VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Kathasagaram Aishwarya (1BM22CS123)

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

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

<u>CERTIFICATE</u>

This is to certify that the Lab work entitled “Artificial Intelligence (23CS5PCAIN)” carried out by **Kathasagaram Aishwarya (1BM22CS123),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

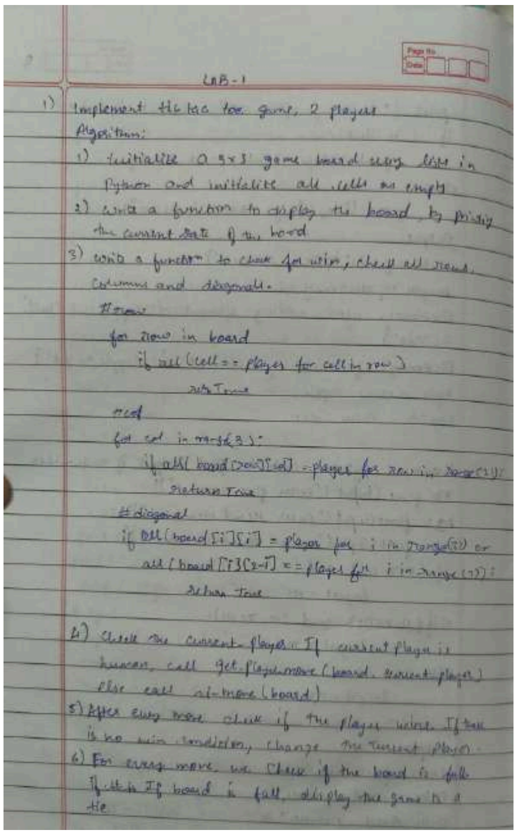
Dr. Pallavi G B Associate Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/Aish-kathasagaram/AI-Lab

WEEK 1:
Program 1
Implement Tic –Tac –Toe Game

Algorithm:


LAB-1

1) Implement tic tac toe game, 2 players
Algorithm:
1) Initialize a 3x3 game board using list in Python and initialize all cells as empty
2) Write a function to display the board by printing the current state of the board
3) Write a function to check for win, check all rows, columns and diagonals.
#rows
for row in board
if all (cell == player for cell in row)
return True
#col
for col in range(3):
if all(board[row][col] == player for row in range(3)):
return True
#diagonal
if all (board[i][i] == player for i in range(3)) or
all (board[i][2-i] == player for i in range(3)):
return True

4) Check the current player. If current player is human, call get_player_move (board, current_player) else call ai_move (board)
5) After every move check if the player wins. If there is no win condition, change the current player.
6) For every move we check if the board is full. If board is full, display the game is a tie

Code:

```
import numpy as np
import random

board = np.array([[-1, -1, -1], [-1, -1, -1], [-1, -1, -1]])

def check_win(b):
    for i in range(3):
        if (b[i, :] == 1).all() or (b[:, i] == 1).all():
            return 1
        if (b[i, :] == 0).all() or (b[:, i] == 0).all():
            return 0
    if (b[0, 0] == b[1, 1] == b[2, 2] == 1) or (b[0, 2] == b[1, 1] == b[2, 0] == 1):
        return 1
    if (b[0, 0] == b[1, 1] == b[2, 2] == 0) or (b[0, 2] == b[1, 1] == b[2, 0] == 0):
        return 0    return -1
```

```
def is_full(b):
    return not (-1 in b)

def print_board(b):
    symbols = {1: 'X', 0: 'O', -1: ' '}
    for row in b:
        print(" | ".join(symbols[cell] for cell in row))
        print("---------")

def player_move():
    while True:
        row = int(input("Enter row (0, 1, or 2): "))
        col = int(input("Enter column (0, 1, or 2): "))
        if((row<=2 and row >=0)and(col<=2 and col >=0)):
          if board[row, col] == -1:
              board[row, col] = 1
              break
          else:
              print("Cell is already occupied! Try again.")
        else:
          print("Invalid input,please try again")


def machine_move():
    empty_cells = [(r, c) for r in range(3) for c in range(3) if board[r, c] == -1]
    if empty_cells:
        row, col = random.choice(empty_cells)
        board[row, col] = 0
        print(f"Machine placed O at ({row}, {col})")

def play_game():
    print("Welcome to Tic Tac Toe!")
    print_board(board)
    for i in range(1, 10):
      if i % 2 !=0:
        print("Player 1's turn (X)")
        player_move()
        print_board(board)
        winner = check_win(board)
        if winner != -1:
            print(f"Player {winner} wins!")
            break
        if is_full(board):
            print("It's a draw!")
            break
      else:
```

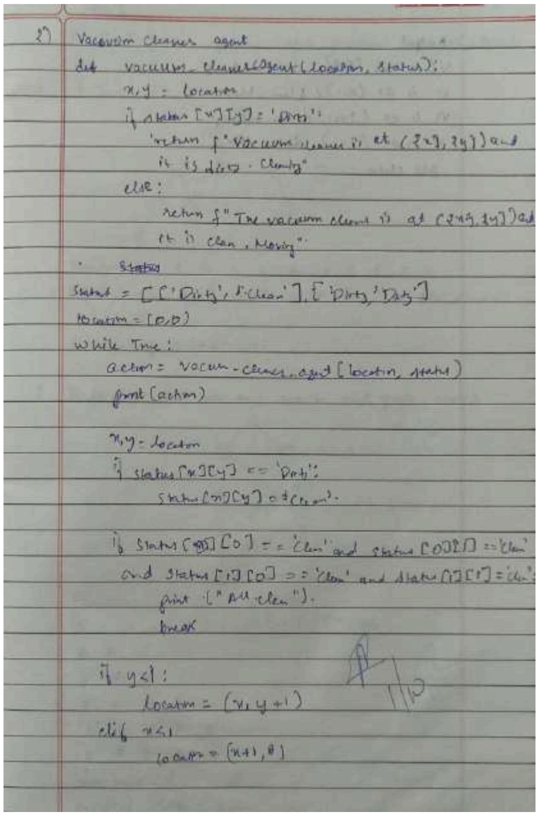
```
        print("Machine's turn (O)")
        machine_move()
        print_board(board)1
        winner = check_win(board)
        if winner != -1:
            print(f"Player {winner} wins!")
            break
        if is_full(board):
            print("It's a draw!")
            break
play_game()
```

Output:

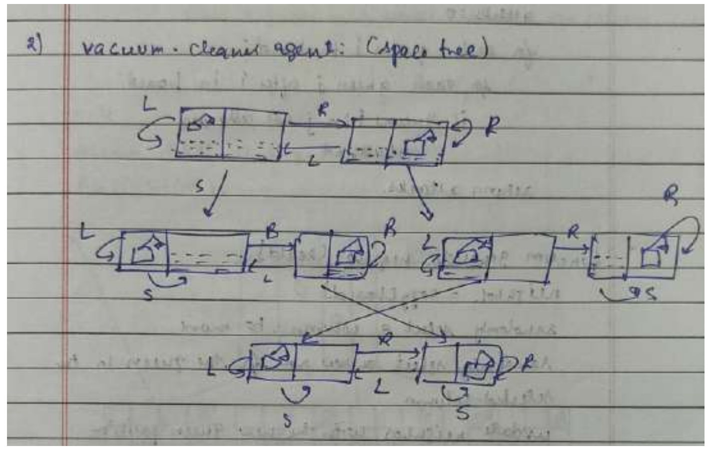
```
Welcome to Tic Tac Toe!
  |   |
---------
  |   |
---------
  |   |
---------
Player 1's turn (X)
Enter row (0, 1, or 2): 1
Enter column (0, 1, or 2): 1
  |   |
---------
  | X |
---------
  |   |
---------
Machine's turn (O)
Machine placed O at (2, 0)
  |   |
---------
  | X |
---------
O |   |
---------
Player 1's turn (X)
Enter row (0, 1, or 2): 2
Enter column (0, 1, or 2): 0
Cell is already occupied! Try again.
Enter row (0, 1, or 2): 2
Enter column (0, 1, or 2): 1
  |   |
---------
  | X |
---------
O | X |
---------
Machine's turn (O)
Machine placed O at (1, 0)
  |   |
---------
O | X |
---------
O | X |
---------
Player 1's turn (X)
Enter row (0, 1, or 2): 2
Enter column (0, 1, or 2): 2
  |   |
---------
O | X |
---------
O | X | X
---------
Machine's turn (O)
Machine placed O at (1, 2)
  |   |
---------
O | X | O
---------
O | X | X
---------
Player 1's turn (X)
Enter row (0, 1, or 2): 0
Enter column (0, 1, or 2): 2
  |   | X
---------
O | X | O
---------
O | X | X
---------
Machine's turn (O)
Machine placed O at (0, 0)
O |   | X
---------
O | X | O
---------
O | X | X
---------
Player 0 wins!
```

WEEK 1:
Program 2:
Implement vacuum cleaner agent

Algorithm:



```
2) Vacuum Cleaner agent
def vacuum_cleaner_agent(location, status):
    x,y = location
    if status[x][y] = 'Dirty':
        return f'Vacuum cleaner is at ({x},{y}) and
        it is dirty. Cleaning'
    else:
        return f"The vacuum cleaner is at ({x},{y}) and
        it is clean, Moving"
Status
status = [['Dirty','Clean'],['Dirty','Dirty']]
location = (0,0)
while True:
    action = vacuum_cleaner_agent(location, status)
    print(action)

    x,y = location
    if status[x][y] == 'Dirty':
        status[x][y] = 'Clean'

    if status[0][0] == 'Clean' and status[0][1] == 'Clean'
    and status[1][0] == 'Clean' and status[1][1] == 'Clean':
        print("All clean")
        break

    if y<1:
        location = (x, y+1)
    elif x<1
        location = (x+1, 0)
```



Code:
```
import random
en = input("Enter state (A/ B): ").strip().upper()
areaA = input("Enter current environment of area A (clean/ dirty): ").strip().lower()
areaB = input("Enter current environment of area B (clean/ dirty): ").strip().lower()
while areaA == 'dirty' or areaB == 'dirty':
    if en == 'A':
        if areaA == 'clean':
            print("Area A clean, moving to area B.")
            en = 'B'
        else:
            print("Area A dirty, cleaning and moving to area B.")
            areaA = 'clean'
            en = 'B'
    else:
        if areaB == 'clean':
            print("Area B clean, moving to area A.")
```

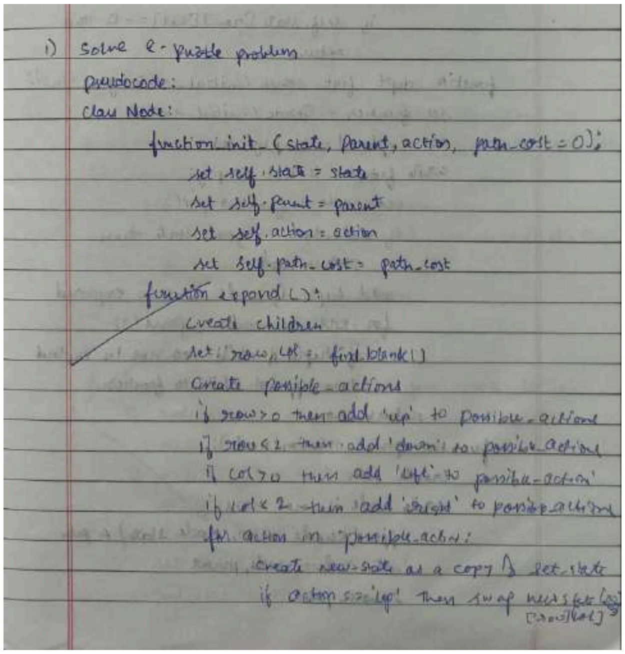
```
        en = 'A'
    else:
        print("Area B dirty, cleaning and moving to area A.")
        areaB = 'clean'
        en = 'A'
if areaA == 'clean' and areaB == 'clean':
    n = random.choice([0, 1])
    if n == 0:
        areaA = 'dirty'
    else:
        areaB = 'dirty'
```

Output:

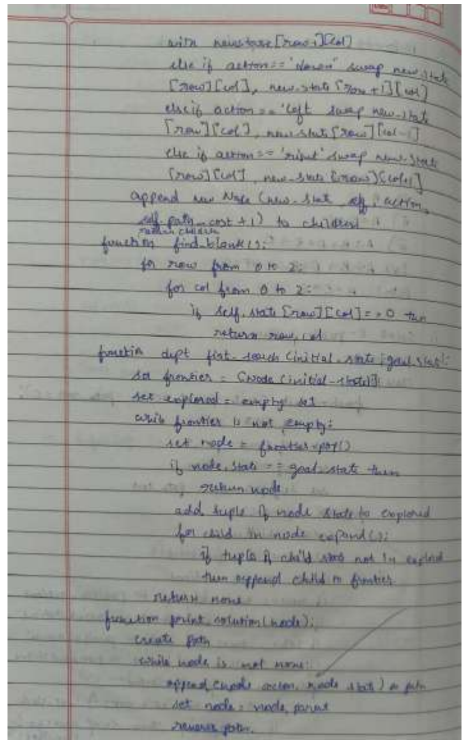
```
Area B dirty, cleaning and moving to area A.
Area A clean, moving to area B.
Area B dirty, cleaning and moving to area A.
Area A clean, moving to area B.
Area B dirty, cleaning and moving to area A.
Area A dirty, cleaning and moving to area B.
Area B clean, moving to area A.
Area A dirty, cleaning and moving to area B.
Area B clean, moving to area A.
Area A dirty, cleaning and moving to area B.
Area B clean, moving to area A.
Area A dirty, cleaning and moving to area B.
Area B dirty, cleaning and moving to area A.
Area A clean, moving to area B.
Area B dirty, cleaning and moving to area A.
Area A dirty, cleaning and moving to area B.
Area B clean, moving to area A.
Area A dirty, cleaning and moving to area B.
Area B clean, moving to area A.
Area A dirty, cleaning and moving to area B.
Area B dirty, cleaning and moving to area A.
Area A dirty, cleaning and moving to area B.
Area B dirty, cleaning and moving to area A.
Area A clean, moving to area B.
Area B dirty, cleaning and moving to area A.
Area A clean, moving to area B.
Area B dirty, cleaning and moving to area A.
Area A clean, moving to area B.
Area B dirty, cleaning and moving to area A.
Area A clean, moving to area B.
Area B dirty, cleaning and moving to area A.
Area A clean, moving to area B.
Area B dirty, cleaning and moving to area A.
Area A dirty, cleaning and moving to area B.
Area B clean, moving to area A.
Area A dirty, cleaning and moving to area B.
```

WEEK 2:
Program 3:
Implement 8 puzzle problems using Depth First Search (DFS)

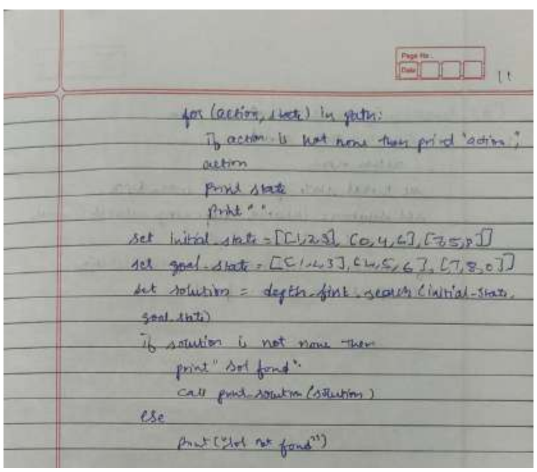
Algorithm:



1) Solve 8-puzzle problem
pseudocode:
class Node:
function init (state, parent, action, path_cost = 0):
set self.state = state
set self.parent = parent
set self.action = action
set self.path_cost = path_cost
function expand ():
create children
set row, col = find_blank()
create possible_actions
if row > 0 then add 'up' to possible_actions
if row < 2 then add 'down' to possible_actions
if col > 0 then add 'left' to possible_actions
if col < 2 then add 'right' to possible_actions
for action in possible_actions:
create new_state as a copy of self.state
if action == 'up' then swap new_state[row][col]



with new_state[row-1][col]
else if action == 'down' swap new_state[row][col], new_state[row+1][col]
else if action == 'left' swap new_state[row][col], new_state[row][col-1]
else if action == 'right' swap new_state[row][col], new_state[row][col+1]
append new Node (new_state, self, action, self.path_cost + 1) to children
return children
function find_blank():
for row from 0 to 2:
for col from 0 to 2:
if self.state[row][col] == 0 then
return row, col
function depth_first_search (initial_state, goal_state):
set frontier = [Node (initial_state)]
set explored = empty set
while frontier is not empty:
set node = frontier.pop()
if node.state == goal_state then
return node
add tuple of node state to explored
for child in node expand():
if tuple of child state not in explored
then append child to frontier
return none
function print_solution(node):
create path
while node is not none:
append (node action, node state) to path
set node = node.parent
reverse path



for (action, state) in path:
If action is not none then print 'action:', action
print state
print " "
set initial_state = [[1,2,3], [0,4,6], [7,5,8]]
set goal_state = [[1,2,3], [4,5,6], [7,8,0]]
set solution = depth_first_search (initial_state, goal_state)
if solution is not none then
print "Sol found"
call print_solution (solution)
else
print ("Sol not found")

Code:
```
from copy import deepcopy
goal_state = [[0, 1, 2],
              [3, 4, 5],
              [6, 7, 8]]
moves = {
    'up': (-1, 0),
    'down': (1, 0),
```

```
    'left': (0, -1),
    'right': (0, 1)
}
def find_blank(board):
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:
                return i, j
    return None

def is_goal(state):
    return state == goal_state

def is_valid_move(x, y):
    return 0 <= x < 3 and 0 <= y < 3

def apply_move(board, move):
    x, y = find_blank(board)
    dx, dy = moves[move]
    new_x, new_y = x + dx, y + dy
    if is_valid_move(new_x, new_y):
        new_board = deepcopy(board)
        new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
        return new_board
    return None
def dfs(start):
    stack = [(start, [])]
    visited = set()
    while stack:
        current_state, path = stack.pop()
        if is_goal(current_state):
            return path + [current_state]
        visited.add(tuple(tuple(row) for row in current_state))
        for move in moves:
            new_state = apply_move(current_state, move)
            if new_state and tuple(tuple(row) for row in new_state) not in visited:
                stack.append((new_state, path + [current_state]))
    return None

def print_board(board):
    for row in board:
        print(row)
    print()

def print_solution(solution):
```

```
    if solution:
        for board in solution:
            print_board(board)
    else:
        print("No solution found")

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

solution = dfs(initial_state)
print_solution(solution)
```

Output:

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

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

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

WEEK 2:
Program 4:
Implement 8 puzzle problems using Iterative deepening search algorithm (IDS)

Algorithm:

```
(2) Implement iterative deepening search algorithm
function iterative_deepening_search(initial_state, goal_state,
max_depth):
        for depth from 0 to max_depth:
            set results = depth_limited_search(initial_state,
            goal_state, depth)
            if result is not none then
            return result
        return none
function depth_limited_search(node, goal_state, limit):
    if node.state == goal_state then
        return node
    If node.depth >= limit then
        return none
    for each child in expand(node):
        set result = depth_limited_search(child,
        goal_state, limit)
            if result is not none then
                return result
        return none
set initial_state, goal_state, max_depth
set solution = iterative_deepening_search(initial
state, goal_state, max_depth)
if solution is not none then print solution
else print "no solution found"
```

Code:

```
from copy import deepcopy
goal_state = [[0, 1, 2],
              [3, 4, 5],
              [6, 7, 8]]
moves = {
    'up': (-1, 0),
    'down': (1, 0),
    'left': (0, -1),
    'right': (0, 1)
}
```

```
def find_blank(board):
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:
                return i, j
    return None

def is_goal(state):
    return state == goal_state

def is_valid_move(x, y):
    return 0 <= x < 3 and 0 <= y < 3

def apply_move(board, move):
    x, y = find_blank(board)
    dx, dy = moves[move]
    new_x, new_y = x + dx, y + dy
    if is_valid_move(new_x, new_y):
        new_board = deepcopy(board)
        new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y], new_board[x][y]
        return new_board
    return None

def dfs_limited(state, path, depth_limit, visited):
    if is_goal(state):
        return path + [state]
    if depth_limit == 0:
        return None
    visited.add(tuple(tuple(row) for row in state))
    for move in moves:
        new_state = apply_move(state, move)
        if new_state and tuple(tuple(row) for row in new_state) not in visited:
            result = dfs_limited(new_state, path + [state], depth_limit - 1, visited)
            if result:
                return result
    visited.remove(tuple(tuple(row) for row in state))
    return None

def ids(start):
    depth_limit = 0
    while True:
        visited = set()
        result = dfs_limited(start, [], depth_limit, visited)
        if result:
            return result
        depth_limit += 1
```

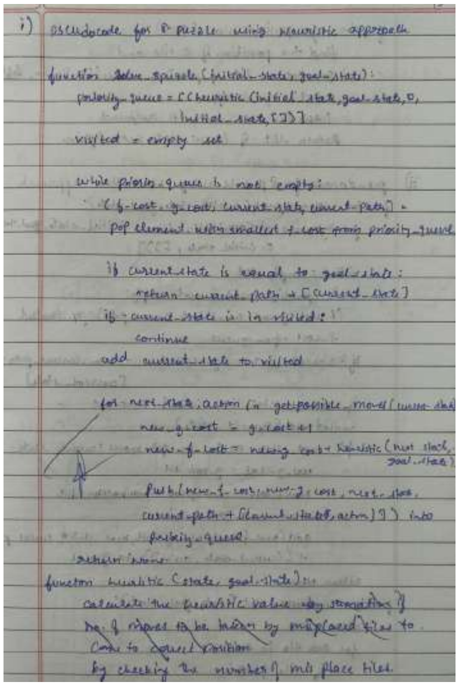
```
def print_board(board):
    for row in board:
        print(row)
    print()

def print_solution(solution):
    if solution:
        for board in solution:
            print_board(board)
    else:
        print("No solution found")

initial_state = [[1, 2, 5],
                 [0, 3, 8],
                 [6, 4, 7]]
solution = ids(initial_state)
print_solution(solution)
```

Output:

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

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

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

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

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

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

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

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

WEEK 3:
Program 5:
Implement A* search algorithm for 8 puzzle problem (based on number of misplaced tiles)

Algorithm:



i) Pseudocode for 8 puzzle using heuristic approach

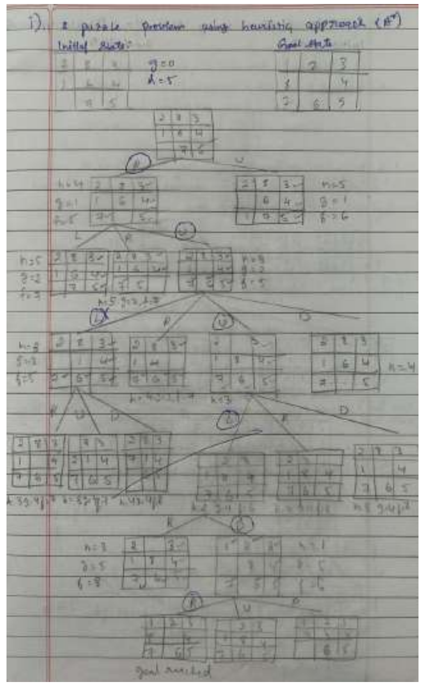
function solve_8puzzle(initial_state, goal_state):
    priority_queue = [(heuristic(initial_state, goal_state), 0, initial_state, [])]
    visited = empty set

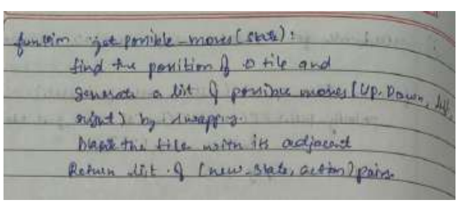
    while priority_queue is not empty:
        (f_cost, g_cost, current_state, current_path) = pop element with smallest f_cost from priority_queue

        if current_state is equal to goal_state:
            return current_path + [current_state]
        if current_state is in visited:
            continue
        add current_state to visited

        for next_state, action in get_possible_moves(current_state):
            new_g_cost = g_cost + 1
            new_f_cost = new_g_cost + heuristic(next_state, goal_state)

            push (new_f_cost, new_g_cost, next_state, current_path + [(current_state, action)]) into priority_queue
    return None
function heuristic(state, goal_state):
    calculate the heuristic value by number of moves to be taken by misplaced tiles to come to correct position
    by checking the number of misplaced tiles.





function get_possible_moves(state):
    find the position of 0 tile and generate a list of possible moves (Up, Down, left, right) by swapping.
    Swap the tile with its adjacent
    Return list of (new_state, action) pairs.

Code:
```
class Node:
    def __init__(self, data, level, fval):
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        x, y = self.find_blank()
        moves = [(x, y-1), (x, y+1), (x-1, y), (x+1, y)]
        children = []
        for new_x, new_y in moves:
            child_data = self.move_blank(x, y, new_x, new_y)
```

```
            if child_data:
                child_node = Node(child_data, self.level + 1, 0)
                children.append(child_node)
        return children

    def move_blank(self, x1, y1, x2, y2):
        if 0 <= x2 < len(self.data) and 0 <= y2 < len(self.data[0]):
            new_data = [row[:] for row in self.data]
            new_data[x1][y1], new_data[x2][y2] = new_data[x2][y2], new_data[x1][y1]
            return new_data
        return None

    def find_blank(self):
        for i, row in enumerate(self.data):
            if '_' in row:
                return i, row.index('_')
class Puzzle:
    def __init__(self, size):
        self.size = size
        self.open = []
        self.closed = []

    def get_input(self):
        return [input().split() for _ in range(self.size)]

    def f(self, start, goal):
        return start.level + self.h(start.data, goal)

    def h(self, start_data, goal):

        return sum(start_data[i][j] != goal[i][j] and start_data[i][j] != '_' for i in range(self.size) for j in
range(self.size))

    def process(self):
        print("Enter the start state matrix:")
        start_data = self.get_input()
        print("Enter the goal state matrix:")
        goal = self.get_input()
        start_node = Node(start_data, 0, 0)
        start_node.fval = self.f(start_node, goal)
        self.open.append(start_node)
        while self.open:
            current_node = self.open.pop(0)
            self.display_state(current_node, goal)
            if self.h(current_node.data, goal) == 0:
                print("Goal reached!")
```

```
            break
        children = current_node.generate_child()
        for child in children:
            child.fval = self.f(child, goal)
            self.open.append(child)
        self.closed.append(current_node)
        self.open.sort(key=lambda node: node.fval)

    def display_state(self, node, goal):
        print("\nNext step:")
        for row in node.data:
            print(" ".join(row))
        heuristic_value = self.h(node.data, goal)
        print(f"Heuristic (h): {heuristic_value}")
        print(f"Function value (f = g + h): {node.fval}")

puz = Puzzle(3)
puz.process()
```

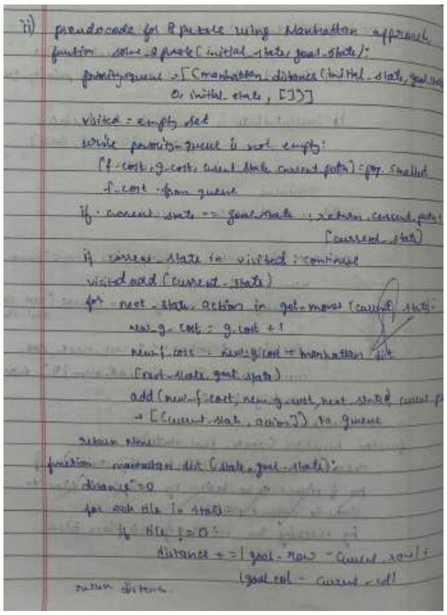
Output:

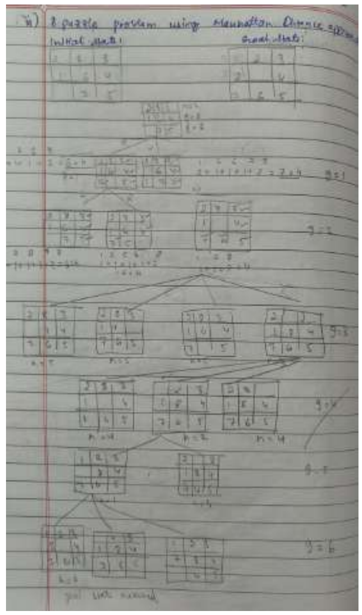
```
Enter the start state matrix:
2 8 3
1 6 4
7 _ 5
Enter the goal state matrix:
1 2 3
8 _ 4
7 6 5

Next step:
2 8 3
1 6 4
7 _ 5
Heuristic (h): 4
Function value (f = g + h): 4

Next step:
2 8 3
1 _ 4
7 6 5
Heuristic (h): 3
Function value (f = g + h): 4

Next step:
2 8 3
_ 1 4
7 6 5
Heuristic (h): 3
Function value (f = g + h): 5

Next step:
2 _ 3
1 8 4
7 6 5
Heuristic (h): 3
Function value (f = g + h): 5
```

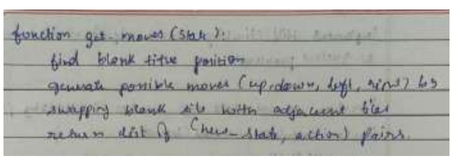
```
Next step:
_ 2 3
1 8 4
7 6 5
Heuristic (h): 2
Function value (f = g + h): 5

Next step:
1 2 3
_ 8 4
7 6 5
Heuristic (h): 1
Function value (f = g + h): 5

Next step:
1 2 3
8 _ 4
7 6 5
Heuristic (h): 0
Function value (f = g + h): 5
Goal reached!
```

WEEK 4:
Program 6:
Implement A* search algorithm for 8 puzzle problem (Manhattan Distance method)

Algorithm:


ii) pseudocode for 8 puzzle using Manhattan approach
function solve_8puzzle(initial_state, goal_state):
priority_queue = [(manhattan_distance(initial_state, goal_state), 0, initial_state, [])]
visited = empty set
while priority_queue is not empty:
(f_cost, g_cost, current_state, current_path) = pop smallest f_cost from queue
if current_state == goal_state : return current_path + [current_state]
if current_state in visited : continue
visited.add(current_state)
for next_state, action in get_moves(current_state):
new_g_cost = g_cost + 1
new_f_cost = new_g_cost + manhattan_dist(next_state, goal_state)
add (new_f_cost, new_g_cost, next_state, current_path + [(current_state, action)]) to queue
return None
function manhattan_dist(state, goal_state):
distance = 0
for each tile in state:
if tile != 0:
distance += |goal_row - current_row| + |goal_col - current_col|
return distance




function get_moves(state):
find blank tile position
generate possible moves (up, down, left, right) by swapping blank tile with adjacent tiles
return list of (new_state, action) pairs.

Code:
```
class Node:
    def __init__(self, data, level, fval):
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        x, y = self.find_blank()
        moves = [(x, y-1), (x, y+1), (x-1, y), (x+1, y)]
        children = []
        for new_x, new_y in moves:
            child_data = self.move_blank(x, y, new_x, new_y)
            if child_data:
                child_node = Node(child_data, self.level + 1, 0)
                children.append(child_node)
        return children
```

```python
    def move_blank(self, x1, y1, x2, y2):
        if 0 <= x2 < len(self.data) and 0 <= y2 < len(self.data[0]):
            new_data = [row[:] for row in self.data]
            new_data[x1][y1], new_data[x2][y2] = new_data[x2][y2], new_data[x1][y1]
            return new_data
        return None

    def find_blank(self):
        for i, row in enumerate(self.data):
            if '_' in row:
                return i, row.index('_')

class Puzzle:
    def __init__(self, size):
        self.size = size
        self.open = []
        self.closed = []

    def get_input(self):
        return [input().split() for _ in range(self.size)]

    def f(self, start, goal):
        h_val = self.manhattan_heuristic(start.data, goal)
        return start.level + h_val, h_val

    def manhattan_heuristic(self, start_data, goal):
        distance = 0
        for i in range(self.size):
            for j in range(self.size):
                if start_data[i][j] != '_' and start_data[i][j] != goal[i][j]:
                    goal_x, goal_y = self.find_position(goal, start_data[i][j])
                    distance += abs(i - goal_x) + abs(j - goal_y)
        return distance

    def find_position(self, state, value):
        for i in range(self.size):
            for j in range(self.size):
                if state[i][j] == value:
                    return i, j

    def process(self):
        print("Enter the start state matrix:")
        start_data = self.get_input()
        print("Enter the goal state matrix:")
        goal = self.get_input()
```

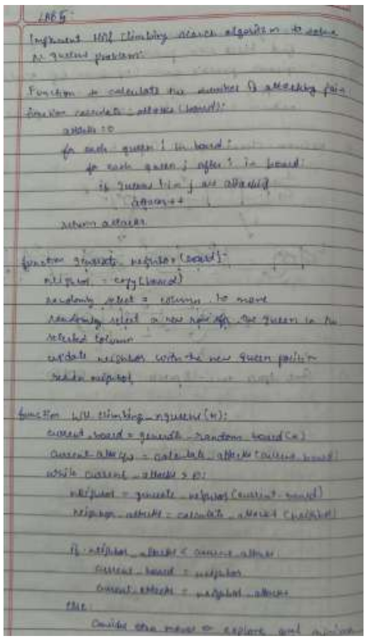
```
        start_node = Node(start_data, 0, 0)
        start_node.fval, h_val = self.f(start_node, goal)
        self.open.append(start_node)
        while self.open:
            current_node = self.open.pop(0)
            self.display_state(current_node.data, current_node.fval, h_val, current_node.level)
            if self.manhattan_heuristic(current_node.data, goal) == 0:
                print("Goal reached!")
                break
            children = current_node.generate_child()
            for child in children:
                child.fval, h_val = self.f(child, goal)
                self.open.append(child)
            self.closed.append(current_node)
            self.open.sort(key=lambda node: node.fval)

    def display_state(self, data, f_val, h_val, g_val):
        print("\nNext step:")
        for row in data:
            print(" ".join(row))
        print(f"f(x) = {f_val} (g(x) = {g_val}, h(x) = {h_val})")

puz = Puzzle(3)
puz.process()
```

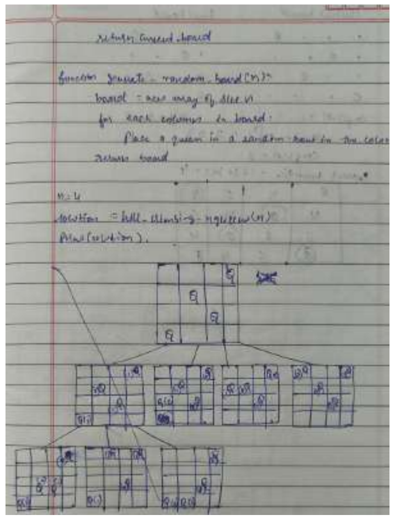
Output:

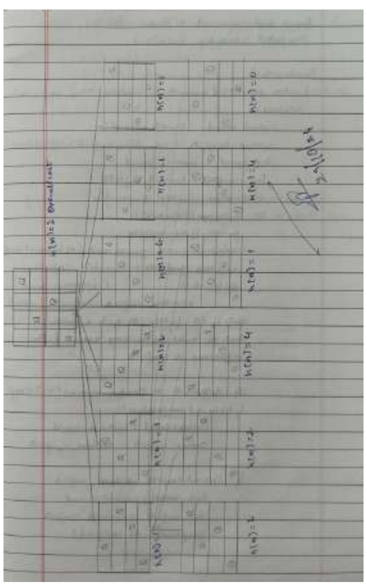
```
Enter the start state matrix:
2 8 3
1 6 4
_ 7 5
Enter the goal state matrix:
1 2 3
8 _ 4
7 6 5

Next step:
2 8 3
1 6 4
_ 7 5
f(x) = 6 (g(x) = 0, h(x) = 6)

Next step:
2 8 3
1 6 4
7 _ 5
f(x) = 6 (g(x) = 1, h(x) = 7)

Next step:
2 8 3
1 _ 4
7 6 5
f(x) = 6 (g(x) = 2, h(x) = 4)

Next step:
2 _ 3
1 8 4
7 6 5
f(x) = 6 (g(x) = 3, h(x) = 5)

Next step:
_ 2 3
1 8 4
7 6 5
f(x) = 6 (g(x) = 4, h(x) = 4)

Next step:
1 2 3
_ 8 4
7 6 5
f(x) = 6 (g(x) = 5, h(x) = 1)

Next step:
1 2 3
8 _ 4
7 6 5
f(x) = 6 (g(x) = 6, h(x) = 2)
Goal reached!
```

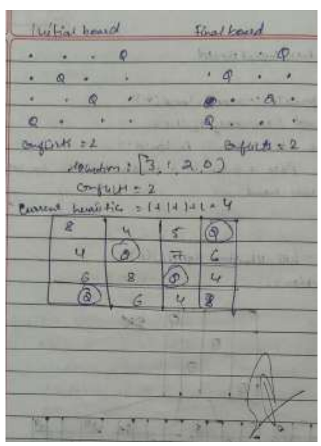
WEEK 5:
Program 7:
Implement Hill Climbing search algorithm to solve N-Queens problem.

Algorithm:









Code:

```
import random

N = 4
def calculateCost(state):
    attacking_pairs = 0
```

```
    for i in range(N):
        for j in range(i + 1, N):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                attacking_pairs += 1
    return attacking_pairs

def getNeighbours(state):
    neighbours = []
    for i in range(N):
        for j in range(i + 1, N):
            new_state = state[:]
            new_state[i], new_state[j] = new_state[j], new_state[i]
            neighbours.append(new_state)
    return neighbours

def hillClimbing(initial_state):
    current_state = initial_state
    current_cost = calculateCost(current_state)
    iteration = 0
    while True:
        print(f"\nIteration {iteration}")
        print(f"Current State: {current_state}, Cost: {current_cost}")
        neighbours = getNeighbours(current_state)
        next_state = current_state
        next_cost = current_cost
        for neighbour in neighbours:
            cost = calculateCost(neighbour)
            print(f"Neighbour: {neighbour}, Cost: {cost}")
            if cost < next_cost:
                next_state = neighbour
                next_cost = cost
        if next_cost == current_cost:
            break
        else:
            current_state, current_cost = next_state, next_cost
        if current_cost == 0:
            break
        iteration += 1
    return current_state, current_cost

initial_state = list(map(int, input("Enter initial state as 4 space-separated integers (0 to 3): ").split()))
solution_state, solution_cost = hillClimbing(initial_state)
print("\nFinal Results")
print("Initial State:", initial_state)
print("Final State (Solution):", solution_state)
print("Final Cost (Attacking Pairs):", solution_cost)
```

```
if solution_cost == 0:
    print("Solution found!")
else:
    print("Local optimum reached, but no solution.")
```

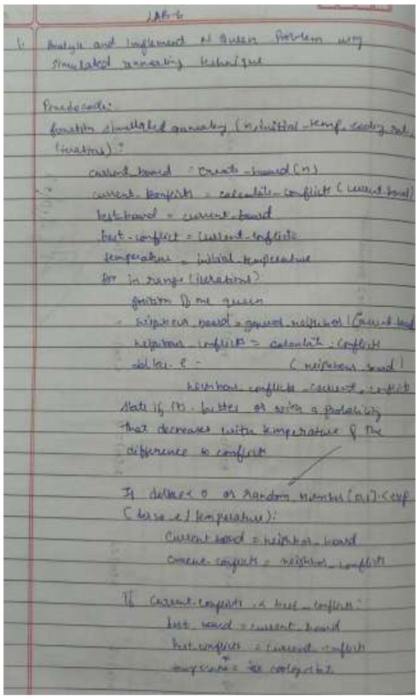
Output:

```
Enter initial state as 4 space-separated integers (0 to 3): 3 1 0 2

Iteration 0
Current State: [3, 1, 0, 2], Cost: 1
Neighbour: [1, 3, 0, 2], Cost: 0
Neighbour: [0, 1, 3, 2], Cost: 2
Neighbour: [2, 1, 0, 3], Cost: 4
Neighbour: [3, 0, 1, 2], Cost: 4
Neighbour: [3, 2, 0, 1], Cost: 2
Neighbour: [3, 1, 2, 0], Cost: 2

Final Results
Initial State: [3, 1, 0, 2]
Final State (Solution): [1, 3, 0, 2]
Final Cost (Attacking Pairs): 0
Solution found!
```

```
Enter initial state as 4 space-separated integers (0 to 3): 0 1 0 0

Iteration 0
Current State: [0, 1, 0, 0], Cost: 5
Neighbour: [1, 0, 0, 0], Cost: 4
Neighbour: [0, 1, 0, 0], Cost: 5
Neighbour: [0, 1, 0, 0], Cost: 5
Neighbour: [0, 0, 1, 0], Cost: 5
Neighbour: [0, 0, 0, 1], Cost: 4
Neighbour: [0, 1, 0, 0], Cost: 5

Iteration 1
Current State: [1, 0, 0, 0], Cost: 4
Neighbour: [0, 1, 0, 0], Cost: 5
Neighbour: [0, 0, 1, 0], Cost: 5
Neighbour: [0, 0, 0, 1], Cost: 4
Neighbour: [1, 0, 0, 0], Cost: 4
Neighbour: [1, 0, 0, 0], Cost: 4
Neighbour: [1, 0, 0, 0], Cost: 4

Final Results
Initial State: [0, 1, 0, 0]
Final State (Solution): [1, 0, 0, 0]
Final Cost (Attacking Pairs): 4
Local optimum reached, but no solution.
```

WEEK 6:
Program 8:
Simulated Annealing to Solve 8-Queens problem.

Algorithm:



LAB-6

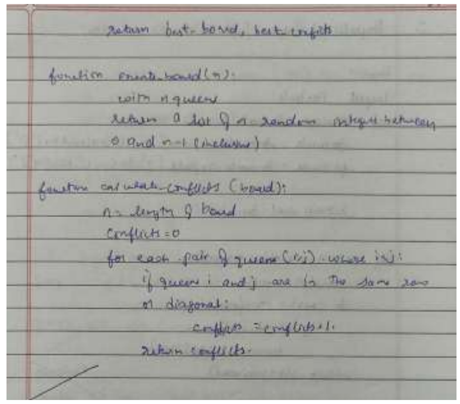
1. Analyse and implement N Queen Problem using Simulated annealing technique

Pseudocode:

function simulated annealing (n, initial_temp, cooling_rate, iterations):
    current_board = create_board(n)
    current_conflicts = calculate_conflicts(current_board)
    best_board = current_board
    best_conflict = current_conflicts
    temperature = initial_temperature
    for in range(iterations)
        position of one queen
        neighbour_board = generate_neighbour(current_board)
        neighbour_conflict = calculate_conflict(neighbour_board)
        delta_e = neighbour_conflicts - current_conflicts
    state if its better or with a probability that decreases with temperature & the difference in conflicts

    If delta_e < 0 or random_number(0,1) < exp(-delta_e / temperature):
        current_board = neighbour_board
        current_conflicts = neighbour_conflicts

    If current_conflicts < best_conflicts:
        best_board = current_board
        best_conflicts = current_conflicts
        temperature *= cooling_rate



return best_board, best_conflicts

function create_board(n):
    with n queens
    return a list of n random integer between 0 and n-1 (inclusive)

function calculate_conflicts (board):
    n = length of board
    conflicts = 0
    for each pair of queens (i, j) where i < j:
        if queens i and j are in the same row or diagonal:
            conflicts = conflicts + 1
        return conflicts.

Code:

```
import random
import copy
import math

class CheckeredPageState:
    def __init__(self, board):
        self.board = board
        self.dimension = len(board)
        self.h = self.calculateHeuristic()

    def calculateHeuristic(self):
        h = 0
        for i in range(self.dimension):
            for j in range(i + 1, self.dimension):
                if self.board[i] == self.board[j] or abs(self.board[i] - self.board[j]) == j - i:
                    h += 1
```

```
        return h

    def randomSuccessor(self):
        new_board = copy.deepcopy(self.board)
        row = random.randint(0, self.dimension - 1)
        new_col = random.randint(0, self.dimension - 1)
        new_board[row] = new_col
        return CheckeredPageState(new_board)

    def getMove(self, next_state):
        self.board = next_state.board
        self.h = next_state.h

    def printPage(self):
        for i in range(self.dimension):
            row = ['Q' if j == self.board[i] else '.' for j in range(self.dimension)]
            print(" ".join(row))
        print()

def SimulatedAnnealing(checkeredPageInitial, T=4000, tChange=0.8):
    current = CheckeredPageState(checkeredPageInitial)
    print("start of simulated annealing algorithm")
    while True:
        print("current state checkered page:")
        current.printPage()
        print("current state h:", current.h)
        T *= tChange
        if T < 1:
            print("final state checkered page:")
            current.printPage()
            print("final state h:", current.h)
            if current.h == 0:
                print("the simulated annealing found a solution")
                return True, current
            else:
                print("the simulated annealing could not find the solution")
                return False, current
        next = current.randomSuccessor()
        deltaE = current.h - next.h
        if deltaE > 0:
            print("Better solution found, moving to next state.")
            current.getMove(next)
            current = next
        else:
            rand = random.uniform(0, 1)
            probability = math.exp(deltaE / T)
```

```
            print(f"Probability of accepting worse solution: {probability:.4f}")
            if rand <= probability:
                print("Accepted worse solution based on probability.")
                current.getMove(next)
                current = next
            else:
                print("Rejected worse solution based on probability.")
def getUserInputBoard(dimension):
    print(f"Enter the initial positions of queens (0 to {dimension-1}) for each
row:")
    board = []
    for i in range(dimension):
        while True:
            try:
                position = int(input(f"Row {i + 1} (Enter column position
0-{dimension - 1}): "))
                if position < 0 or position >= dimension:
                    print(f"Invalid position. Please enter a number between 0 and
{dimension - 1}.")
                else:
                    board.append(position)
                    break
            except ValueError:
                print("Invalid input. Please enter an integer.")
    return board

def main():
    dimension = int(input("Enter the dimension of the board (e.g., 8 for 8x8):
"))
    initial_checkered_page = getUserInputBoard(dimension)

    print("Initial checkered page configuration:")
    for i in range(dimension):
        print("Row", i + 1, ":", '.' * initial_checkered_page[i] + 'Q' + '.' *
(dimension - initial_checkered_page[i] - 1))

    solution_found, final_state = SimulatedAnnealing(initial_checkered_page)

    if solution_found:
        print("\nSimulated Annealing found a solution:")
    else:
        print("\nSimulated Annealing did not find a solution:")
    final_state.printPage()
    print("Final heuristic (number of attacking pairs):", final_state.h)

main()
```

Output:

```
Enter the dimension of the board (e.g., 8 for 8x8): 8
Enter the initial positions of queens (0 to 7) for each row:
Row 1 (Enter column position 0-7): 0
Row 2 (Enter column position 0-7): 3
Row 3 (Enter column position 0-7): 7
Row 4 (Enter column position 0-7): 5
Row 5 (Enter column position 0-7): 1
Row 6 (Enter column position 0-7): 4
Row 7 (Enter column position 0-7): 2
Row 8 (Enter column position 0-7): 6
Initial checkered page configuration:
Row 1 : Q.......
Row 2 : ...Q....
Row 3 : .......Q
Row 4 : .....Q..
Row 5 : .Q......
Row 6 : ....Q...
Row 7 : ..Q.....
Row 8 : ......Q.
start of simulated annealing algorithm
current state checkered page:
Q . . . . . . .
. . . Q . . . .
. . . . . . . Q
. . . . . Q . .
. Q . . . . . .
. . . . Q . . .
. . Q . . . . .
. . . . . . Q .

current state h: 4
Probability of accepting worse solution: 0.9991
Accepted worse solution based on probability.
current state checkered page:
Q . . . . . . .
. . . Q . . . .
. . . . . . . Q
. . . . . Q . .
. . . Q . . . .
. . . . Q . . .
. . Q . . . . .
. . . . . . Q .

current state h: 7
Probability of accepting worse solution: 0.9992
Accepted worse solution based on probability.
current state checkered page:
Q . . . . . . .
. . . Q . . . .
. . . . . . Q .
. . . . . Q . .
. . . Q . . . .
. . . . Q . . .
. . Q . . . . .
. . . . . . Q .

current state h: 9
Probability of accepting worse solution: 1.0000
Accepted worse solution based on probability.
current state checkered page:
Q . . . . . . .
. . . Q . . . .
. . . . . . Q .
. . . . . Q . .
. . . Q . . . .
. . . . Q . . .
. . Q . . . . .
. Q . . . . . .

current state h: 9
Better solution found, moving to next state.
current state checkered page:
Q . . . . . . .
. . . Q . . . .
. . . . . . Q .
. . . . . Q . .
. . . Q . . . .
. . . . Q . . .
Q . . . . . . .
. Q . . . . . .

current state h: 8
Probability of accepting worse solution: 0.9992
Accepted worse solution based on probability.
current state checkered page:
. . . Q . . . .
. . . Q . . . .
. . . . . . Q .
. . . . . Q . .
. . . Q . . . .
. . . . Q . . .
Q . . . . . . .
. Q . . . . . .

current state h: 9
Better solution found, moving to next state.
current state checkered page:
. Q . . . . . .
. . . Q . . . .
. . . . . . Q .
. . . . . Q . .
. . . Q . . . .
. . . . Q . . .
Q . . . . . . .
. Q . . . . . .

current state h: 8
Better solution found, moving to next state.
current state checkered page:
. Q . . . . . .
. . . Q . . . .
. . . . . . Q .
. . . . . Q . .
. . . Q . . . .
. . . . Q . . .
Q . . . . . . .
. . . . . . . Q

current state h: 4
Better solution found, moving to next state.
current state checkered page:
. Q . . . . . .
. . . Q . . . .
. . . . . . Q .
. . . . . Q . .
. . . . . . . Q
. . . . Q . . .
Q . . . . . . .
. . . . . . . Q

current state h: 3
Probability of accepting worse solution: 1.0000
Accepted worse solution based on probability.
current state checkered page:
. Q . . . . . .
. . . . . . Q .
. . . . . . Q .
. . . . . Q . .
. . . . . . . Q
. . . . Q . . .
Q . . . . . . .
. . . . . . . Q

current state h: 3
Probability of accepting worse solution: 0.9977
Accepted worse solution based on probability.
current state checkered page:
. Q . . . . . .
. . . . . . Q .
. . . . . . Q .
. . . . . Q . .
. . . . . . . Q
. . . . Q . . .
. . . Q . . . .
. . . . . . . Q

current state h: 4
Probability of accepting worse solution: 0.9942
Accepted worse solution based on probability.
current state checkered page:
. . . . . . Q .
. . . . . . Q .
. . . . . . Q .
. . . . . Q . .
. . . . . . . Q
. . . . Q . . .
. . . Q . . . .
. . . . . . . Q

current state h: 6
Probability of accepting worse solution: 1.0000
Accepted worse solution based on probability.
current state checkered page:
. . . . . . Q .
. . . . . . Q .
. . . . . . Q .
Q . . . . . . .
. . . . . . . Q
. . . . Q . . .
. . . Q . . . .
. . . . . . . Q

current state h: 6
Probability of accepting worse solution: 0.9909
Accepted worse solution based on probability.
current state checkered page:
. . . . . . Q .
. . . . . . Q .
. . . . . . Q .
Q . . . . . . .
. . . . . . . Q
. . . . Q . . .
. . . . . . Q .
. . . . . . . Q

current state h: 8
Probability of accepting worse solution: 0.9887
Accepted worse solution based on probability.
current state checkered page:
. . . . . . Q .
. . . . . . Q .
. . . . . . Q .
Q . . . . . . .
. . . . . . . Q
. . Q . . . . .
. . . . . . Q .
. . . . . . . Q

current state h: 10
Probability of accepting worse solution: 1.0000
Accepted worse solution based on probability.
current state checkered page:
. . . . . . Q .
. . . . . . Q .
. . . . . . Q .
Q . . . . . . .
Q . . . . . . .
. . Q . . . . .
. . . . . . Q .
. . . . . . . Q

current state h: 10
Probability of accepting worse solution: 1.0000
Accepted worse solution based on probability.
current state checkered page:
. . . . . . Q .
. . . . . . Q .
. . . . . . Q .
Q . . . . . . .
Q . . . . . . .
. Q . . . . . .
. . . . . . Q .
. . . . . . . Q

current state h: 10
Better solution found, moving to next state.
current state checkered page:
. . . . . . Q .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
Q . . . . . . .
. Q . . . . . .
. . . . . . Q .
. . . . . . . Q
```

```
current state h: 7
Better solution found, moving to next state.
current state checkered page:
. . . . . . Q .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
Q . . . . . . .
. . Q . . . . .
. . . . . . Q .
. . . . . . . Q

current state h: 6
Probability of accepting worse solution: 0.9659
Accepted worse solution based on probability.
current state checkered page:
. . . . . . Q .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
. . . . . . Q .
. . . . . . . Q

current state h: 8
Probability of accepting worse solution: 0.9576
Rejected worse solution based on probability.
current state checkered page:
. . . . . . Q .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
. . . . . . Q .
. . . . . . . Q

current state h: 8
Probability of accepting worse solution: 1.0000
Accepted worse solution based on probability.
current state checkered page:
. . . . . . Q .
. . . . Q . . .
. . . Q . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
. . . . . . Q .
. . . . . . . Q
```

```
current state h: 8
Probability of accepting worse solution: 1.0000
Accepted worse solution based on probability.
current state checkered page:
. . . . . . Q .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
. . . . . . Q .
. . . . . . . Q

current state h: 8
Better solution found, moving to next state.
current state checkered page:
. . . . . . Q .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
. Q . . . . . .
. . . . . . . Q

current state h: 6
Better solution found, moving to next state.
current state checkered page:
. . . . . . Q .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
. . . . . Q . .
. . . . . . . Q

current state h: 5
Probability of accepting worse solution: 0.9360
Accepted worse solution based on probability.
current state checkered page:
. . . . . . Q .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
. . . . . Q . .
. . Q . . . . .
```

```
current state h: 6
Probability of accepting worse solution: 0.9206
Accepted worse solution based on probability.
current state checkered page:
. . . . . . Q .
. . . . Q . . .
. Q . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
. . . . . Q . .
. . Q . . . . .

current state h: 7
Probability of accepting worse solution: 1.0000
Accepted worse solution based on probability.
current state checkered page:
. . . . . . Q .
. . . . Q . . .
. Q . . . . . .
Q . . . . . . .
. . Q . . . . .
Q . . . . . . .
. . . . . Q . .
. . Q . . . . .

current state h: 7
Probability of accepting worse solution: 0.7722
Accepted worse solution based on probability.
current state checkered page:
. . . . . . Q .
. . . . Q . . .
. Q . . . . . .
Q . . . . . . .
. Q . . . . . .
Q . . . . . . .
. . . . . Q . .
. . Q . . . . .

current state h: 9
Better solution found, moving to next state.
current state checkered page:
. . . . . . Q .
. . . . Q . . .
. Q . . . . . .
. . . . . Q . .
. Q . . . . . .
Q . . . . . . .
. . . . . Q . .
. . Q . . . . .
```

```
current state h: 5
final state checkered page:
. . . . . . Q .
. . . . Q . . .
. Q . . . . . .
. . . . . Q . .
. . . . . . . Q
. . Q . . . . .
. . . . . Q . .
. . Q . . . . .

final state h: 5
the simulated annealing could not find the solution

Simulated Annealing did not find a solution:
. . . . . . Q .
. . . . Q . . .
. Q . . . . . .
. . . . . Q . .
. . . . . . . Q
. . Q . . . . .
. . . . . Q . .
. . Q . . . . .

Final heuristic (number of attacking pairs): 5
```

```
current state h: 7
Better solution found, moving to next state.
current state checkered page:
. . . . . . Q .
. . . . Q . . .
. Q . . . . . .
. . . . . Q . .
. . . . . . Q .
Q . . . . . . .
. . . . . Q . .
. . Q . . . . .

current state h: 6
Probability of accepting worse solution: 0.4689
Rejected worse solution based on probability.
current state checkered page:
. . . . . . Q .
. . . . Q . . .
. Q . . . . . .
. . . . . Q . .
. . . . . . Q .
Q . . . . . . .
. . . . . Q . .
. . Q . . . . .

current state h: 6
Better solution found, moving to next state.
current state checkered page:
. . . . . . Q .
. . . . Q . . .
. Q . . . . . .
. . . . . Q . .
. . . . . . Q .
. . Q . . . . .
. . . . . Q . .
. . Q . . . . .

current state h: 5
Probability of accepting worse solution: 0.6741
Rejected worse solution based on probability.
current state checkered page:
. . . . . . Q .
. . . . Q . . .
. Q . . . . . .
. . . . . Q . .
. . . . . . Q .
. . Q . . . . .
. . . . . Q . .
. . Q . . . . .
```

```
current state h: 5
Probability of accepting worse solution: 0.6108
Rejected worse solution based on probability.
current state checkered page:
. . . . . . Q .
. . . . Q . . .
. Q . . . . . .
. . . . . Q . .
. . . . . . Q .
. . Q . . . . .
. . . . . Q . .
. . Q . . . . .

current state h: 5
Probability of accepting worse solution: 1.0000
Accepted worse solution based on probability.
current state checkered page:
. . . . . . Q .
. . . . Q . . .
. Q . . . . . .
. . . . . Q . .
. . . . . . . Q
. . Q . . . . .
. . . . . Q . .
. . Q . . . . .

current state h: 5
Probability of accepting worse solution: 0.2142
Rejected worse solution based on probability.
current state checkered page:
. . . . . . Q .
. . . . Q . . .
. Q . . . . . .
. . . . . Q . .
. . . . . . . Q
. . Q . . . . .
. . . . . Q . .
. . Q . . . . .

current state h: 5
Probability of accepting worse solution: 0.3818
Rejected worse solution based on probability.
current state checkered page:
. . . . . . Q .
. . . . Q . . .
. Q . . . . . .
. . . . . Q . .
. . . . . . . Q
. . Q . . . . .
. . . . . Q . .
. . Q . . . . .
```

WEEK 7:
Program 9:
Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

Algorithm:

2. Propositional Logic : Truth Table Enumeration

Program:

```
import itertools
def evaluate_formula(formula, valuation):
    formula = formula.replace('p', str(valuation['p']))
    formula = formula.replace('q', str(valuation['q']))

    return eval(formula)

def extract_variables(formula):
    variables = set()
    for char in formula:
        if char.isalpha():
            variables.add(char)
    return list(variables)

def generate_truth_table(kb, query):
    variables = extract_variables(kb) + extract_variables(query)
    variables = list(set(variables))

    print("Truth Table")
    print("|".join(variables + ["KB", "Query"]))
    print("-" * (len(variables) * 4 + 12))

    entails_query = True

for assignment in itertools.product([False, True], repeat
                                    = len(variables)):

    valuation = dict(zip(variables, assignment))
    KB_truth = evaluate_formula(query, valuation)

    row = [str('T' if valuation[var] else 'F') for var
    in variables]
    row.append(str('T' if query_truth else 'F'))

print(
if KB_truth and not query_truth:
    entails_query = False

KB = input("Enter KB ")
query = input("enter query")

generate_truth_table(KB, query)
```

OUTPUT:
Enter KB (eg : 'P and (q1 = q)'): 'p'
Enter the query : q
Truth Table:

p	q	KB	Query
F	F	False	F
F	T	T	T
T	F	T	F
T	T	F	T

KB entails query : False

LAB-2
First Order Logic:
1. "John is a human"
FOL:
H(John)
Explanation:
H(x) : x is human
John is a constant representing an individual named John.
2. "Every human is mortal"
FOL:
$\forall x (H(x) \rightarrow M(x))$
H(x): x is a human
M(x): x is mortal
∀x: For all individual x, if x is a human then x is mortal
3. "John loves May"
L(John, May)
L(x,y): x loves y
John and May are individuals.
4. "There is someone who loves May"
$\exists x\, (L(x, May))$
L(x,y): x loves y
∃x: There exists an individual x such that x loves May
5. All dogs are animals.
FOL
$\forall x\, (D(x) \rightarrow A(x))$
D(x): x is a dog
A(x): x is an animal
∀x: For all individuals x, if x is dog, x is animal

6. "Some dogs are brown"
$\exists x\, (D(x) \wedge B(x))$
D(x) is: x is a dog
B(x): x is brown
∃x: There exists an individual x such that x is a dog and x is brown

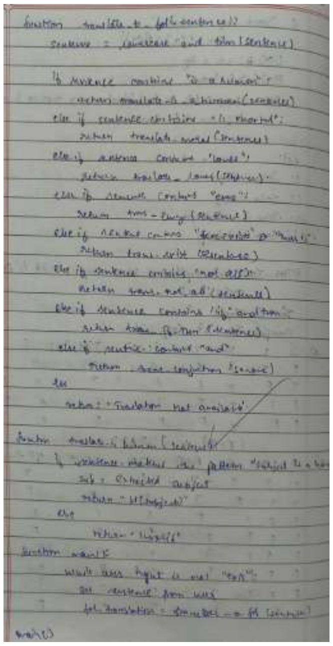
1) Let:
S: John is stupid
L: John is lazy
¬S: John is not stupid
¬L: John isn't lazy

Either John isn't stupid and he is lazy or he is stupid
$(\neg S \wedge L) \vee S$
John is stupid S
Therefore John isn't lazy ¬L

S	L	¬S	¬L	(¬S∧L)	(¬S∧L)∨S	S	¬L
T	T	F	F	F	T	T	F
T	F	F	F	F	T	T	T
F	T	T	T	T	T	F	F
F	F	T	F	F	F	F	T

$(P \rightarrow Q) \vee (\neg P \wedge R)$

2) P	Q	R	P→Q	¬P	¬P∧R	If P then Q else R
T	T	T	T	F	T	T
T	T	F	T	F	F	T
T	F	T	F	F	T	T
T	F	F	F	F	F	F
F	T	T	T	T	T	T
F	T	F	T	T	F	T
F	F	T	T	T	T	T
F	F	F	T	T	F	F



Code:

```
import itertools

def evaluate_kb(Q, P, R):
    q_imp_p = (not Q or P)
    p_imp_not_q = (not P or not Q)
    q_union_r = (Q or R)

    r = R
    r_imp_p = (not R or P)
    q_imp_r = (not Q or R)
    return q_imp_p, p_imp_not_q, q_union_r, r, r_imp_p, q_imp_r

def truth_table():
    values = [True, False]
    print(f"{'Q':<5}{'P':<5}{'R':<5}{'Q -> P':<10}{'P -> ¬Q':<10}{'Q ∪ R':<10}{'R':<5}{'R -> 
P':<10}{'Q -> R':<10}")
    for Q, P, R in itertools.product(values, repeat=3):
        q_imp_p, p_imp_not_q, q_union_r, r, r_imp_p, q_imp_r = evaluate_kb(Q, P, R)

print(f"{Q:<5}{P:<5}{R:<5}{q_imp_p:<10}{p_imp_not_q:<10}{q_union_r:<10}{r:<5}{r_imp_p:
<10}{q_imp_r:<10}")

        if q_imp_p and p_imp_not_q and q_union_r:
            print(f"\nKB is true for Q = {Q}, P = {P}, R = {R}:")
            print(f"- Q -> P: {q_imp_p}")
```

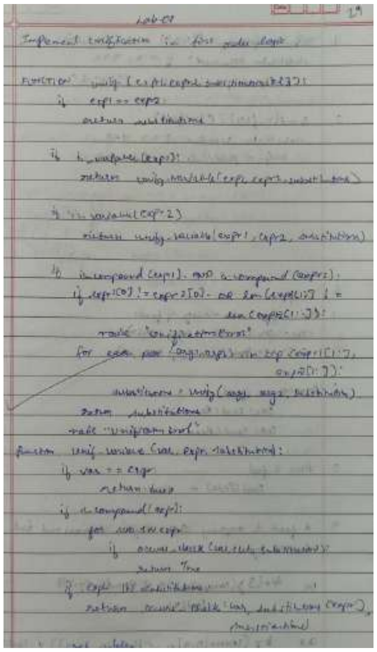
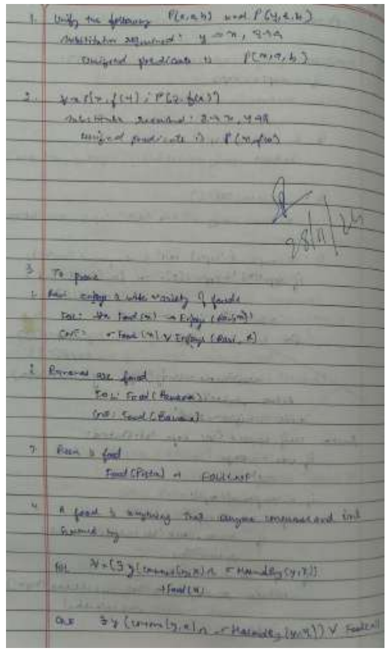
```
        print(f"- P -> ¬Q: {p_imp_not_q}")
        print(f"- Q ∪ R: {q_union_r}")
        print(f"Entailments:")
        print(f"- R: {r}")
        print(f"- R -> P: {r_imp_p}")
        print(f"- Q -> R: {q_imp_r}")
        print("-" * 50)

def main():
    truth_table()

main()
```

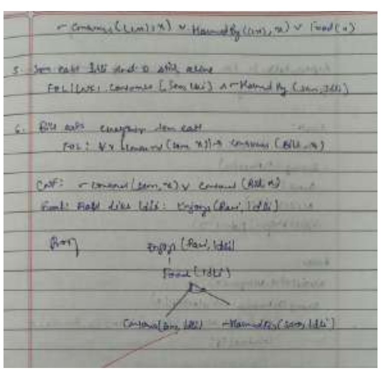
Output:

```
Q    P    R    Q -> P    P -> ¬Q    Q ∪ R     R    R -> P    Q -> R
1    1    1    1         0          1         1    1         1
1    1    0    1         0          1         0    1         0
1    0    1    0         1          1         1    0         1
1    0    0    0         1          1         0    1         0
0    1    1    1         1          1         1    1         1

KB is true for Q = False, P = True, R = True:
- Q -> P: True
- P -> ¬Q: True
- Q ∪ R: True
Entailments:
- R: True
- R -> P: True
- Q -> R: True
--------------------------------------------------
0    1    0    1         1          0         0    1         1
0    0    1    1         1          1         1    0         1

KB is true for Q = False, P = False, R = True:
- Q -> P: True
- P -> ¬Q: True
- Q ∪ R: True
Entailments:
- R: True
- R -> P: False
- Q -> R: True
--------------------------------------------------
0    0    0    1         1          0         0    1         1
```

<u>WEEK 8:</u>
<u>Program 10:</u>
Implement unification in first order logic.

Algorithm:



Code:

```
def unify(term1, term2, substitution=None):
    if substitution is None:
        substitution = {}
    if term1 == term2:
        return substitution
    if is_var(term1):
        return unify_var(term1, term2, substitution)
    if is_var(term2):
        return unify_var(term2, term1, substitution)
```

```
    if is_compound(term1) and is_compound(term2):
        if term1[0] != term2[0] or len(term1[1]) != len(term2[1]):
            return None
        for arg1, arg2 in zip(term1[1], term2[1]):
            substitution = unify(arg1, arg2, substitution)
            if substitution is None:
                return None
        return substitution
    if isinstance(term1, list) and isinstance(term2, list):
        if len(term1) != len(term2):
            return None
        for elem1, elem2 in zip(term1, term2):
            substitution = unify(elem1, elem2, substitution)
            if substitution is None:
                return None
        return substitution
    return None

def unify_var(variable, expr, substitution):
    if variable in substitution:
        return unify(substitution[variable], expr, substitution)
    if expr in substitution:
        return unify(variable, substitution[expr], substitution)
    if occurs(variable, expr, substitution):
        return None
    substitution[variable] = expr
    return substitution

def occurs(variable, expr, substitution):
    if variable == expr:
        return True
    if is_compound(expr):
        return any(occurs(variable, arg, substitution) for arg in expr[1])
    if isinstance(expr, list):
        return any(occurs(variable, item, substitution) for item in expr)
    if expr in substitution:
        return occurs(variable, substitution[expr], substitution)
    return False

def is_var(term):
    return isinstance(term, str) and term.startswith('?')

def is_compound(term):
    return isinstance(term, tuple) and len(term) == 2 and isinstance(term[1],
list)
```

```python
if __name__ == "__main__":
    print("Input format:")
    print("Compound: ('predicate', ['arg1', 'arg2'])")
    print("Variable: '?var'")
    print("List: ['a', 'b']")
    print("Constant: 'a', 'b'\n")
    t1 = eval(input("Enter first term: "))
    t2 = eval(input("Enter second term: "))
    result = unify(t1, t2)
    if result is None:
        print("Result: Unification failed")
    else:
        print("Result: Unification successful")
        print("Substitution:", result)
```

Output:

```
Input format:
Compound: ('predicate', ['arg1', 'arg2'])
Variable: '?var'
List: ['a', 'b']
Constant: 'a', 'b'

Enter first term: ('Parent',['?x','John'])
Enter second term: ('Parent',['David','?y'])
Result: Unification successful
Substitution: {'?x': 'David', '?y': 'John'}
```

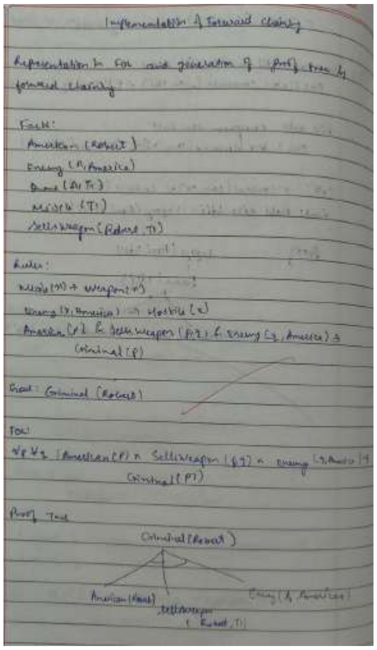
```
Input format:
Compound: ('predicate', ['arg1', 'arg2'])
Variable: '?var'
List: ['a', 'b']
Constant: 'a', 'b'

Enter first term: ('parent',['Jon','?x'])
Enter second term: ('son',['Jon','?x'])
Result: Unification failed
```

WEEK 9:

Program 11:

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

Algorithm:



Implementation of forward chaining

Facts:

American (Robert)

Enemy (A, America)

Missile (T1)

Sells weapon (Robert, T1)

Rules:

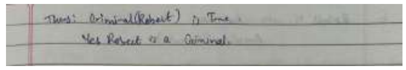
Goal: Criminal (Robert)

FOL:

Proof Tree

Criminal (Robert)

American (Robert)



Thus: Criminal(Robert) is True

Yes Robert is a Criminal.

Code:

```
class KnowledgeBase:
    def __init__(self):
        self.facts = set()
        self.rules = []

    def add_fact(self, fact):
        self.facts.add(fact)

    def add_rule(self, rule):
        self.rules.append(rule)

    def infer(self):
        derived_new = True
        while derived_new:
            derived_new = False
            for rule in self.rules:
                if rule.evaluate(self.facts):
                    derived_new = True
```

```
class Rule:
    def __init__(self, conditions, conclusion):
        self.conditions = conditions
        self.conclusion = conclusion

    def evaluate(self, facts):
        if all(condition in facts for condition in self.conditions):
            if self.conclusion not in facts:
                facts.add(self.conclusion)
                print(f"Derived: {self.conclusion}")
                return True
        return False

kb = KnowledgeBase()
kb.add_fact("American(Robert)")
kb.add_fact("Missile(T1)")
kb.add_fact("Owns(A, T1)")
kb.add_fact("Enemy(A, America)")
kb.add_rule(Rule(["Missile(T1)"], "Weapon(T1)"))
kb.add_rule(Rule(["Enemy(A, America)"], "Hostile(A)"))
kb.add_rule(Rule(["Missile(T1)", "Owns(A, T1)"], "Sells(Robert, T1, A)"))
kb.add_rule(Rule(["American(Robert)", "Weapon(T1)", "Sells(Robert, T1, A)", "Hostile(A)"],
"Criminal(Robert)"))
kb.infer()

if "Criminal(Robert)" in kb.facts:
    print("Outcome: Robert is a criminal.")
else:
    print("Outcome: Unable to prove Robert is a criminal.")
```

Output:

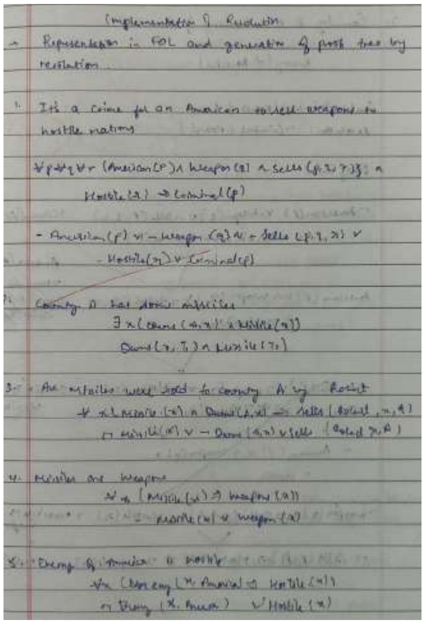
```
Derived: Weapon(T1)
Derived: Hostile(A)
Derived: Sells(Robert, T1, A)
Derived: Criminal(Robert)
Outcome: Robert is a criminal.
```

WEEK 10:

Program 12;

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

Algorithm:



Implementation of Resolution

Representation in FOL and generation of proof tree by resolution.

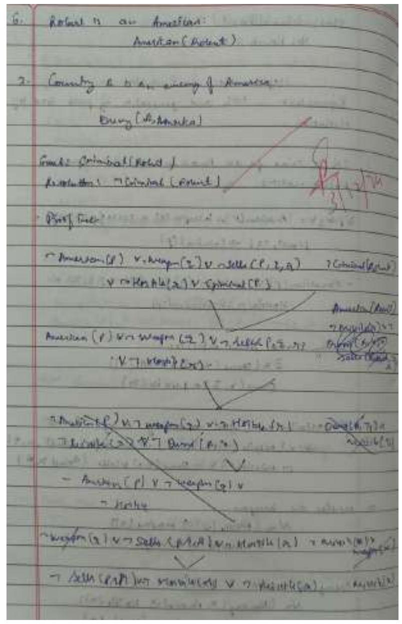
1. It's a crime for an American to sell weapons to hostile nations

2. Country A has some missiles

3. All missiles were sold to country A by Robert

4. Missiles are weapons

5. Enemy of America is hostile



6. Robert is an American:

American(Robert)

7. Country A is an enemy of America:

Enemy(A, America)

Goal: Criminal(Robert)

Resolution: ¬Criminal(Robert)

Proof Tree:

Code:

```
from itertools import combinations

def unify(term1, term2, substitution={}):
    if substitution is None:
        return None
    elif term1 == term2:
        return substitution
    elif isinstance(term1, str) and term1.islower():
        return unify_var(term1, term2, substitution)
    elif isinstance(term2, str) and term2.islower():
        return unify_var(term2, term1, substitution)
    elif isinstance(term1, tuple) and isinstance(term2, tuple) and len(term1) == len(term2):
        return unify(term1[1:], term2[1:], unify(term1[0], term2[0], substitution))
    else:
        return None

def unify_var(variable, value, substitution):
    if variable in substitution:
        return unify(substitution[variable], value, substitution)
```

```
    elif value in substitution:
        return unify(variable, substitution[value], substitution)
    else:
        substitution[variable] = value
        return substitution

def resolve_clause(clause1, clause2):
    resolvents = []
    for term1 in clause1:
        for term2 in clause2:
            substitution = unify(term1, negate_term(term2))
            if substitution is not None:
                new_clause = (apply_substitution(clause1, substitution) | apply_substitution(clause2,
substitution)) - {term1, term2}
                resolvents.append(frozenset(new_clause))
    return resolvents

def negate_term(predicate):
    return ('not', predicate) if isinstance(predicate, str) else predicate[1]

def apply_substitution(clause, substitution):
    return {apply_single_substitution(p, substitution) for p in clause}

def apply_single_substitution(predicate, substitution):
    if isinstance(predicate, str):
        return substitution.get(predicate, predicate)
    else:
        return (predicate[0],) + tuple(substitution.get(arg, arg) for arg in predicate[1:])

def resolution_proof(kb, query):
    negated_query = frozenset({negate_term(query)})
    clauses = kb | {negated_query}
    new_clauses = set()

    while True:
        for clause1, clause2 in combinations(clauses, 2):
            resolvents = resolve_clause(clause1, clause2)
            if frozenset() in resolvents:
                return True
            new_clauses.update(resolvents)
        if new_clauses.issubset(clauses):
            return False
        clauses |= new_clauses

kb = {
    frozenset({('Mother', 'Leela', 'Oshin')}),
```

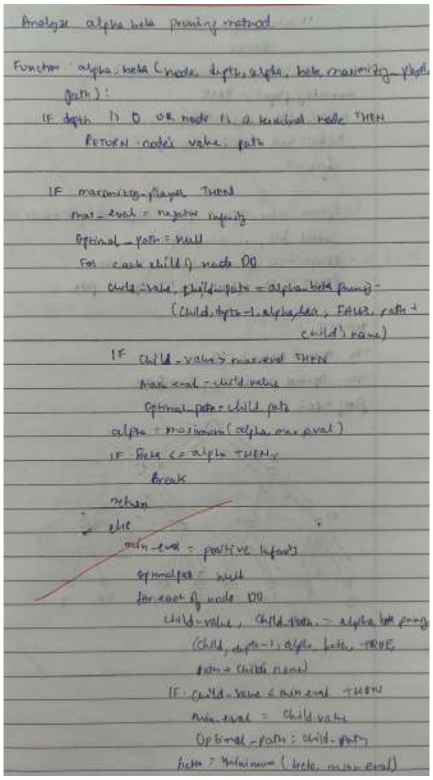
```
    frozenset({('Alive', 'Leela')}),
    frozenset({('not','Mother', 'x','y')}),
    frozenset({('Parent','x','y')}),
    frozenset({('not','Parent', 'w', 'z')}),
    frozenset({('not','Alive','w','z')}),
    frozenset({('Older','w','z')}),
}

query = ('Older', 'Leela', 'Oshin')

result = resolution_proof(kb, query)
if result:
    print("Proved by resolution: Leela is older than Oshin.")
else:
    print("Cannot prove: Leela is not older than Oshin.")
```

Output:

```
Proved by resolution: Leela is older than Oshin.
```

WEEK 11:
Program 13:
Implement Alpha-Beta Pruning.

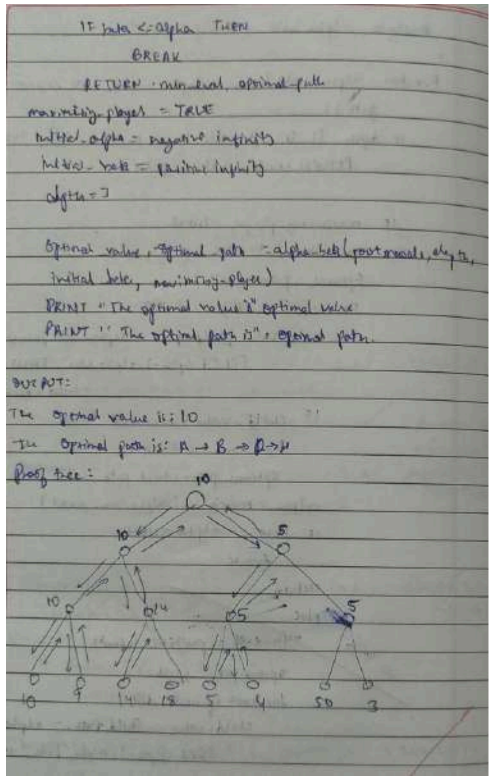
Algorithm:



Analyze alpha beta pruning method

Function alpha_beta (node, depth, alpha, beta, maximizing_player, path):
IF depth is 0 OR node is a terminal node THEN
RETURN node's value, path

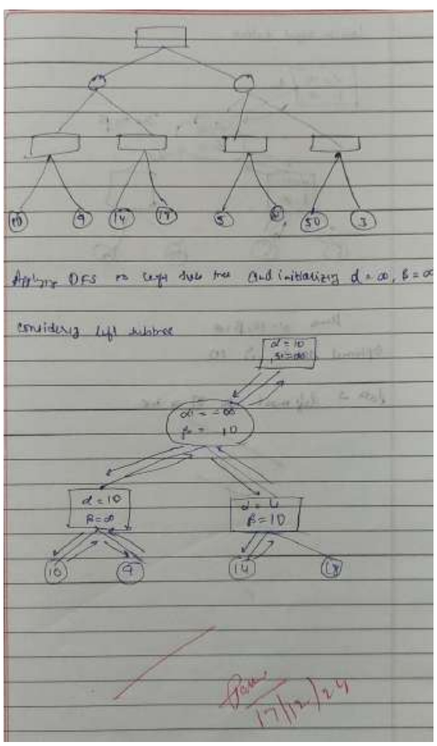
IF maximizing_player THEN
max_eval = negative infinity
optimal_path = null
FOR each child of node DO
child_value, child_path = alpha_beta_pruning (child, depth-1, alpha, beta, FALSE, path + child's name)
IF child_value > max_eval THEN
max_eval = child_value
optimal_path = child_path
alpha = maximum(alpha, max_eval)
IF beta <= alpha THEN
break
return
else
min_eval = positive infinity
optimal_path = null
for each of node DO
child_value, child_path = alpha_beta_pruning (child, depth-1, alpha, beta, TRUE, path + child's name)
IF child_value < min_eval THEN
min_eval = child_value
optimal_path = child_path
beta = minimum (beta, min_eval)



IF beta <= alpha THEN
BREAK
RETURN min_eval, optimal_path
maximizing_player = TRUE
initial_alpha = negative infinity
initial_beta = positive infinity
depth = 3

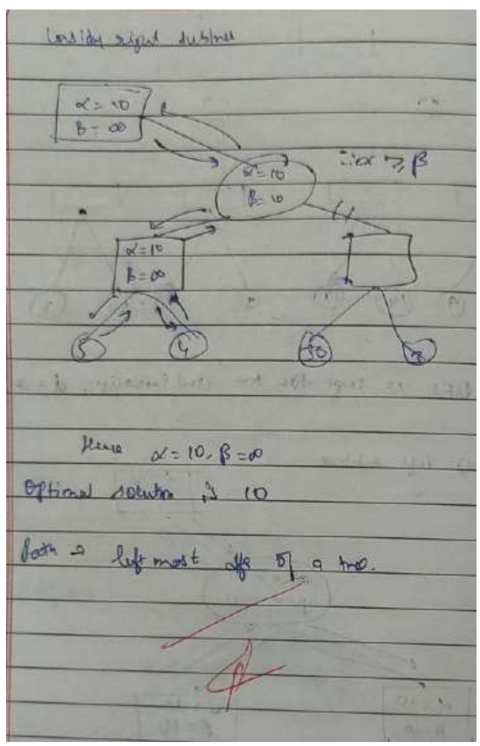
optimal value, optimal path = alpha_beta(root_node, depth, initial_alpha, initial_beta, maximizing_player)
PRINT "The optimal value is" optimal value
PRINT "The optimal path is" optimal path

OUTPUT:
The optimal value is: 10
The optimal path is: A → B → D → H
Proof tree:



Applying DFS to left sub tree and initializing $\alpha = \infty, \beta = \infty$

Considering left subtree



Consider right subtree

Hence $\alpha = 10, \beta = \infty$
Optimal solution is 10
Path → left most dfs of a tree.

Code:

```
class GameNode:
    def __init__(self, value=None, children=None):
        self.value = value
        self.children = children if children else []

def alpha_beta(node, depth, alpha, beta, is_maximizing):
    if not node.children or depth == 0:
        return node.value
    if is_maximizing:
        best_value = float('-inf')
        for child in node.children:
            value = alpha_beta(child, depth - 1, alpha, beta, False)
            best_value = max(best_value, value)
            alpha = max(alpha, value)
            if beta <= alpha:
                print(f"Pruned at MAX node with alpha={alpha}, beta={beta}")
                break
        node.value = best_value
        return best_value
    else:
        best_value = float('inf')
        for child in node.children:
            value = alpha_beta(child, depth - 1, alpha, beta, True)
            best_value = min(best_value, value)
            beta = min(beta, value)
            if beta <= alpha:
                print(f"Pruned at MIN node with alpha={alpha}, beta={beta}")
                break
        node.value = best_value
        return best_value

def display_tree(node, level=0):
    print("  " * level + f"Node Value: {node.value}")
    for child in node.children:
        display_tree(child, level + 1)

if __name__ == "__main__":
    root = GameNode(None, [
        GameNode(None, [
            GameNode(None, [GameNode(8), GameNode(7)]),
            GameNode(None, [GameNode(12), GameNode(15)])
        ]),
        GameNode(None, [
            GameNode(None, [GameNode(3), GameNode(2)]),
            GameNode(None, [GameNode(25), GameNode(1)])
```

```
    ])
])

print("Game Tree Before Alpha-Beta Pruning:")
display_tree(root)

final_value = alpha_beta(root, depth=3, alpha=float('-inf'), beta=float('inf'),
is_maximizing=True)

print("\nGame Tree After Alpha-Beta Pruning:")
display_tree(root)

print("\nFinal Value at MAX node:", final_value)
```

Output:

```
Game Tree Before Alpha-Beta Pruning:
Node Value: None
  Node Value: None
    Node Value: None
      Node Value: 8
      Node Value: 7
    Node Value: None
      Node Value: 12
      Node Value: 15
  Node Value: None
    Node Value: None
      Node Value: 3
      Node Value: 2
    Node Value: None
      Node Value: 25
      Node Value: 1
Pruned at MAX node with alpha=12, beta=8
Pruned at MIN node with alpha=8, beta=3

Game Tree After Alpha-Beta Pruning:
Node Value: 8
  Node Value: 8
    Node Value: 8
      Node Value: 8
      Node Value: 7
    Node Value: 12
      Node Value: 12
      Node Value: 15
  Node Value: 3
    Node Value: 3
      Node Value: 3
      Node Value: 2
    Node Value: None
      Node Value: 25
      Node Value: 1

Final Value at MAX node: 8
```