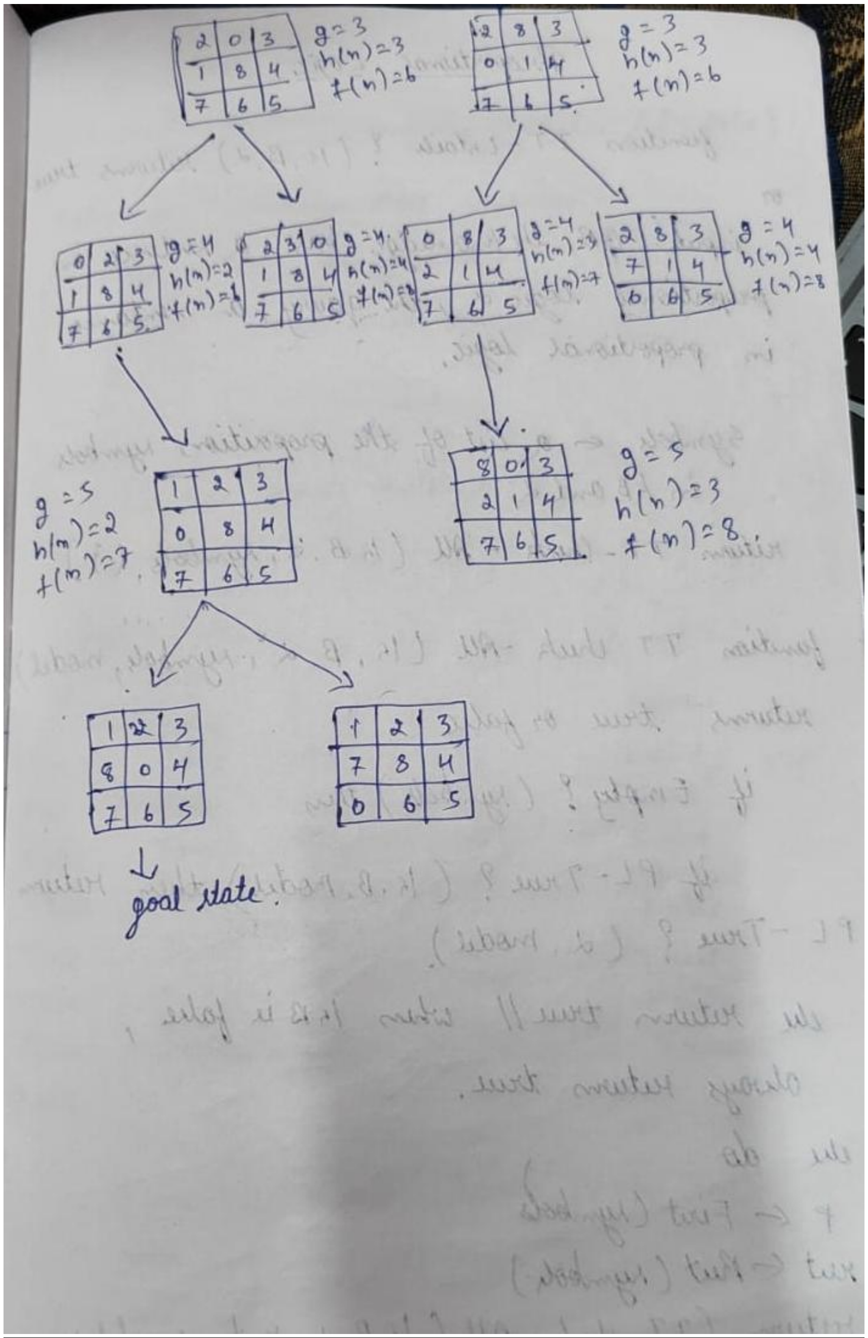
2	8	3
1	6	4
0	7	5

Initial state.

1	2	3.
8	0	4.
7	6	5

final state


Initial State.
g=0
h(n)= 5
f(n) = 5.
g = 1
h(n) = 5
f(n) = 6
g = 1
h(n) = 1
f(n) = 5.
g = 2
h(n) = 5
f(n) = 7
g = 2
h(n) = 3
f(n) = 5
g = 3.
h(n) = 3
f(n) = 6
g = 3
h(n) = 3
f(n) = 6



Code

```
def print_b(src):
    state = src.copy()
    state[state.index(-1)] = ' '
    print(
f"""
 {state[0]}  {state[1]}  {state[2]}
 {state[3]}  {state[4]}  {state[5]}
 {state[6]}  {state[7]}  {state[8]}
“””
    )
```

```
def h(state, target):
    count = 0
    i = 0
    for j in state:
        if state[i] != target[i]:
            count = count+1
    return count
def astar(state, target):
    states = [src]
    g = 0
    visited_states = []
    while len(states):
        print(f"Level: {g}")
        moves = []
        for state in states:
            visited_states.append(state)
            print_b(state)
            if state == target:
                print("Success")
                return
            moves += [move for move in possible_moves(
                state, visited_states) if move not in moves]
        costs = [g + h(move, target) for move in moves]
        states = [moves[i]
                for i in range(len(moves)) if costs[i] == min(costs)]
        g += 1
    print("Fail")
def possible_moves(state, visited_state):
```

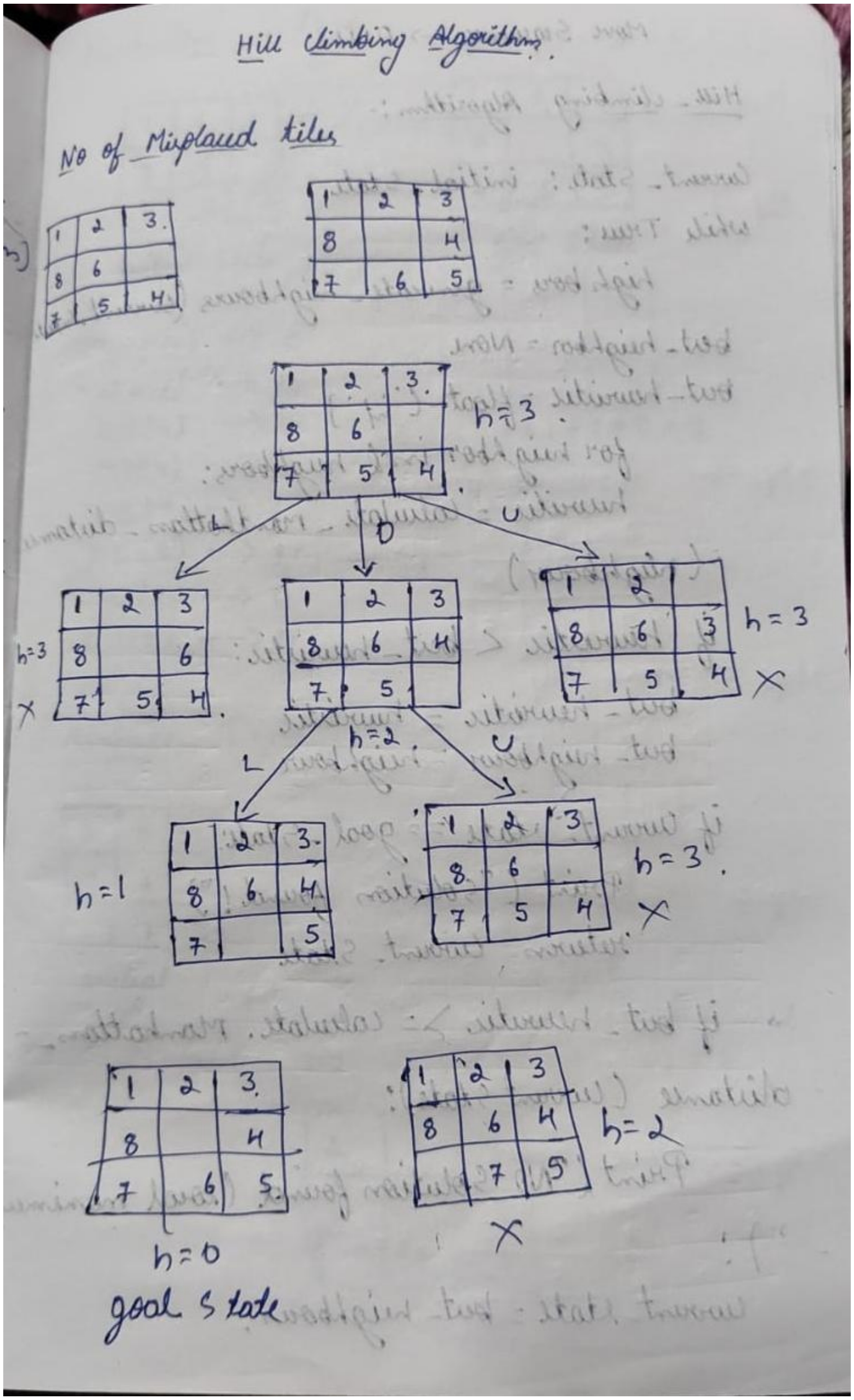
```
    b = state.index(-1)
    d = []
    if b - 3 in range(9):
        d.append('u')
    if b not in [0, 3, 6]:
        d.append('l')
    if b not in [2, 5, 8]:
        d.append('r')
    if b + 3 in range(9):
        d.append('d')
    pos_moves = []
    for m in d:
        pos_moves.append(gen(state, m, b))
    return [move for move in pos_moves if move not in visited_state]
def gen(state, m, b):
    temp = state.copy()
    if m == 'u':
        temp[b - 3], temp[b] = temp[b], temp[b - 3]
    if m == 'l':
        temp[b - 1], temp[b] = temp[b], temp[b - 1]
    if m == 'r':
        temp[b + 1], temp[b] = temp[b], temp[b + 1]
    if m == 'd':
        temp[b + 3], temp[b] = temp[b], temp[b + 3]
    return temp
src = [1, 2, 3, -1, 4, 5, 6, 7, 8]
target = [1, 2, 3, 4, 5,6, 7, 8,-1]
astar(src, target)
```

Output Snapshot

```
Enter the start state matrix

1 0 1 0
1 0 0 1
1 1 1 1
Enter the goal state matrix

1 1 0 1
1 0 0 1
1 1 1 0
|
  |
 \'/

1 0 1 0
1 0 0 1
1 1 1 1
```

Program 05 – Hill Climbing

Algorithm



Move Sequence → DLU

Hill_climbing_Algorithm:-

```
current_state: initial_state
while True:
    neighbors = generate_neighbours (current_state)
    best_neighbor = None
    best_heuristic = float ('inf')
    for neighbor in neighbors:
        heuristic = calculate_manhattan_distance
        (neighbour)
        if heuristic < best_heuristic:
            best_heuristic = heuristic
            best_neighbour = neighbour
    if current_state == goal_state:
        Print ("Solution found!")
        return current_state
    if best_heuristic >= calculate_manhattan_distance (current state):
        Print ("No solution found (local maximum)")
    current_state = best_neighbour
```

Manhattan Distance :-

1	2	3
8	6	
7	5	4

Initial State

1	2	3
8		4
7	6	5

Goal State.

MD(1) → 0
MD(2) → 0
MD(3) → 0
MD(4) → 1
MD(5) → 1
MD(6) → 1
MD(7) → 0
MD(8) → 0

$0+0+1+1+1+0+0$

$= 3$

ii) Manhattan Distance

2	8	3
1	6	4
	7	5

initial state.

1	2	3
8		4
7	6	5

Goal state.

$h = 1+1+0+0+0+1+1+2 = 6$

2	8	3
1	6	4
	7	5

Blank at (3,1) can move right h = 5 but

start → R → U → U → L → D → R

step - 1 : move right

h = 5.

2	8	3
1	6	4
7		5

step - 2 move up

2	8	3
1		4
7	6	5

h = 4

step 3 : move up.

2		3
1	8	4
7	6	5

h = 3.

step 4 : move left

1	2	3
1		4
7	6	5

h = 3.

step 5 : Move down.

1	2	3
	8	4
7	6	5

h = 2 .

step : 6 Move right

1	2	3
8		4
7	6	5

h = 0.

Manhattan distance Algorithm.

1. start with Initial state
2. calculate the Manhattan distance of the current state
3. while the current state is not goal.
   a. Generate all valid neighbouring state by sliding the blank tile up/down/left/right.
   b. calculate the Manhattan distance for each neighbour.
   c. choose the neighbor with lowest manhattan distance
4. If the Manhattan dist is zero goal state is reached.

Code

```
from random import randint
N = 8
def configureRandomly(board, state):
    for i in range(N):
        state[i] = randint(0, 100000) % N;
        board[state[i]][i] = 1;
  def printBoard(board):
    for i in range(N):
        print(*board[i])
def printState( state):
    print(*state)
def compareStates(state1, state2):
    for i in range(N):
        if (state1[i] != state2[i]):
```

```
                return False;
        return True;
    def fill(board, value):
        for i in range(N):
                for j in range(N):
                        board[i][j] = value;
def calculateObjective( board, state):
        for i in range(N):
                row = state[i]
                col = i - 1;
                while (col >= 0 and board[row][col] != 1) :
                        col -= 1
                if (col >= 0 and board[row][col] == 1) :
                        attacking += 1;
                row = state[i]
                col = i + 1;
                while (col < N and board[row][col] != 1):
                        col += 1;
                if (col < N and board[row][col] == 1) :
                        attacking += 1;
                row = state[i] - 1
                col = i - 1;
                while (col >= 0 and row >= 0 and board[row][col] != 1) :
                        col-= 1;
                        row-= 1;

                if (col >= 0 and row >= 0 and board[row][col] == 1) :
                        attacking+= 1;

                # Diagonally to the right down
                # (row and col simultaneously
                # increase)
                row = state[i] + 1
                col = i + 1;
                while (col < N and row < N and board[row][col] != 1) :
                        col+= 1;
                        row+= 1;

                if (col < N and row < N and board[row][col] == 1) :
                        attacking += 1;
```

```
                row = state[i] + 1
                col = i - 1;
                while (col >= 0 and row < N and board[row][col] != 1) :
                        col -= 1;
                        row += 1;
                if (col >= 0 and row < N and board[row][col] == 1) :
                        attacking += 1;
                row = state[i] - 1
                col = i + 1;
                while (col < N and row >= 0 and board[row][col] != 1) :
                        col += 1;
                        row -= 1;
                if (col < N and row >= 0 and board[row][col] == 1) :
                        attacking += 1;
        return int(attacking / 2);
def generateBoard( board, state):
        fill(board, 0);
        for i in range(N):
                board[state[i]][i] = 1;
     def copyState( state1, state2):

        for i in range(N):
                state1[i] = state2[i];

     def getNeighbour(board, state):
        opBoard = [[0 for _ in range(N)] for _ in range(N)]
        opState = [0 for _ in range(N)]

        copyState(opState, state);
        generateBoard(opBoard, opState);
        opObjective = calculateObjective(opBoard, opState);
        NeighbourBoard = [[0 for _ in range(N)] for _ in range(N)]

        NeighbourState = [0 for _ in range(N)]
        copyState(NeighbourState, state);
        generateBoard(NeighbourBoard, NeighbourState);
        for i in range(N):
                for j in range(N):
                        if (j != state[i]) :
                                NeighbourState[i] = j;
```

```
                NeighbourBoard[NeighbourState[i]][i] = 1;
                NeighbourBoard[state[i]][i] = 0;
                temp = calculateObjective( NeighbourBoard, NeighbourState)
                if (temp <= opObjective) :
                    opObjective = temp;
                    copyState(opState, NeighbourState);
                    generateBoard(opBoard, opState);
                NeighbourBoard[NeighbourState[i]][i] = 0;
                NeighbourState[i] = state[i];
                NeighbourBoard[state[i]][i] = 1;
    copyState(state, opState);
    fill(board, 0);
    generateBoard(board, state);

def hillClimbing(board, state):
    neighbourBoard = [[0 for _ in range(N)] for _ in range(N)
    neighbourState = [0 for _ in range(N)]
    copyState(neighbourState, state);
    generateBoard(neighbourBoard, neighbourState);

    while True:
        copyState(state, neighbourState);
        generateBoard(board, state);
        # Getting the optimal neighbour
        getNeighbour(neighbourBoard, neighbourState);
        if (compareStates(state, neighbourState)) :

            printBoard(board);
            break;

        elif (calculateObjective(board, state) == calculateObjective(
neighbourBoard,neighbourState)):
            # Random neighbour
            neighbourState[randint(0, 100000) % N] = randint(0, 100000) % N;
            generateBoard(neighbourBoard, neighbourState);

# Driver code
state = [0] * N
board = [[0 for _ in range(N)] for _ in range(N)]
configureRandomly(board, state);
```

```
hillClimbing(board, state);
```

OUTPUT

```
Step 0: Initial state
. . . Q
. Q . .
. . Q .
Q . . .

Cost = 2

Step 1: Move to better neighbour
. . . Q
Q . . .
. . Q .
. Q . .

Cost = 1

Step 2: Move to better neighbour
. . Q .
Q . . .
. . . Q
. Q . .

Cost = 0

Step 3: Reached local minimum
Final state:
. . Q .
Q . . .
. . . Q
. Q . .

Final cost = 0
```

Program-07 Knowledge-Base

ALGORITHM :

Proportional Logic

function TT-Entails ? (KB, α) returns true or

inputs:- KB, the knowledge base, a sentence in proportional logic α, the query, a sentance in proportional logic.

Symbols ← a list of the proposition symbols in KB and α

return TT-Check-All (KB. α, symbols, { })

function TT check-All (KB, α, symbols, model) returns true or false

if Empty ? (symbols) then

if PL-True ? (KB, model) then return PL-True ? (α, model).

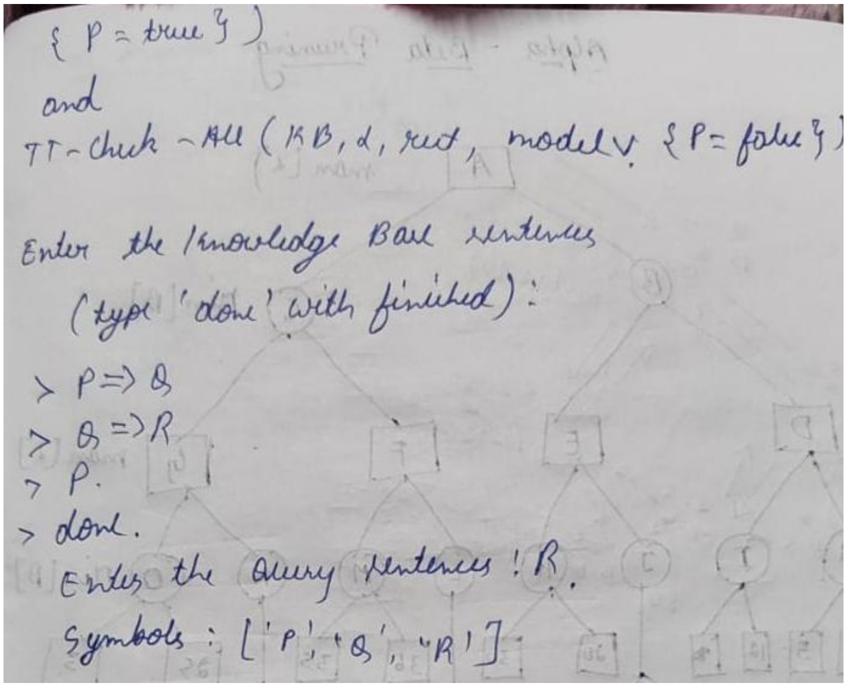
else return true // when KB is false, always return true.

else do

P ← First (symbols)

rest ← Rest (symbols)

return (TT-check-All (KB, α, rest, model ∪


{ P = true })
and
TT-Check-All (KB, α, rest, model ∨ {P = false})

Enter the Knowledge Base sentences
(type 'done' with finished):
> P => Q
> Q => R
> P.
> done.
Enter the Query sentences ! R.
Symbols : ['P', 'Q', 'R']

Code

```
combinations=[(True,True, True),(True,True,False),(True,False,True),(True,False,
False),(False,True, True),(False,True, False),(False, False,True),(False,False, False)]
variable={'p':0,'q':1, 'r':2}
kb="
q="
priority={'~':3,'v':1,'^':2}
def input_rules()
        global kb, q
  kb = (input("Enter rule: "))
  q = input("Enter the Query: ")
def entailment():
  global kb, q
  print("*10+"Truth Table Reference"+"*10)
  print('kb','alpha')
  print('*'*10)
  for comb in combinations:
    s = evaluatePostfix(toPostfix(kb), comb)
    f = evaluatePostfix(toPostfix(q), comb)
    print(s, f)
    print('-'*10)
    if s and not f:
      return False
  return True
def isOperand(c):
  return c.isalpha() and c!='v'
```

```
def isLeftParanthesis(c):
    return c == '('

def isRightParanthesis(c):
    return c == ')'

def isEmpty(stack):
    return len(stack) == 0

def peek(stack):
    return stack[-1]

def hasLessOrEqualPriority(c1, c2):
    try:
        return priority[c1]<=priority[c2]
    except KeyError:
        return False
def toPostfix(infix):
    stack = []
    postfix = ''
    for c in infix:
        if isOperand(c):
        postfix += c
        else:
            if isLeftParanthesis(c):
                stack.append(c)
            elif isRightParanthesis(c):
                operator = stack.pop()
                while not isLeftParanthesis(operator):
                    postfix += operator
                    operator = stack.pop()
            else:
                while (not isEmpty(stack)) and hasLessOrEqualPriority(c, peek(stack)):
                    postfix += stack.pop()
                stack.append(c)
    while (not isEmpty(stack)):
        postfix += stack.pop()

    return postfix
def evaluatePostfix(exp, comb):
    stack = []
    for i in exp:
```

```
        if isOperand(i):
            stack.append(comb[variable[i]])
        elif i == '~':
            val1 = stack.pop()
            stack.append(not val1)
        else:
            val1 = stack.pop()
            val2 = stack.pop()
            stack.append(_eval(i,val2,val1))
    return stack.pop()
def _eval(i, val1, val2):
    if i == '^':
        return val2 and val1
    return val2 or val1
#Test 1
input_rules()
ans = entailment()
if ans:
print("Knowledge Base entails query") else:
    print("Knowledge Base does not entail query")
#Test 2
input_rules()
ans = entailment()

if ans:
    print("Knowledge Base entails query")
else:
    print("Knowledge Base does not entail query")
```

OUTPUT:

```
Enter rule: (~qv~pvr)^(~q^p)^q
Enter the Query: r
Truth Table Reference
kb alpha
**********
False True
----------
False False
----------
False True
----------
False False
----------
False True
----------
False False
----------
False True
----------
False False
----------
Knowledge Base entails query
```

Program-08 Unification in first order logic

Algorithm :

Unification in FOL

Algorithm: unify(ϕ_1, ϕ_2)

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

a) If ϕ_1 or ϕ_2 are indentical, then Return NIL.

b) Else if ϕ_1 is available,

a. then if ϕ_1 occurs in ϕ_2, then Return Failure

b. Else return $\{(\phi_2, \phi_1)\}$

c) Else if ϕ_2 is a variable,

a. if ϕ_2 occurs in ϕ_1, then return Failure

b. Else return $\{(\phi_1/\phi_2)\}$

d) else return Failure.

Step 2:- if the initial predicate symbol in ϕ_1 & ϕ_2 are not same, then return Failure.

Step 3:- If ϕ_1 & ϕ_2 have a different number of arguments, then return failure.

Step 4:- Set substitution set (SUB) to NIL.

Step 5:- for i=1 to the number of elements in ϕ_1,

a) call unify function with the ith element of ϕ_1 and ith element of ϕ_2 and put the result into S.

b) if S = failure then Return Failure

c) If $S \neq$ NIL then do

a) apply S to the remainder of both L_1 and L_2

b) Subst = append (S, SUBST)

Step 6: Return SUBST

Code

```
import re
def getAttributes(expression):
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = expression.split(")")[:-1]
    expression = ")".join(expression)
    attributes = expression.split(',')
    return attributes

def getInitialPredicate(expression):
    return expression.split("(")[0]

def isConstant(char):
    return char.isupper() and len(char) == 1

def isVariable(char):
    return char.islower() and len(char) == 1

def replaceAttributes(exp, old, new):



    attributes = getAttributes(exp)
    predicate = getInitialPredicate(exp)
    for index, val in enumerate(attributes):
        if val == old:
            attributes[index] = new
    return predicate + "(" + ",".join(attributes) + ")"

def apply(exp, substitutions):
    for substitution in substitutions:
        new, old = substitution
        exp = replaceAttributes(exp, old, new)
    return exp

def checkOccurs(var, exp):
    if exp.find(var) == -1:
        return False
    return True
def getFirstPart(expression):
```

```
    attributes = getAttributes(expression)
    return attributes[0]
def getRemainingPart(expression):
    predicate = getInitialPredicate(expression)
    attributes = getAttributes(expression)
    newExpression = predicate + "(" + ",".join(attributes[1:]) + ")"
    return newExpression
def unify(exp1, exp2):
    if exp1 == exp2:
        return []
    if isConstant(exp1) and isConstant(exp2):
        if exp1 != exp2:
            print(f"{exp1} and {exp2} are constants. Cannot be unified")
            return []

    if isConstant(exp1):
        return [(exp1, exp2)]

    if isConstant(exp2):
        return [(exp2, exp1)]

    if isVariable(exp1):
      return [(exp2, exp1)] if not checkOccurs(exp1, exp2) else [] if isVariable(exp2):
        return [(exp1, exp2)] if not checkOccurs(exp2, exp1) else []

    if getInitialPredicate(exp1) != getInitialPredicate(exp2):
        print("Cannot be unified as the predicates do not match!")
        return []

    attributeCount1 = len(getAttributes(exp1))
    attributeCount2 = len(getAttributes(exp2))
    if attributeCount1 != attributeCount2:
        print(f"Length of attributes {attributeCount1} and {attributeCount2} do not match. Cannot
be unified")
        return []

    head1 = getFirstPart(exp1)
    head2 = getFirstPart(exp2)
    initialSubstitution = unify(head1, head2)
    if not initialSubstitution:
        return []
```

```
    if attributeCount1 == 1:
        return initialSubstitution

    tail1 = getRemainingPart(exp1)
    tail2 = getRemainingPart(exp2)

    if initialSubstitution != []:
        tail1 = apply(tail1, initialSubstitution)
        tail2 = apply(tail2, initialSubstitution)

    remainingSubstitution = unify(tail1, tail2)
    if not remainingSubstitution:
        return []

    return initialSubstitution + remainingSubstitution

if __name__ == "__main__":
    print("Enter the first expression")
    e1 = input()



print("Enter the second expression")
e2 = input()
substitutions = unify(e1, e2)
print("The substitutions are:")
print([' / '.join(substitution) for substitution in substitutions])
```

Output Snapshot

```
Enter the first expression
king(x)
Enter the second expression
king(john)
The substitutions are:
['john / x']
```

Program-9 Forward Reasoning

ALGORITHM :

First order logic: Forward Chaining

Example:

As per the law. it is a crime for an American to sell wepons to hostile nations. Country A, an enemy of America, has some missiles and all the missiles where sold to it by Robert, who is an American citizen

Prove that "Robert is a criminal."

Representation in FOL

- It is a crime for an american to sell wepons to hostile Nations.

Lets say P, Q and R are variables

American (P) $\wedge$ weapons (Q) $\wedge$ sells (P, Q, R) $\wedge$ Hostile (R) => criminal (P)

- Country A has some missiles

$\exists x$ owns (A, x) $\wedge$ missiles (x)

- Existential instantiation, introducing a new constant T_1, "owns (A, T_1). missile (T_1)

- All missiles were sold to country A by Robert

$\forall x$ Missiles (x) $\wedge$ Owns (A, x) => sells (Robert, x

- Missiles as wepons

Missile (x) => wepon (x)

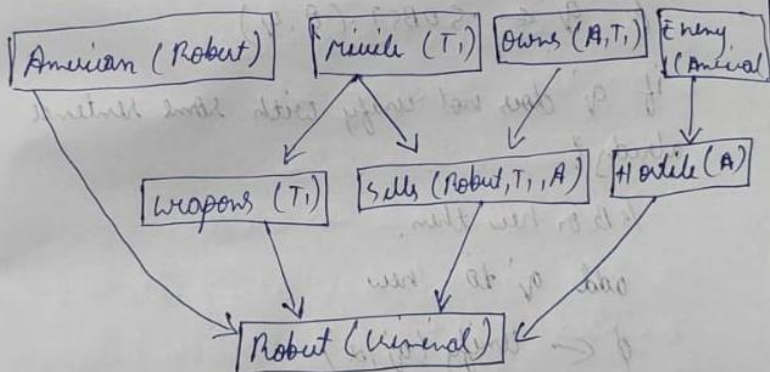
- Enemy of America is known as hostile
  the Enemy (x, America) => Hostile (x).

- Robut is an American
  American (Robut)

- the country A, an Enemy of America.
  Enemy (A, America)

TO Prove:
Robut is a criminal
criminal (Robut)

Forward chaining:



Forward Reasoning Algorithm:
function FOL-FC-ASK(KB, α)
returns a substitution or false.
inputs: KB, the knowledge base.
α, the query

American(P) ∧ weapon(Q) ∧ sells (P, Q, s) ∧ Hostile => criminal (P)

local variables : new, the new sentences infer-
-ed on each iteration

repeat until new is empty

$New \leftarrow \{\}$

for each rule in KB do

$(P_1 \wedge \dots \wedge P_n \Rightarrow q) \leftarrow$ Standardize-variables (rule)

for each θ such that Subst $(\theta, P_1 \wedge \dots \wedge P_n) =$

Subst $(\theta, P_1' \wedge \dots \wedge P_n')$

for some $P_1' \dots P_n'$ in KB

$q' \leftarrow$ SUBST (θ, q)

if q' does not unify with some sentence already ?

KB or new then.

add q' to new

$\phi \leftarrow$ Unify (q', α)

if ϕ is not fail then return ϕ

add new to KB

return false.

Code

```
import re
def isVariable(x):
    return len(x) == 1 and x.islower() and x.isalpha()

def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return matches
```

```
def getPredicates(string):
    expr = '([a-z~]+)\([^&|]+\)'
    return re.findall(expr, string)

class Fact:

    def _init_(self, expression):
        self.expression = expression
        predicate, params = self.splitExpression(expression)
        self.predicate = predicate
        self.params = params
        self.result = any(self.getConstants())

    def splitExpression(self, expression):
        predicate = getPredicates(expression)[0]
        params = getAttributes(expression)[0].strip('()').split(',')
        return [predicate, params]
    def getResult(self):
    return self.result

    def getConstants(self):
        return [None if isVariable(c) else c for c in self.params]

    def getVariables(self):
        return [v if isVariable(v) else None for v in self.params]

    def substitute(self, constants):
        c = constants.copy()
        f = f"{self.predicate}({','.join([constants.pop(0) if isVariable(p) else p for p in
self.params])})"
        return Fact(f)

class Implication:

    def _init_(self, expression):
        self.expression = expression
        l = expression.split('=>')
        self.lhs = [Fact(f) for f in l[0].split('&')]
        self.rhs = Fact(l[1])

    def evaluate(self, facts):
        constants = {}
        new_lhs = []
        for fact in facts:
```

```
            for val in self.lhs:
                if val.predicate == fact.predicate:
                    for i, v in enumerate(val.getVariables()):
                        if v:
                            constants[v] = fact.getConstants()[i]
                    new_lhs.append(fact)
        predicate, attributes = getPredicates(self.rhs.expression)[0],
str(getAttributes(self.rhs.expression)[0])
        for key in constants:
            if constants[key]:
                attributes = attributes.replace(key, constants[key])

        expr = f'{predicate}{attributes}'
        return Fact(expr) if len(new_lhs) and all([f.getResult() for f in new_lhs]) else None

class KB:

    def __init__(self):
        self.facts = set()
        self.implications = set()

    def tell(self, e):
        if '=>' in e:
            self.implications.add(Implication(e))
        else:
            self.facts.add(Fact(e))
        for i in self.implications:
            res = i.evaluate(self.facts)
            if res:
                self.facts.add(res)

    def ask(self, e):
        facts = set([f.expression for f in self.facts])
        i = 1
        print(f'Querying {e}:')
        for f in facts:
            if Fact(f).predicate == Fact(e).predicate:
                print(f'\t{i}. {f}')
                i += 1

    def display(self):
        print("All facts: ")
        for i, f in enumerate(set([f.expression for f in self.facts])):
```

```
            print(f'\t{i+1}. {f}')

def main():
    kb = KB()
    print("Enter the number of FOL expressions present in KB:")
    n = int(input())
    print("Enter the expressions:")
    for i in range(n):
        fact = input()
kb.tell(fact)
print("Enter the query:")
query = input()
kb.ask(query)
kb.display()

main()
```

Output Snapshot

```
Querying criminal(x):
1. criminal(West)
All facts:
    1. american(West)
    2. sells(West,M1,Nono)
    3. owns(Nono,M1)
    4. missile(M1)
    5. enemy(Nono,America)
    6. weapon(M1)
    7. hostile(Nono)
    8. criminal(West)
Querying evil(x):
    1. evil(John)
```

Program-10 KnowledgeBase - Resolution

Algorithm :

Resolution in FOL

Steps for proving in Resolution.

Premise, ... Premises

(all expressed in FOL)

1. Convert all series to CNF
2. Negative conclusion S and convert result to CNF
3. Add Negated conclusion S to the premise clauses
4. Repeat until contradiction or no progress is made:
   a. Select 2 clauses (call them parent clauses)
   b. Resolve them together, performing all required unifications
   c. If Resolvent is the empty clause, a contradiction has been found (i.e) S follows from the premises.
   d. If not add Resolved to the prior

Example:

a. John likes all kind of food
b. Apple and vegitables are food
c. Anything anyone eats and not killed is food.
d. Anil eats peanuts and still alive.
e. Harry eats peanuts and still that Anil eats

f. Anyone who is alive implies not killed.
g. Anyone who is not killed ~~A~~ implies alive

<u>Prove by resolution that:</u>

h. John likes peanuts

Step 1: Representation in FOL:

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

b. $food(Apple) \land food(vegetables)$

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

d. $eats(Anil, peanuts) \land alive(Anil)$

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

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

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

h. $likes(John, peanuts)$

Step 2: Eliminate implication

$\alpha \Rightarrow \beta$ with $\neg \alpha \lor \beta$.

a. $\forall x \neg food(x) \lor likes(John, x)$.

b. $food(Apple) \land food(vegetables)$

c. $\forall x \forall y \neg [eats(x, y) \land \neg killed(x)] \lor food(y)$

d. $eats(Anil, peanuts) \land Alive(Anil)$

e. $\forall x \neg eats(Anil, x) \lor eats(Harry, x)$

f. $\forall x \neg alive(x) \lor \neg killed(x)$

g. $\forall x \neg alive(x) \lor \neg killed(x)$

h. likes (John, peanuts)

Step 3: move negation inwards and write

a. $\forall x \neg$ food (x) $\vee$ likes (John, x)

b. food (Apple) $\wedge$ food (vegetables)

c. $\forall x \forall y \neg$ eats (x, y) $\vee$ killed (x) $\vee$ food - (y)

d. eats (Anil, peanuts) $\wedge$ alive (Anil)

e. $\forall x \neg$ eats (Anil. x) $\vee$ eats (Harry, x)

f. $\forall x$ killed (x) $\vee$ alive (x)

g. $\forall x \neg$ alive (x) $\vee \neg$ killed (x)

h. likes (John, Peanuts)

Steps 4: Rename variables or standardize variables

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

b. food (Apple) $\wedge$ food (vegetables)

c. $\forall y \forall z \rightarrow$ eats (y, z) $\vee$ killed (y) $\vee$. food, (z)

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

e. $\forall w \rightarrow$ eats. (Anil, w) $\vee$ eats (Harry, w)

f. $\forall g$ killed (g) $\neg \vee$ alive. (g)

g. $\forall k. \neg$ alive (k) $\vee \neg$ killed

h. likes (John, Peanuts)

Step 5: Drop universal.

a. ¬ food (x) V likes (John, x)
b. food (Apple)
c. food (vegetables)
d. ¬ eats (y, z) V killed (y) V food (z)
e. eats (Anil, Peanuts)
f. alive (Anil)
g. ¬ eats (Anil, w) V eats (Harry, w)
h. killed (g) V alive (g)
i. ¬ alive (k) V ¬ killed (k)
j. likes (John, Peanuts)

¬ likes (John, Peanuts) — food (x) V likes (John, x)
{Peanuts/x}

¬ food (Peanuts) — ¬ eats (y, z) V killed (y) V food (z) {Peanuts/z}

¬ eats (y, Peanuts) V killed (y) — eats (anil, Peanuts)
{Anil/y}

killed (Anil) — ¬ alive (k) V ¬ killed (k)
{Anil/k}

¬ alive (Anil) — alive (Anil)

{ } Hence proved

Code

```
def disjunctify(clauses):
    disjuncts = []
    for clause in clauses:
        disjuncts.append(tuple(clause.split('v')))
return disjuncts

def getResolvant(ci, cj, di, dj):
    resolvant = list(ci) + list(cj)
    resolvant.remove(di)
    resolvant.remove(dj)
    return tuple(resolvant)

def resolve(ci, cj):
    for di in ci:
        for dj in cj:
            if di == '~' + dj or dj == '~' + di:
                return getResolvant(ci, cj, di, dj)

def checkResolution(clauses, query):
    clauses += [query if query.startswith('~') else '~' + query]
    proposition = '^'.join(['(' + clause + ')' for clause in clauses])
    print(f'Trying to prove {proposition} by contradiction.  ')

    clauses = disjunctify(clauses)
    resolved = False
    new = set()

    while not resolved:
        n = len(clauses)


pairs = [(clauses[i], clauses[j]) for i in range(n) for j in range(i + 1, n)]
for (ci, cj) in pairs:
resolvant = resolve(ci, cj)
if not resolvant:
resolved = True
break
new = new.union(set(resolvents))
if new.issubset(set(clauses)):
break
for clause in new:
```

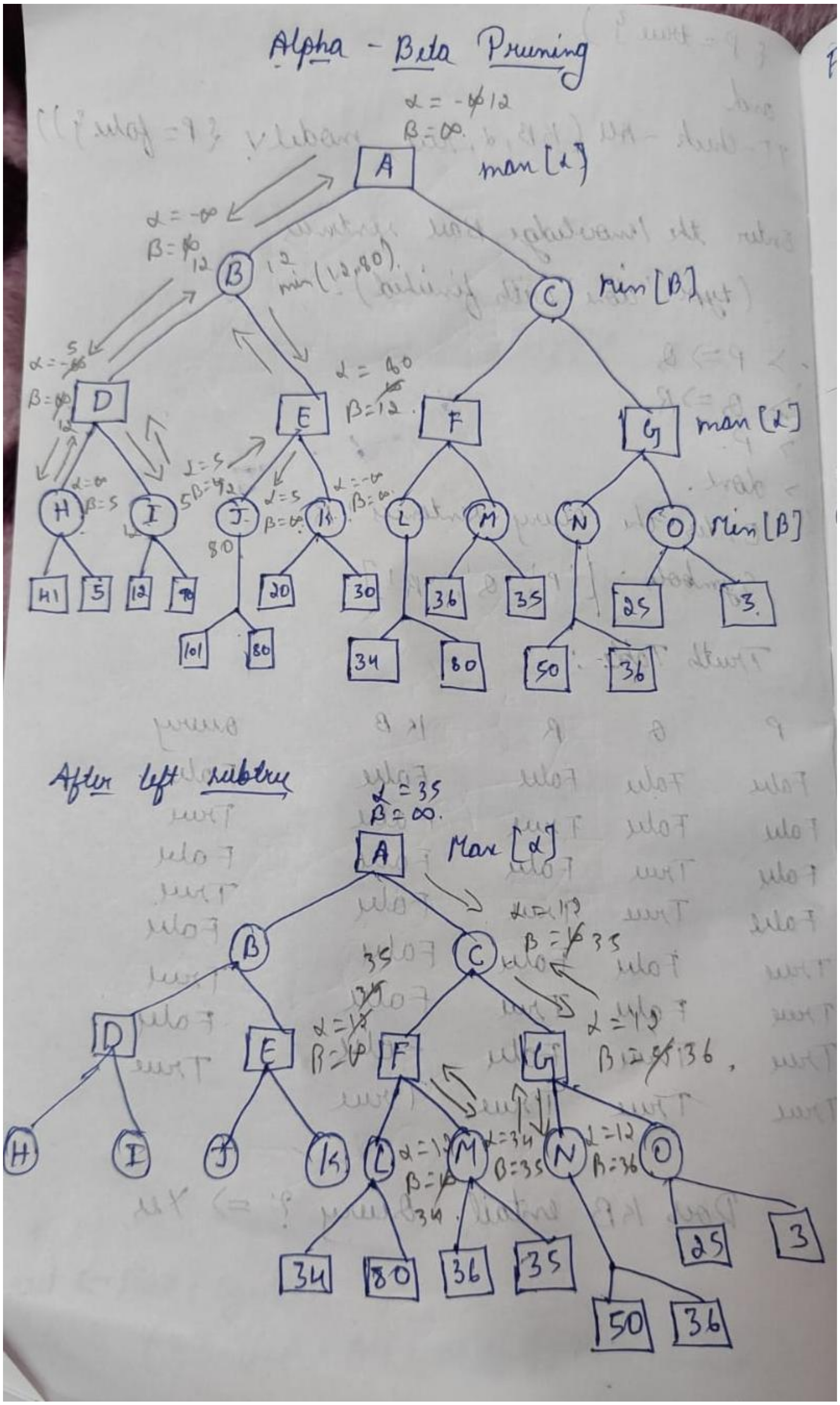
```
if clause not in clauses:
clauses.append(clause)

if resolved:
print('Knowledge Base entails the query, proved by resolution')
else:
print("Knowledge Base doesn't entail the query, no empty set produced after resolution") clauses
= input('Enter the clauses ').split()
query = input('Enter the query: ')
checkResolution(clauses, query)
```

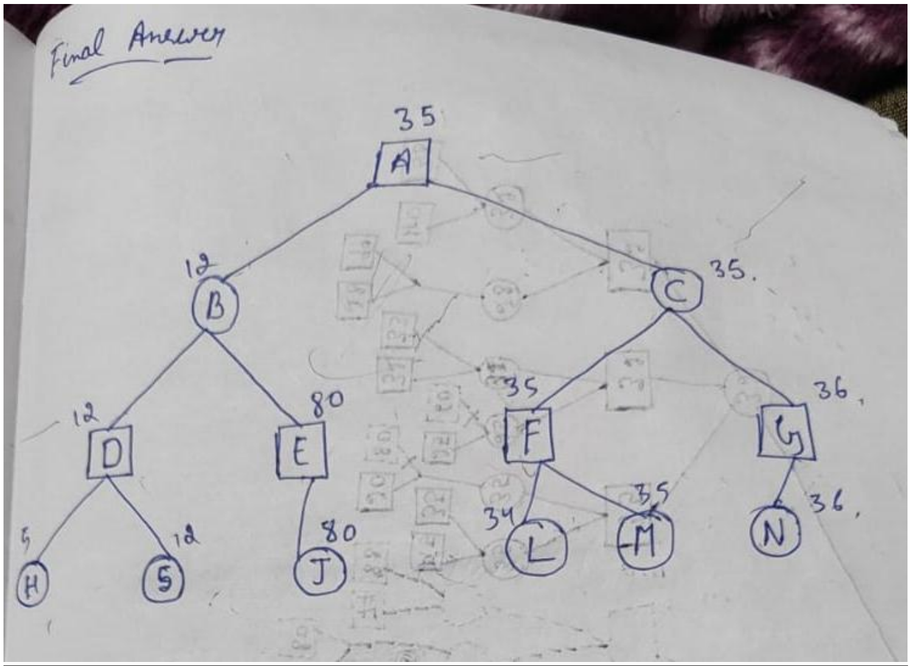
<u>Output Snapshot</u>

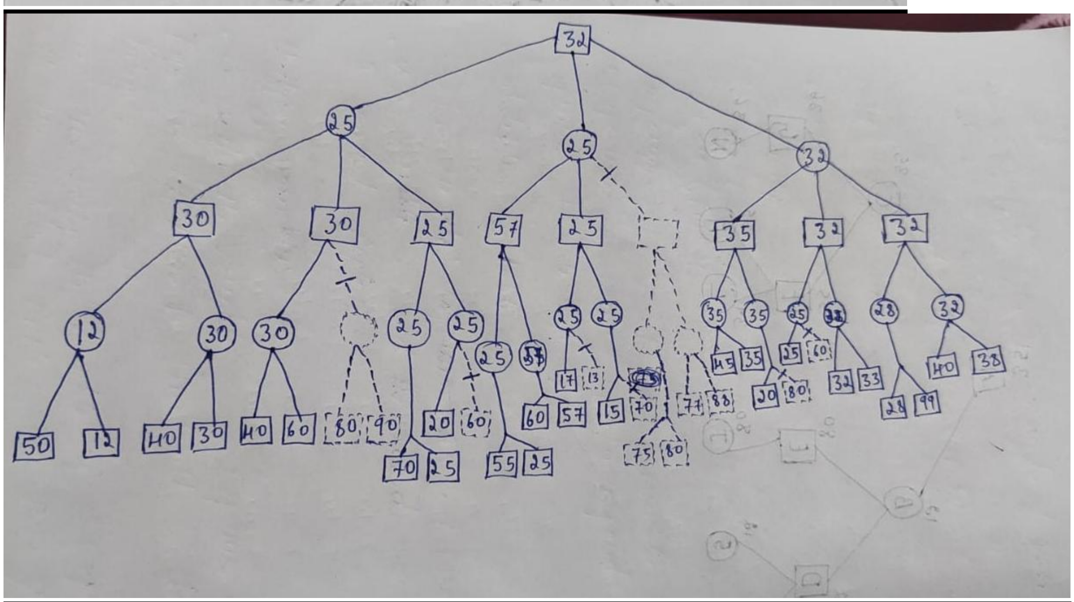
```
Enter the clauses (~qv~pvr)^(~q^p)^q
Enter the query: r
Trying to prove ((~qv~pvr)^(~q^p)^q)^(~r) by contradiction....
Knowledge Base entails the query, proved by resolution
```

Program-11

Alpha Beta Pruning<u>Algorithm :</u>







Code

```
import math
# Alpha-Beta Pruning Functions
def alpha_beta_search(state):
    """ Alpha-Beta Search to get the optimal action """
    value = max_value(state, -math.inf, math.inf)
    print("Optimal Value:", value)
    return value

def max_value(state, alpha, beta):
    """ Function to calculate the MAX value node """
    if terminal_test(state):  # If leaf node, return utility value
        return utility(state)
    v = -math.inf
    for child in state["children"]:  # Iterate through child nodes
```

```python
        v = max(v, min_value(child, alpha, beta))
        if v >= beta:
            return v  # Beta cutoff
        alpha = max(alpha, v)
    return v

def min_value(state, alpha, beta):
    """ Function to calculate the MIN value node """
    if terminal_test(state):  # If leaf node, return utility value
        return utility(state)
    v = math.inf
    for child in state["children"]:  # Iterate through child nodes
        v = min(v, max_value(child, alpha, beta))
        if v <= alpha:
            return v  # Alpha cutoff
        beta = min(beta, v)
    return v

# Utility Functions
def terminal_test(state):
    """ Check if the node is a leaf node """
    return "value" in state  # Leaf node if it contains 'value'

def utility(state):
    """ Return the utility value of a leaf node """
    return state["value"]

# Build the Binary Tree Based on Leaf Nodes
def build_tree(values):
    """ Recursively build a binary tree from a list of leaf node values """
    if len(values) == 1:  # Single value -> Leaf node
        return {"value": values[0]}
    mid = len(values) // 2
    left_subtree = build_tree(values[:mid])
    right_subtree = build_tree(values[mid:])
    return {"children": [left_subtree, right_subtree]}

# Main Program
if __name__ == "__main__":
    leaf_nodes = [10, 9, 14, 18, 5, 4, 50, 3]
    tree = build_tree(leaf_nodes)  # Build the binary tree
    print("Alpha-Beta Pruning Search:")
    alpha_beta_search(tree)
```

Output

```
Alpha-Beta Pruning Search:
Optimal Value: 10
```