VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

SHREE SANKET (1BM22CS261)

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

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **SHREE SANKET (1BM22CS261),** 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.

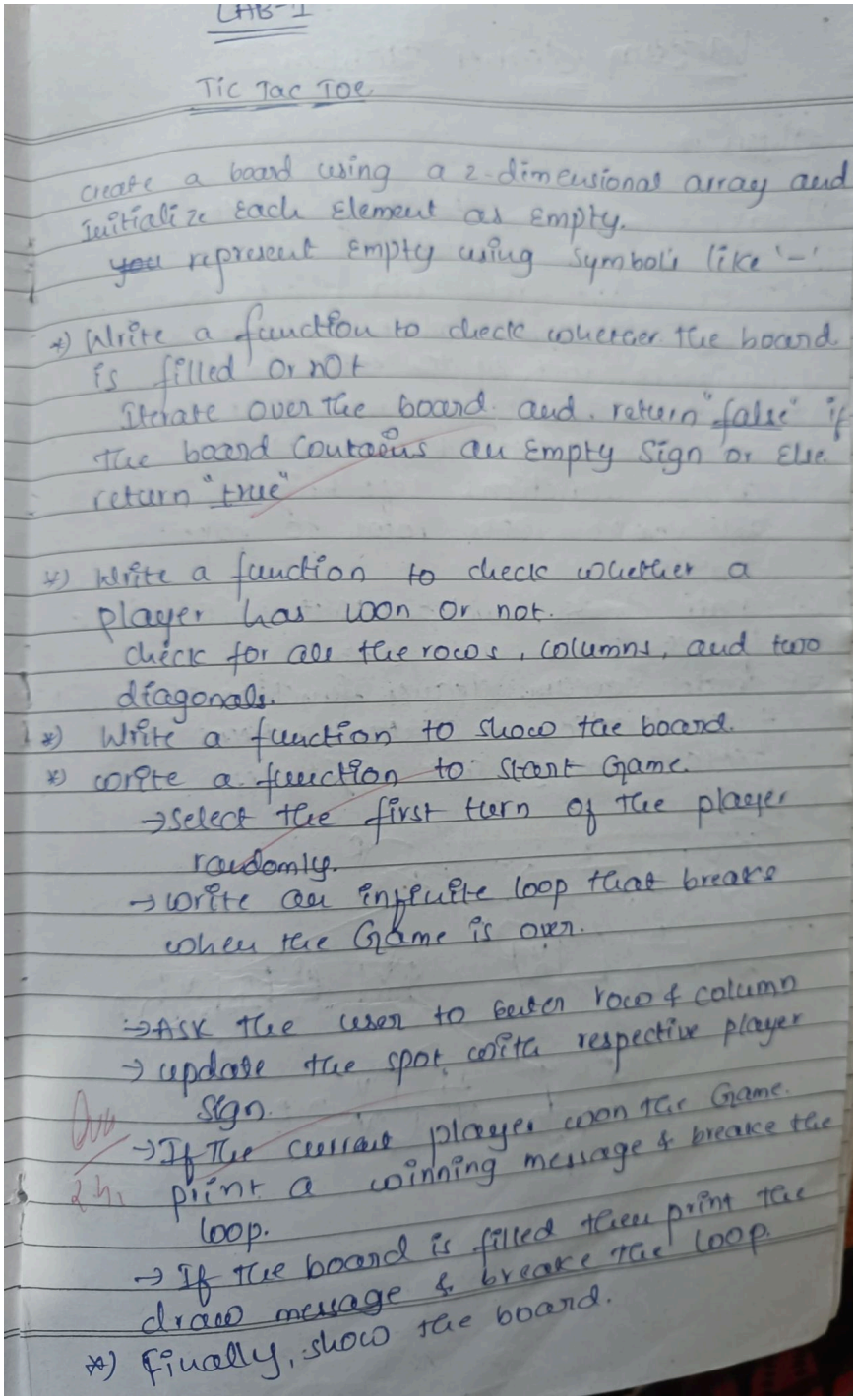
Sonika Sharma D Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/shreesanket/AI_LAB

Program 1
Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:


LAB-1

Tic Tac Toe

Create a board using a 2-dimensional array and initialize each element as Empty.
you represent Empty using symbols like '-'

*) Write a function to check whether the board is filled or not
Iterate over the board and return "false" if the board contains an Empty sign or else return "true"

*) Write a function to check whether a player has won or not.
check for all the rows, columns, and two diagonals.

*) Write a function to show the board.

*) write a function to start Game.
→ Select the first turn of the player randomly.
→ write an infinite loop that breaks when the Game is over.

→ Ask the user to enter row & column
→ update the spot with respective player sign.
→ If the current player won the Game. print a winning message & breake the loop.
→ If the board is filled then print the draw message & breake the loop.

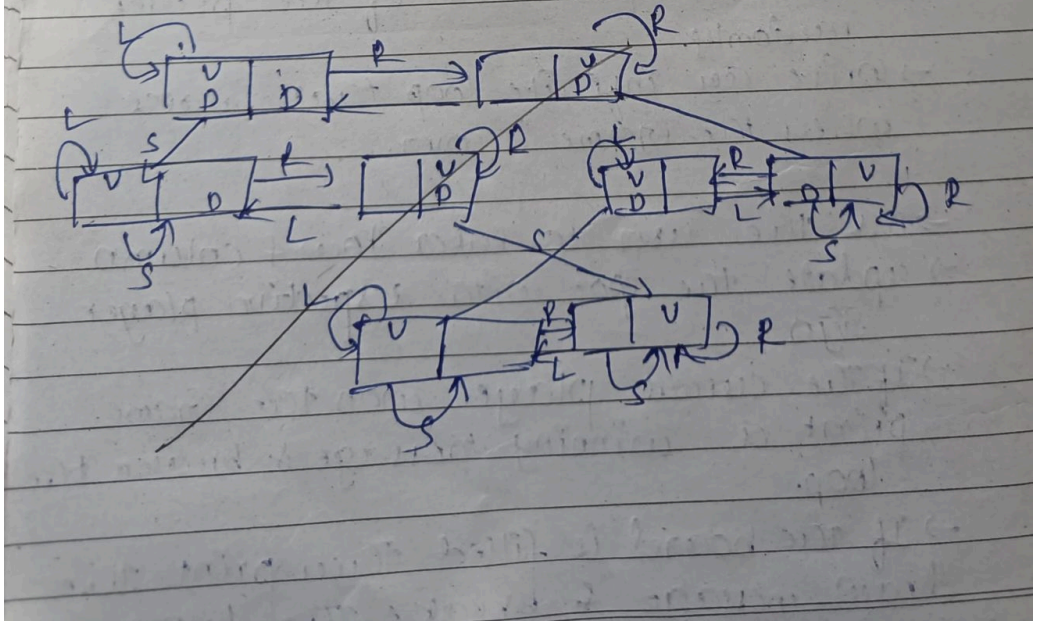
*) finally, show the board.

LAB -

Vaccum cleaner problem

Algorithm

1) Make an array to hold data of dirt in vaccum world
2) Take input from user about where the robot is and where the dirt is
3) initialize cost of the path to 0
4) Create a function suck that will make box clean & increases cost
5) create functions left & right which will move above robot to respective locations.
6) Create check function that will check if all boxes are clean
7) Create vaccum_cleaner function that will take location & status & returns an action among suck, right, left.



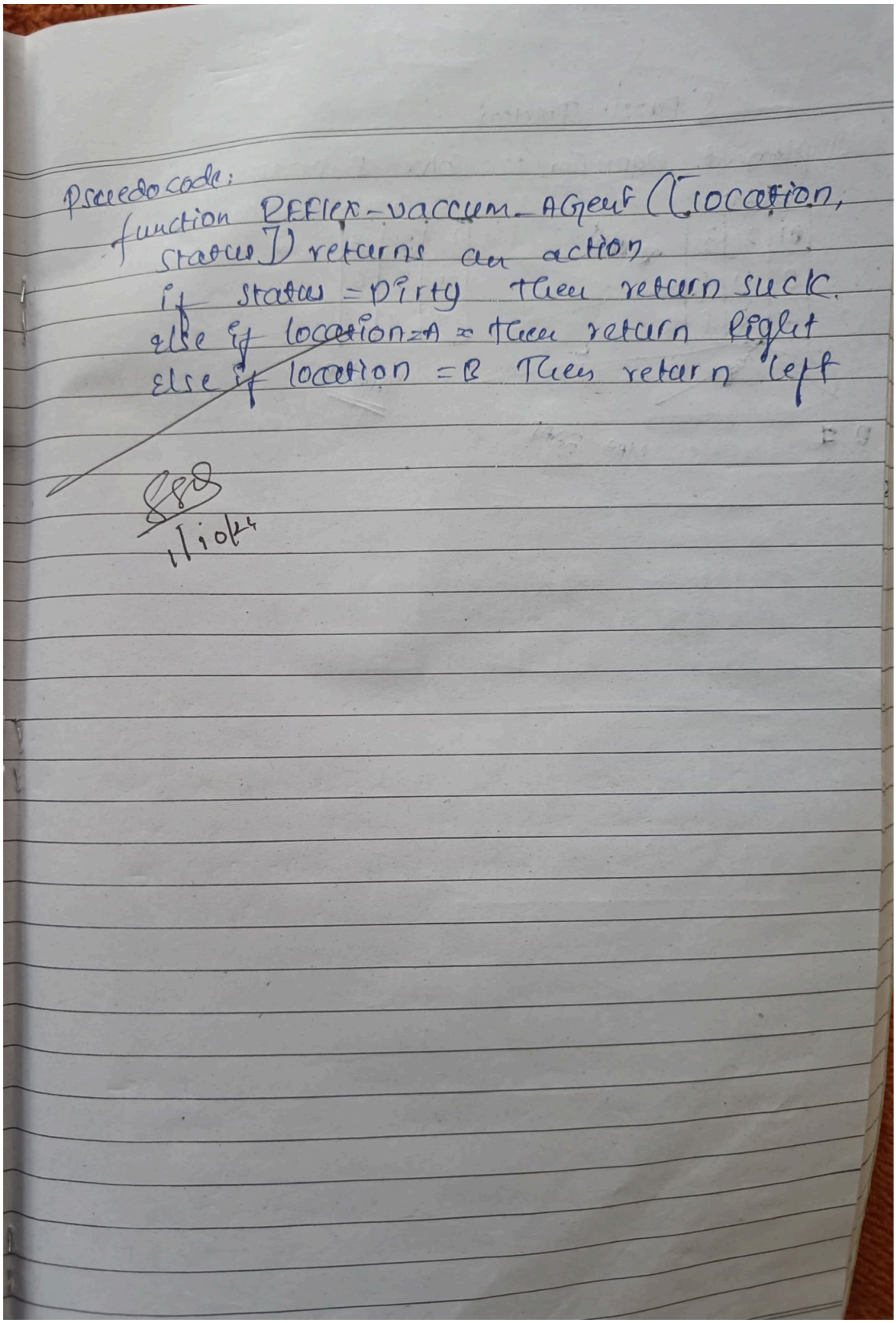
Pseudo code:

function REFLEX-VACCUM-AGENT ([location, status]) returns an action

if status = Dirty Then return SUCK

else if location = A Then return Right

Else if location = B Then return left



Code 1**:** Tic-Tac-Toe

```
import random

grid = [['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]

def is_grid_full():
    for row in grid:
        for spot in row:
            if spot == '_':
                return False
    return True

def check_winner(current_player):
    for row in grid:
        if all(spot == current_player for spot in row):
            return True
    for col in range(3):
        if all(grid[row][col] == current_player for row in range(3)):
            return True
    if all(grid[i][i] == current_player for i in range(3)):
        return True
    if all(grid[i][2 - i] == current_player for i in range(3)):
        return True
    return False

def display_grid():
    print("\n A B C ")
    print(" +---+---+---+")
    for idx, row in enumerate(grid):
        print(f"{idx + 1} | " + " | ".join(row) + " |")
        print(" +---+---+---+")
    print()

def run_game():
    active_player = random.choice(['X', 'O'])
    print(f"\nPlayer {active_player} goes first!\n")
    while True:
        display_grid()
        try:
            row = int(input(f"Player {active_player}, pick your row (1-3): ")) - 1
            col = input(f"Player {active_player}, pick your column (A-C): ").upper()
            col = ord(col) - ord('A')
        except (ValueError, IndexError):
            print("Invalid input! Please try again.")
            continue
        if 0 <= row < 3 and 0 <= col < 3:
```

```
            if grid[row][col] == '_':
                grid[row][col] = active_player
            else:
                print("That spot is already taken. Choose another.")
                continue
        else:
            print("Invalid row or column. Try again.")
            continue
        if check_winner(active_player):
            display_grid()
            print(f"Player {active_player} wins!")
            break
        elif is_grid_full():
            display_grid()
            print("It's a draw!")
            break
        active_player = 'O' if active_player == 'X' else 'X'

run_game()
```

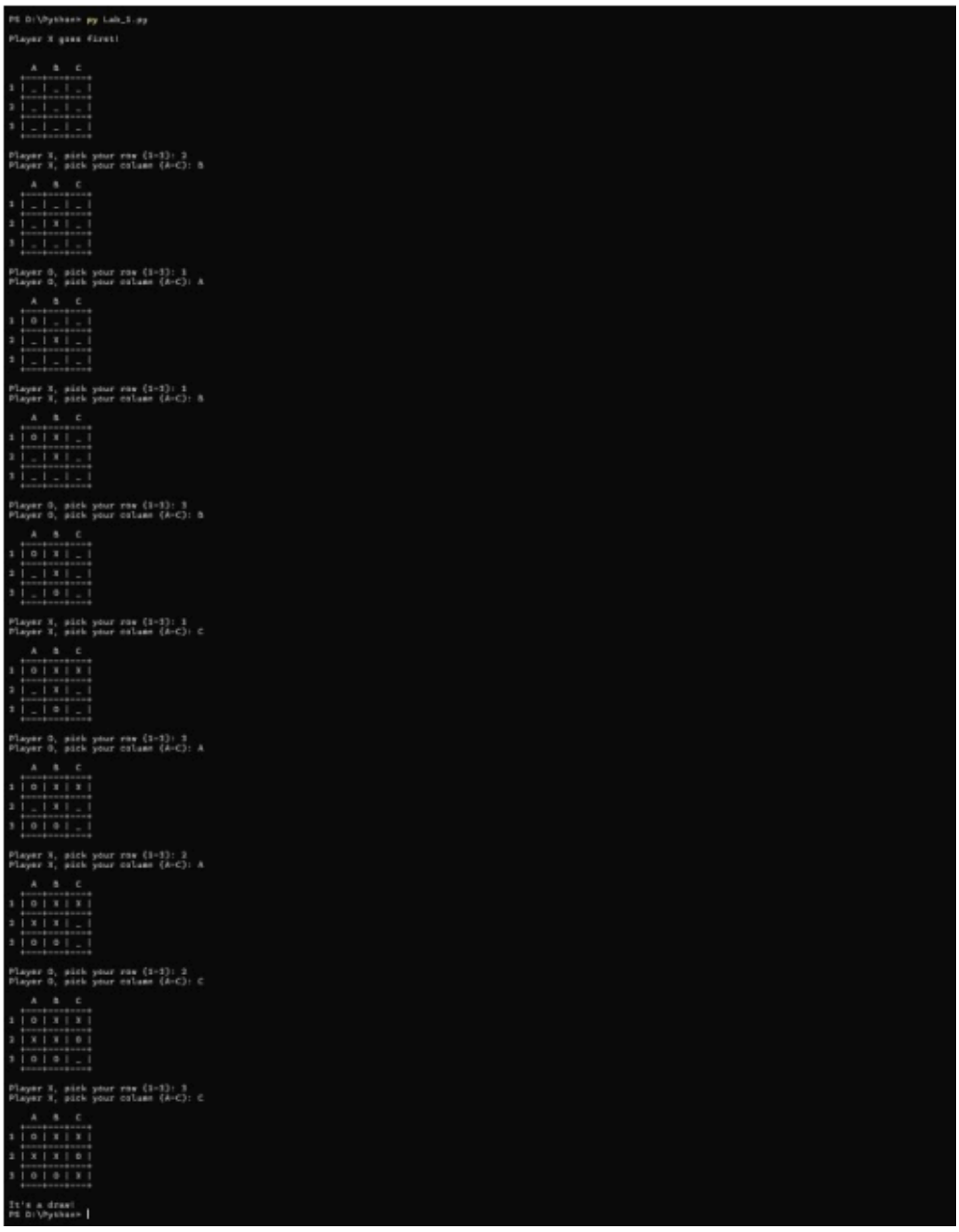
OutPut :

1) Invalid

2) Win


PS D:\Python> py Lab_1.py
Player O goes first!
A B C
1 | _ | _ | _ |
2 | _ | _ | _ |
3 | _ | _ | _ |
Player O, pick your row (1-3): 1
Player O, pick your column (A-C): A
A B C
1 | O | _ | _ |
2 | _ | _ | _ |
3 | _ | _ | _ |
Player X, pick your row (1-3): 2
Player X, pick your column (A-C): B
A B C
1 | O | _ | _ |
2 | _ | X | _ |
3 | _ | _ | _ |
Player O, pick your row (1-3): 1
Player O, pick your column (A-C): B
A B C
1 | O | O | _ |
2 | _ | X | _ |
3 | _ | _ | _ |
Player X, pick your row (1-3): 1
Player X, pick your column (A-C): C
A B C
1 | O | O | X |
2 | _ | X | _ |
3 | _ | _ | _ |
Player O, pick your row (1-3): 2
Player O, pick your column (A-C): A
A B C
1 | O | O | X |
2 | O | X | _ |
3 | _ | _ | _ |
Player X, pick your row (1-3): 3
Player X, pick your column (A-C): A
A B C
1 | O | O | X |
2 | O | X | _ |
3 | X | _ | _ |
Player X wins!
PS D:\Python>

3) Draw


PS D:\Python> py Lab_5.py
Player X goes first!
A B C
1 | _ | _ | _ |
2 | _ | _ | _ |
3 | _ | _ | _ |
Player X, pick your row (1-3): 2
Player X, pick your column (A-C): B
A B C
1 | _ | _ | _ |
2 | _ | X | _ |
3 | _ | _ | _ |
Player O, pick your row (1-3): 1
Player O, pick your column (A-C): A
A B C
1 | O | _ | _ |
2 | _ | X | _ |
3 | _ | _ | _ |
Player X, pick your row (1-3): 1
Player X, pick your column (A-C): B
A B C
1 | O | X | _ |
2 | _ | X | _ |
3 | _ | _ | _ |
Player O, pick your row (1-3): 3
Player O, pick your column (A-C): B
A B C
1 | O | X | _ |
2 | _ | X | _ |
3 | _ | O | _ |
Player X, pick your row (1-3): 1
Player X, pick your column (A-C): C
A B C
1 | O | X | X |
2 | _ | X | _ |
3 | _ | O | _ |
Player O, pick your row (1-3): 3
Player O, pick your column (A-C): A
A B C
1 | O | X | X |
2 | _ | X | _ |
3 | O | O | _ |
Player X, pick your row (1-3): 2
Player X, pick your column (A-C): A
A B C
1 | O | X | X |
2 | X | X | _ |
3 | O | O | _ |
Player O, pick your row (1-3): 2
Player O, pick your column (A-C): C
A B C
1 | O | X | X |
2 | X | X | O |
3 | O | O | _ |
Player X, pick your row (1-3): 3
Player X, pick your column (A-C): C
A B C
1 | O | X | X |
2 | X | X | O |
3 | O | O | X |
It's a draw!
PS D:\Python>

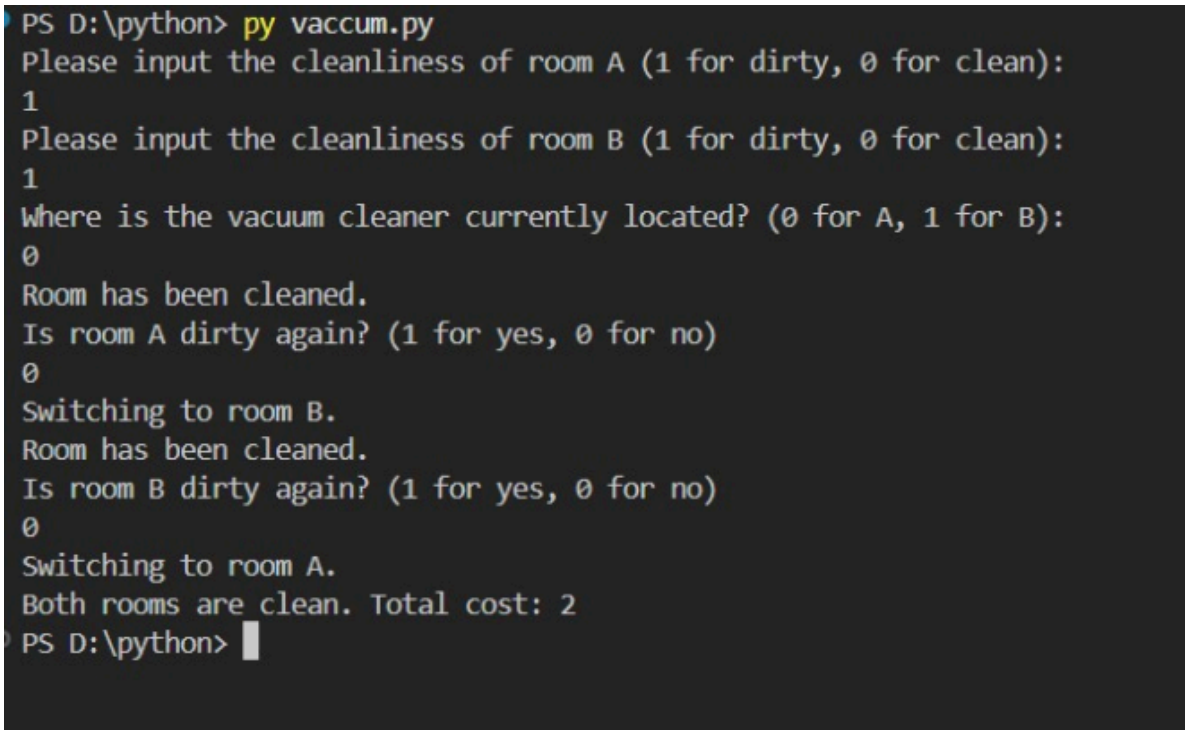
Code 2: Vacuum cleaner

```
cost = 0
status = [0, 0]
A = 0
B = 1

def suck(location):
    global cost
    if status[location] == 0:
        print("The room is already tidy.")
    else:
        status[location] = 0
        cost += 1
        print("Room has been cleaned.")
    if location == A:
        print("Is room A dirty again? (1 for yes, 0 for no)")
        status[A] = int(input())
    elif location == B:
        print("Is room B dirty again? (1 for yes, 0 for no)")
        status[B] = int(input())

def move_left(location):
    print("Switching to room A.")
    return A

def move_right(location):
    print("Switching to room B.")
    return B

def vaccume_cleaner(location):
    global cost, status, A, B
    if status[A] == 0 and status[B] == 0:
        print("Both rooms are clean. Total cost: " + str(cost))
        return
    if status[location] == 1:
        suck(location)
    else:
        print("The current room is already clean.")
    if location == A:
        new_loc = move_right(location)
        vaccume_cleaner(new_loc)
    elif location == B:
        new_loc = move_left(location)
        vaccume_cleaner(new_loc)

def main():
```

```
    global A, B, status
    print("Please input the cleanliness of room A (1 for dirty, 0 for clean):")
    status[A] = int(input())
    print("Please input the cleanliness of room B (1 for dirty, 0 for clean):")
    status[B] = int(input())
    print("Where is the vacuum cleaner currently located? (0 for A, 1 for B):")
    location = int(input())
    vaccume_cleaner(location)

main()
```

Output:

CASE 1) ROOM A IS DIRTY AND ROOM B IS ALSO DIRTY



CASE 2) BOTH THE ROOMS ARE CLEAN

```
PS D:\python> py vaccum.py
Please input the cleanliness of room A (1 for dirty, 0 for clean):
0
Please input the cleanliness of room B (1 for dirty, 0 for clean):
0
Where is the vacuum cleaner currently located? (0 for A, 1 for B):
0
Both rooms are clean. Total cost: 0
PS D:\python>
```

CASE 3) ROOM A IS DIRTY ROOM B IS CLEAN

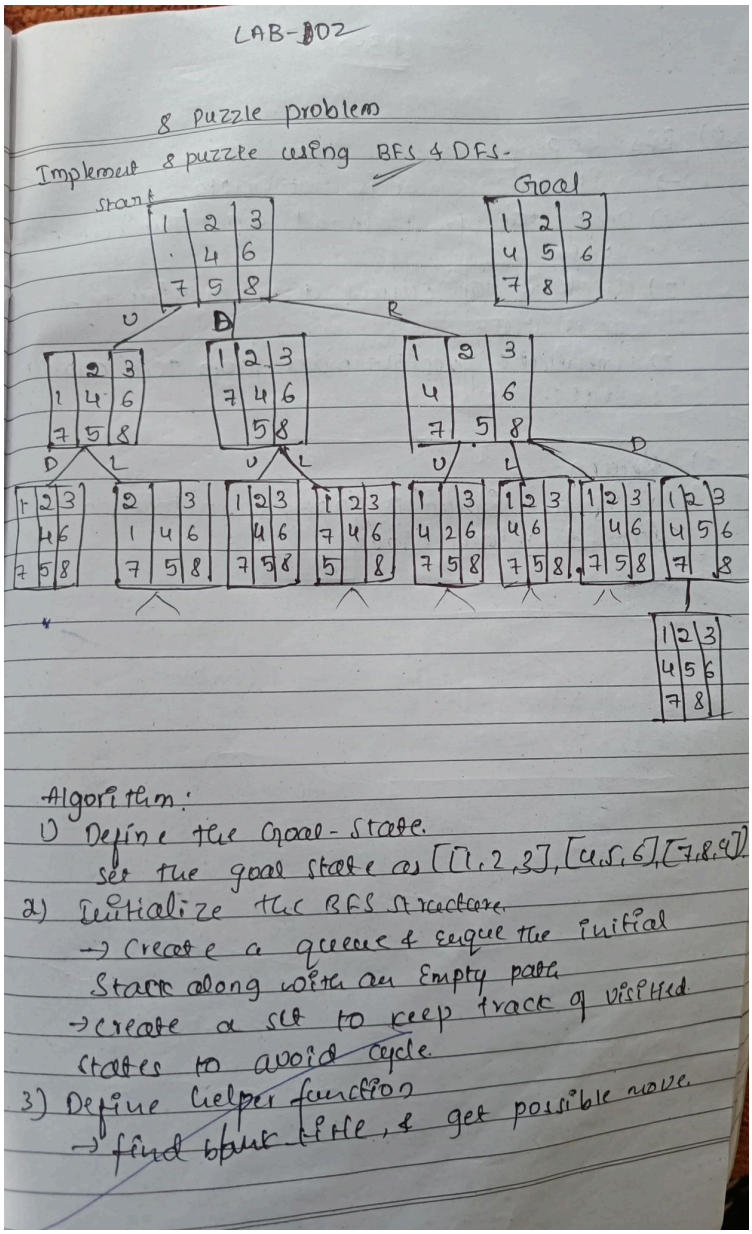
```
PS D:\python> py vaccum.py
Please input the cleanliness of room A (1 for dirty, 0 for clean):
1
Please input the cleanliness of room B (1 for dirty, 0 for clean):
0
Where is the vacuum cleaner currently located? (0 for A, 1 for B):
1
The current room is already clean.
Switching to room A.
Room has been cleaned.
Is room A dirty again? (1 for yes, 0 for no)
0
Switching to room B.
Both rooms are clean. Total cost: 1
PS D:\python>
```

CASE 4) ROOM A IS CLEAN B IS DIRTY

```
PS D:\python> py vaccum.py
Please input the cleanliness of room A (1 for dirty, 0 for clean):
0
Please input the cleanliness of room B (1 for dirty, 0 for clean):
1
Where is the vacuum cleaner currently located? (0 for A, 1 for B):
0
The current room is already clean.
Switching to room B.
Room has been cleaned.
Is room B dirty again? (1 for yes, 0 for no)
0
Switching to room A.
Both rooms are clean. Total cost: 1
PS D:\python>
```

Program 2:
Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm

Algorithm:



4) BFS loop.
→ while the queue is not empty
1) Dequeue the front element & get the current state
2) check if the current state matches the goal state.
3) Generate possible moves from the current state.

5) Return.
1) If the Queue is empty & no solution is found return None.

8/10/24

LAB 02 22/10/24.

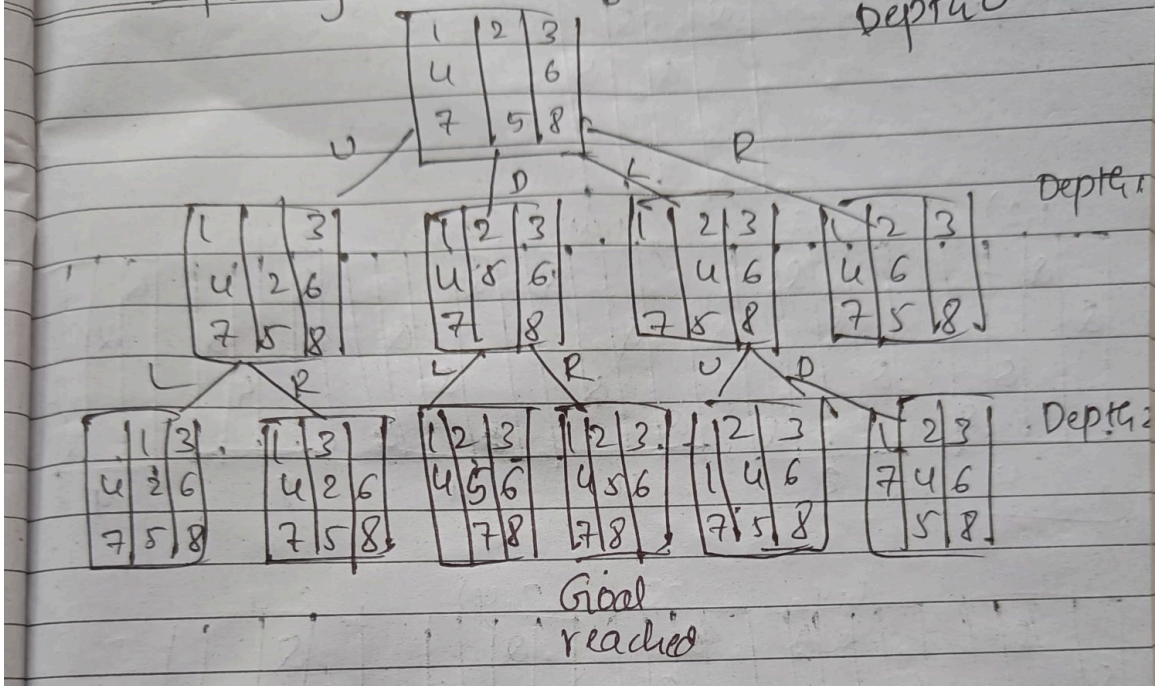
Lab : Implement Iterative Deepening Search Algorithm.

function Iterative-Deepening-Search(problem) returns a solution, or failure
for depth = 0 to ∞
result ← Depth-Limited-Search(problem, depth)
if result ≠ cutoff then return result

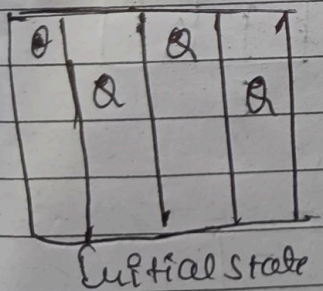
1) For each child of the current node
2) If it is the largest node, return
3) If the current maximum depth is reached, return
4) Set the current node to this node & go back to 1
5) After having gone through all children, go to the next child of the parent. (the next sibling)
6) After having gone through all children of the start node, increase the maximum depth & go back to 1
7. If we have reached all leaf (bottom) nodes, the goal node doesn't exist.

LAB-02

Depth Search Algorithm.



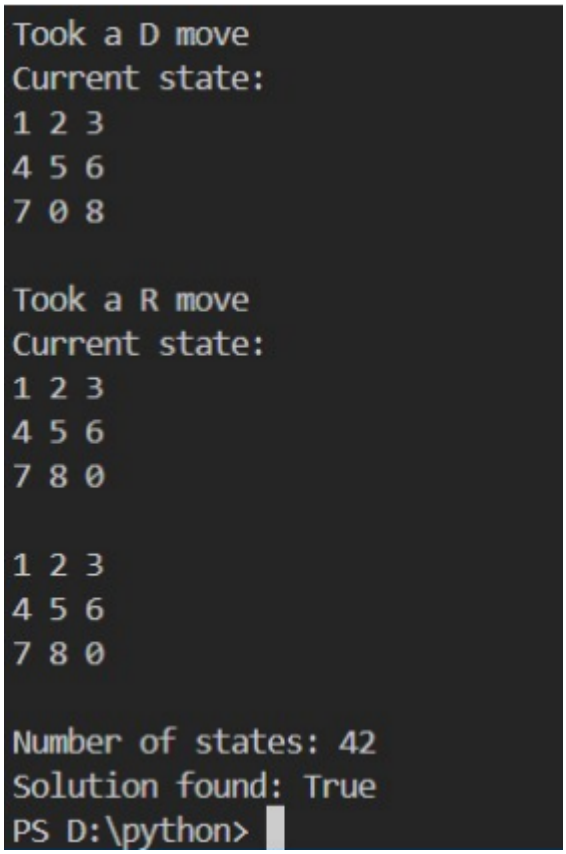
8 Queens Problem



Code 1: Implement 8 puzzle problems using Depth First Search (DFS)

```
cnt = 0

def print_state(in_array):
    global cnt
    cnt += 1
    for row in in_array:
        print(' '.join(str(num) for num in row))
    print()

def helper(goal, in_array, row, col, vis):
    vis[row][col] = 1
    drow = [-1, 0, 1, 0]
    dcol = [0, 1, 0, -1]
    dchange = ['U', 'R', 'D', 'L']
    print("Current state:")
    print_state(in_array)
    if in_array == goal:
        print_state(in_array)
        print(f"Number of states: {cnt}")
        return True
    for i in range(4):
        nrow = row + drow[i]
        ncol = col + dcol[i]
        if 0 <= nrow < len(in_array) and 0 <= ncol < len(in_array[0]) and not vis[nrow][ncol]:
            print(f"Took a {dchange[i]} move")
            in_array[row][col], in_array[nrow][ncol] = in_array[nrow][ncol],
in_array[row][col]
            if helper(goal, in_array, nrow, ncol, vis):
                return True
            in_array[row][col], in_array[nrow][ncol] = in_array[nrow][ncol],
in_array[row][col]
    vis[row][col] = 0
    return False

initial_state = [[1, 2, 3], [0, 4, 6], [7, 5, 8]]
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
visited = [[0] * 3 for _ in range(3)]
empty_row, empty_col = 1, 0
found_solution = helper(goal_state, initial_state, empty_row, empty_col, visited)
print("Solution found:", found_solution)
```

Output:


Took a D move
Current state:
1 2 3
4 5 6
7 0 8
Took a R move
Current state:
1 2 3
4 5 6
7 8 0
1 2 3
4 5 6
7 8 0
Number of states: 42
Solution found: True
PS D:\python>

Code 2: Implement Iterative deepening search algorithm

```
class PuzzleState:
    def __init__(self, board, empty_tile_pos, depth=0, path=[]):
        self.board = board
        self.empty_tile_pos = empty_tile_pos
        self.depth = depth
        self.path = path

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

    def generate_moves(self):
        row, col = self.empty_tile_pos
        moves = []
        directions = [(-1, 0, 'Up'), (1, 0, 'Down'), (0, -1, 'Left'), (0, 1, 'Right')]
        for dr, dc, move_name in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_board = self.board[:]
                new_board[row * 3 + col], new_board[new_row * 3 + new_col] = (
                    new_board[new_row * 3 + new_col],
                    new_board[row * 3 + col],
                )
                new_path = self.path + [move_name]
                moves.append(PuzzleState(new_board, (new_row, new_col), self.depth + 1,
new_path))
        return moves

    def display(self):
        for i in range(0, 9, 3):
            print(self.board[i:i + 3])
        print(f"Moves: {self.path}")
        print()


def iddfs(initial_state, goal, max_depth):
    for depth in range(max_depth + 1):
        print(f"Searching at depth: {depth}")
        found = dls(initial_state, goal, depth)
        if found:
            print(f"Goal found at depth: {found.depth}")
            found.display()
            return found
    print("Goal not found within max depth.")
    return None
```

```
def dls(state, goal, depth):
    if state.is_goal(goal):
        return state
    if depth <= 0:
        return None
    for move in state.generate_moves():
        print("Current state:")
        move.display()
        result = dls(move, goal, depth - 1)
        if result is not None:
            return result
    return None


def main():
    initial_state_input = input(
        "Enter initial state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8 0'): "
    )
    goal_state_input = input(
        "Enter goal state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8 0'): "
    )
    max_depth = int(input("Enter maximum depth: "))
    initial_board = list(map(int, initial_state_input.split()))
    goal_board = list(map(int, goal_state_input.split()))
    empty_tile_pos = initial_board.index(0) // 3, initial_board.index(0) % 3
    initial_state = PuzzleState(initial_board, empty_tile_pos)
    solution = iddfs(initial_state, goal_board, max_depth)


if __name__ == "__main__":
    main()
```

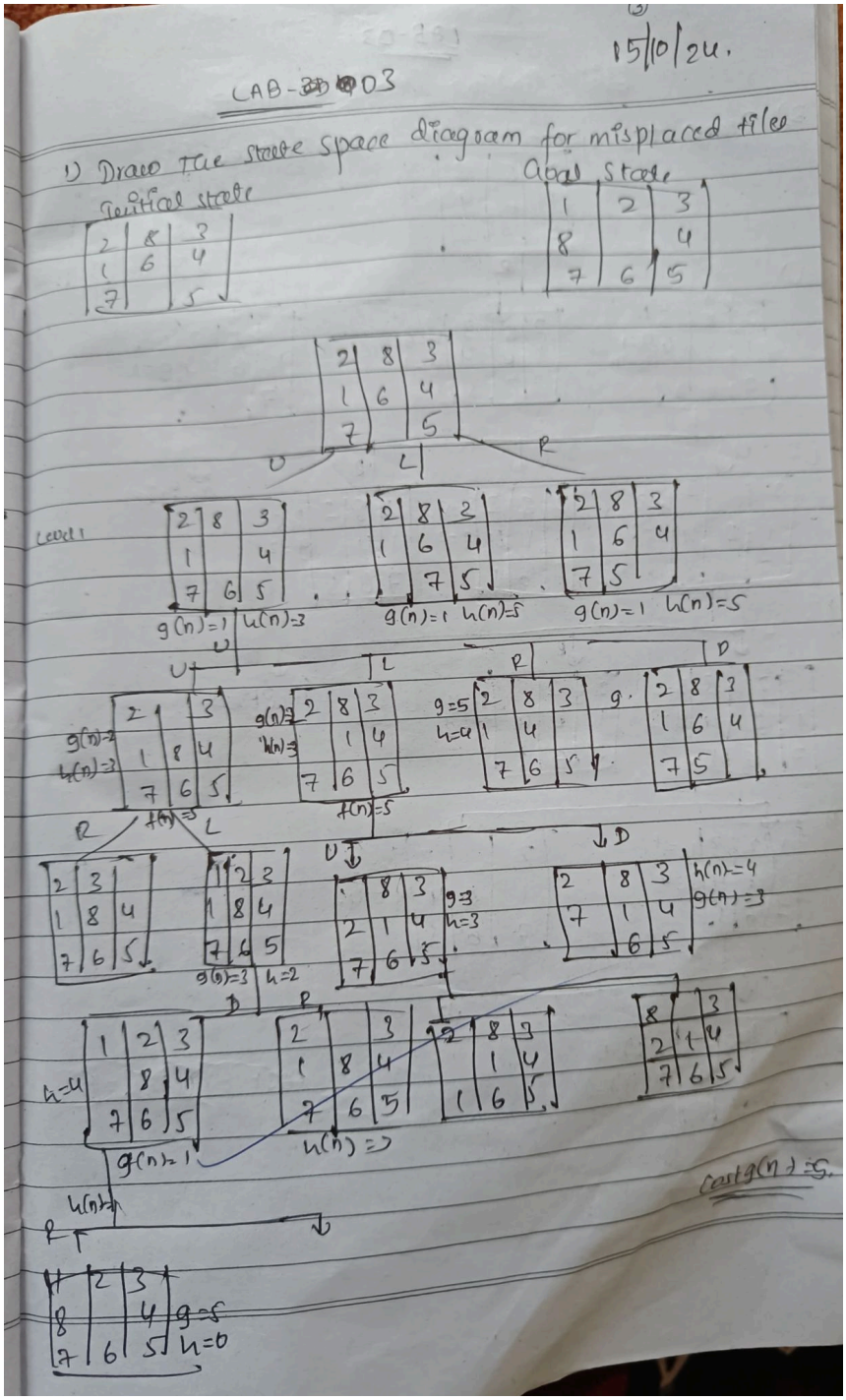
Output:

```
Enter initial state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8 0'): 1 2 3 4 0 6 7 5 8
Enter goal state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8 0'): 1 2 3 4 5 6 7 8 0
Enter maximum depth: 3
Searching at depth: 0
Searching at depth: 1
Current state:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]
Moves: ['Up']

Current state:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
Moves: ['Down']

Current state:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
Moves: ['Left']

Current state:
[1, 2, 3]
[4, 6, 0]
[7, 5, 8]
Moves: ['Right']
```

```
Current state:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
Moves: ['Down']

Current state:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
Moves: ['Down', 'Up']

Current state:
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
Moves: ['Down', 'Left']

Current state:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Moves: ['Down', 'Right']

Goal found at depth: 2
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Moves: ['Down', 'Right']
```

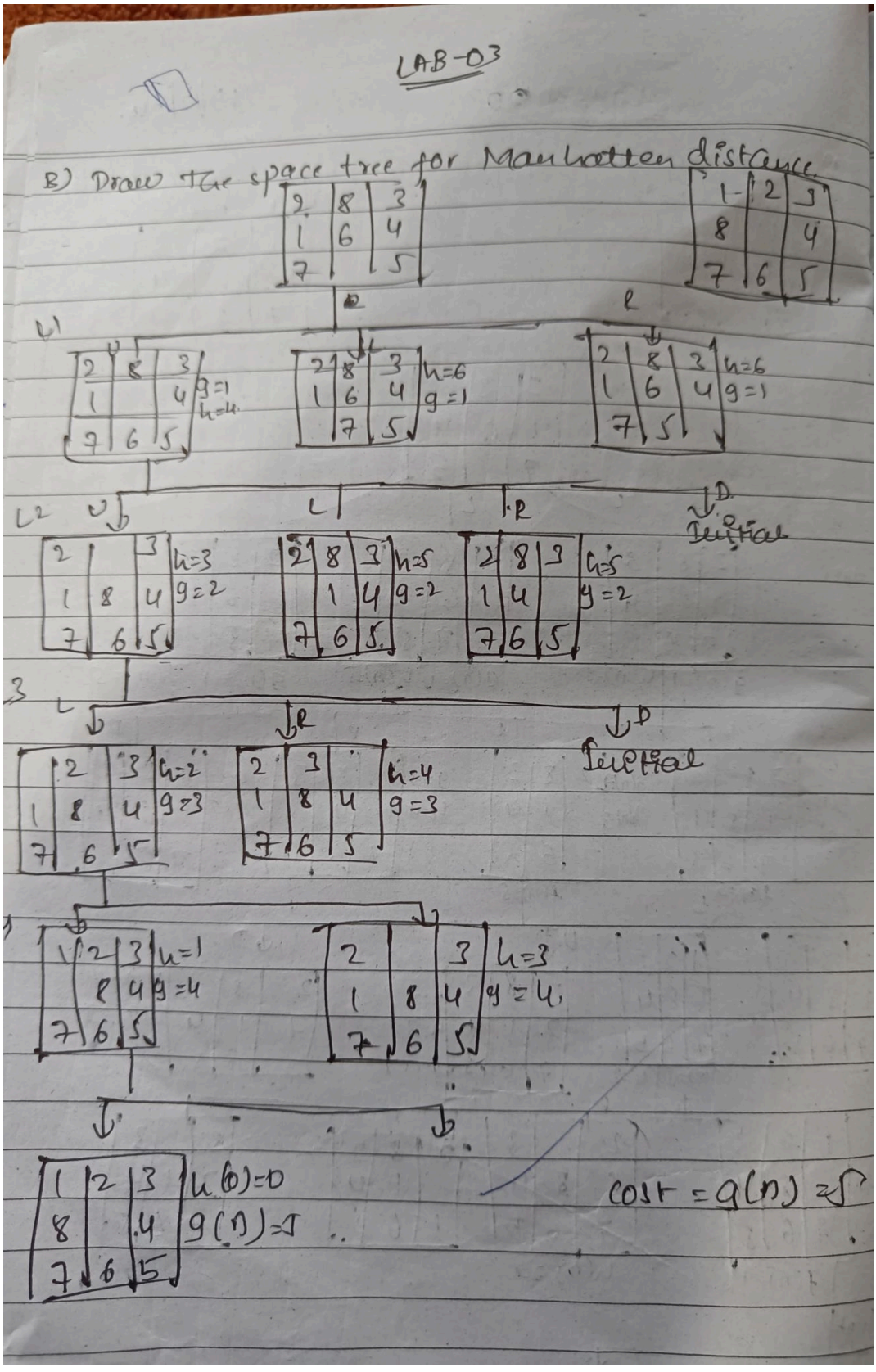
Program 3:
Implement A* search algorithm

Algorithm:



LAB-03

B) Draw the space tree for Manhatten distance.



Code :

```
import numpy as np
import heapq

class PuzzleState:
    def __init__(self, board, level, goal):
        self.board = board
        self.level = level
        self.goal = goal
        self.blank_pos = self.find_blank()
        self.cost = self.level + self.misplaced_tiles()

    def find_blank(self):
        return tuple(np.argwhere(self.board == 0)[0])

    def misplaced_tiles(self):
        return np.sum(self.board != self.goal) - (self.board[self.board == 0] !=
self.goal[self.goal == 0]).sum()

    def get_neighbors(self):
        neighbors = []
        x, y = self.blank_pos
        moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        move_names = ['Up', 'Down', 'Left', 'Right']
        for (dx, dy), move_name in zip(moves, move_names):
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = self.board.copy()
                new_board[x, y], new_board[new_x, new_y] = new_board[new_x,
new_y], new_board[x, y]
                neighbors.append((PuzzleState(new_board, self.level + 1, self.goal),
move_name))
        return neighbors

    def __lt__(self, other):
        return self.cost < other.cost

def print_board(state):
    print("\nCurrent State:")
    print(state.board)
    print(f"Misplaced Tiles: {state.misplaced_tiles()}\n")

def a_star(initial_state, goal_state):
    open_set = []
    closed_set = set()
    heapq.heappush(open_set, initial_state)
```

```
    print("Initial State:")
    print_board(initial_state)

    while open_set:
        current = heapq.heappop(open_set)
        if np.array_equal(current.board, current.goal):
            print("Goal state reached!")
            print(f"Total Cost: {current.cost}")
            print_board(current)
            return

        closed_set.add(tuple(map(tuple, current.board)))
        neighbors = current.get_neighbors()
        best_neighbor = None

        for neighbor, move_name in neighbors:
            if tuple(map(tuple, neighbor.board)) in closed_set:
                continue
            if best_neighbor is None or neighbor < best_neighbor[0]:
                best_neighbor = (neighbor, move_name)

        if best_neighbor:
            print(f"Moved {best_neighbor[1]}")
            print_board(best_neighbor[0])
            heapq.heappush(open_set, best_neighbor[0])

def main():
    print("Enter the initial state (3x3) as a single line of numbers (0 for blank):")
    initial_state_input = list(map(int, input().split()))
    initial_state = np.array(initial_state_input).reshape(3, 3)
    print("Enter the goal state (3x3) as a single line of numbers (0 for blank):")
    goal_state_input = list(map(int, input().split()))
    goal_state = np.array(goal_state_input).reshape(3, 3)
    initial_puzzle = PuzzleState(initial_state, 0, goal_state)
    a_star(initial_puzzle, goal_state)

if __name__ == "__main__":
    main()
```

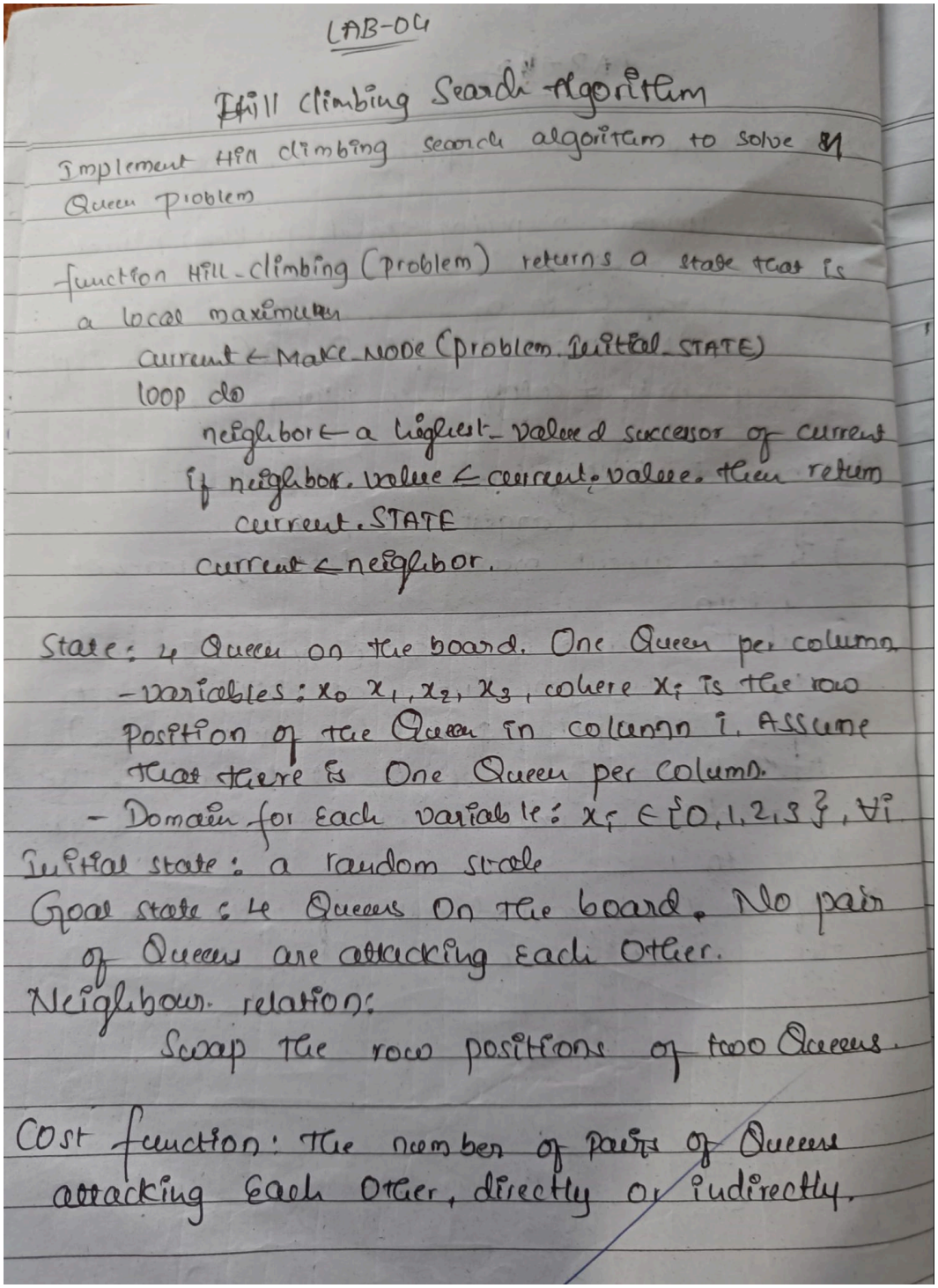
Output:

```
PS D:\python> py puzzleheuristic.py
Enter the initial state (3x3) as a single line of numbers (0 for blank):
1 2 3 0 4 6 7 5 8
Enter the goal state (3x3) as a single line of numbers (0 for blank):
1 2 3 4 5 6 7 8 0
Initial State:

Current State:
[[1 2 3]
 [0 4 6]
 [7 5 8]]
Misplaced Tiles: 4

Moved Right

Current State:
[[1 2 3]
 [4 0 6]
 [7 5 8]]
Misplaced Tiles: 3

Moved Down

Current State:
[[1 2 3]
 [4 5 6]
 [7 0 8]]
Misplaced Tiles: 2
```

```
Moved Right

Current State:
[[1 2 3]
 [4 5 6]
 [7 8 0]]
Misplaced Tiles: 0

Goal state reached!
Total Cost: 3

Current State:
[[1 2 3]
 [4 5 6]
 [7 8 0]]
Misplaced Tiles: 0
```

Program 4:
Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm :



LAB-04

Hill climbing Search Algorithm

Implement Hill climbing search algorithm to solve N Queen problem

function Hill-climbing (problem) returns a state that is a local maximum

current ← Make_Node (problem.Initial_STATE)

loop do

neighbor ← a highest_valued successor of current

if neighbor.value ≤ current.value. then return current.STATE

current ← neighbor.

State: 4 Queen on the board. One Queen per column

- Variables: x_0 x_1, x_2, x_3, where x_i is the row position of the Queen in column i. Assume that there is One Queen per column.
- Domain for each variable: $x_i \in \{0,1,2,3\}, \forall i$

Initial state: a random state

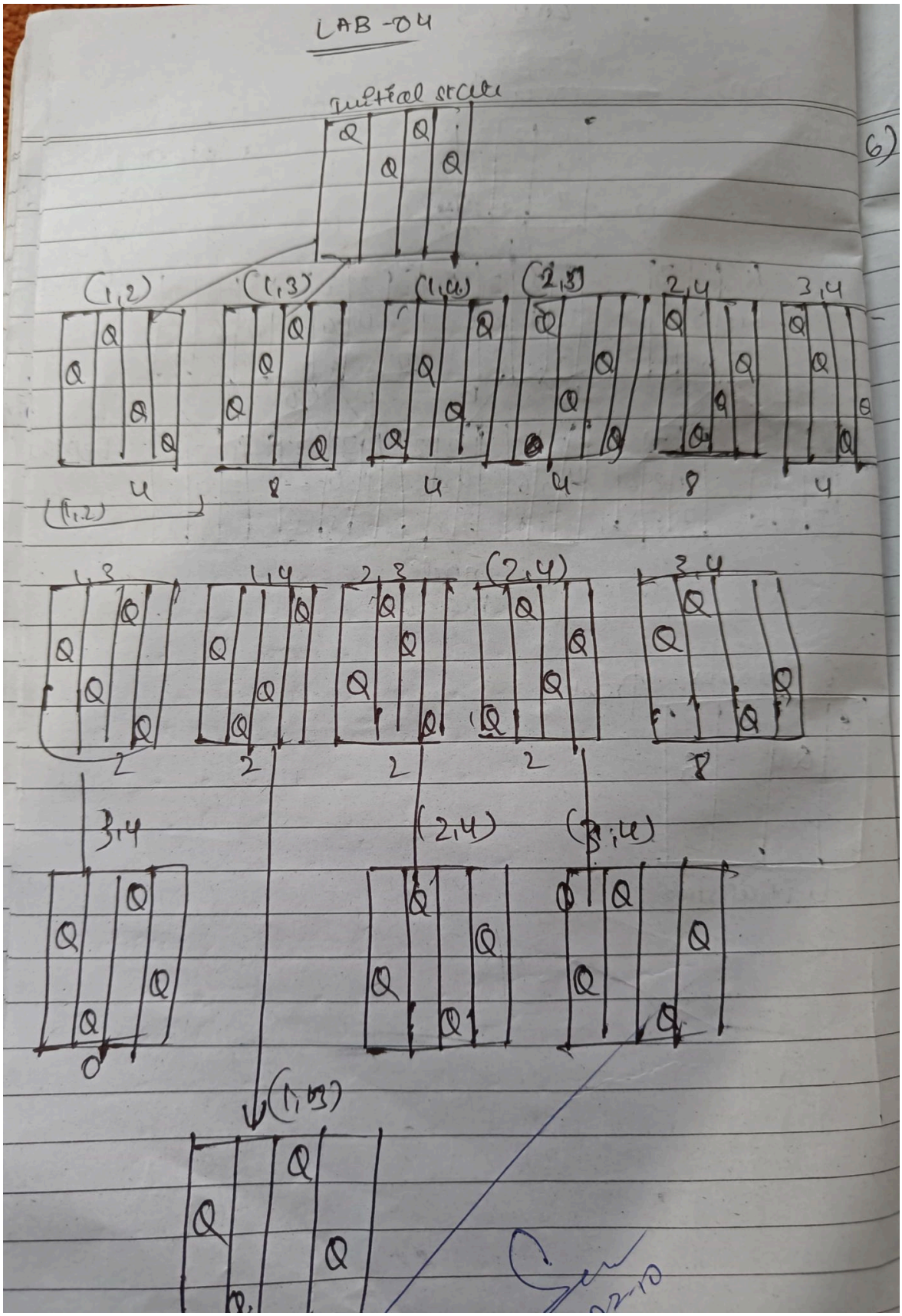
Goal state: 4 Queens On the board. No pair of Queens are attacking Each Other.

Neighbour relation:

Swap the row positions of two Queens.

Cost function: The number of pairs of Queens attacking Each Other, directly or indirectly.

LAB - 04


Initial state
6)
(1,2)
(1,3)
(1,4)
(2,3)
2,4
3,4
4
8
4
4
8
4
(1,2)
3
1,4
2,3
(2,4)
3,4
2
2
2
2
8
3,4
(2,4)
(3,4)
(1,3)

Code :

```
import random

def calculate_cost(board):
    n = len(board)
    attacks = 0
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j]:  # Same column
                attacks += 1
            if abs(board[i] - board[j]) == abs(i - j):  # Same diagonal
                attacks += 1
    return attacks

def get_neighbors(board):
    neighbors = []
    n = len(board)
    for col in range(n):
        for row in range(n):
            if row != board[col]:  # Only change the row of the queen
                new_board = board[:]
                new_board[col] = row
                neighbors.append(new_board)
    return neighbors

def hill_climb(board):
    current_cost = calculate_cost(board)
    print("Initial board configuration:")
    print_board(board, current_cost)
    iteration = 0
    while True:
        neighbors = get_neighbors(board)
        best_neighbor = None
        best_cost = current_cost
        for neighbor in neighbors:
            cost = calculate_cost(neighbor)
            if cost < best_cost:  # Looking for a lower cost
                best_cost = cost
                best_neighbor = neighbor
        if best_neighbor is None:  # No better neighbor found, we're done
            break
        board = best_neighbor
        current_cost = best_cost
        iteration += 1
        print(f"Iteration {iteration}:")
        print_board(board, current_cost)
```

```
    return board, current_cost

def print_board(board, cost):
    n = len(board)
    # Create an empty board
    display_board = [['.'] * n for _ in range(n)]
    # Place queens on the board
    for col in range(n):
        display_board[board[col]][col] = 'Q'
    # Print the board
    for row in range(n):
        print(' '.join(display_board[row]))
    print(f"Cost: {cost}\n")

if __name__ == "__main__":
    n = int(input("Enter the number of queens (N): "))  # User input for N
    initial_state = list(map(int, input(f"Enter the initial state (row numbers for each
column, space-separated): ").split()))
    if len(initial_state) != n or any(r < 0 or r >= n for r in initial_state):
        print("Invalid initial state. Please ensure it has N elements with values from 0 to
N-1.")
    else:
        solution, cost = hill_climb(initial_state)
        print(f"Final board configuration with cost {cost}:")
        print_board(solution, cost)
```

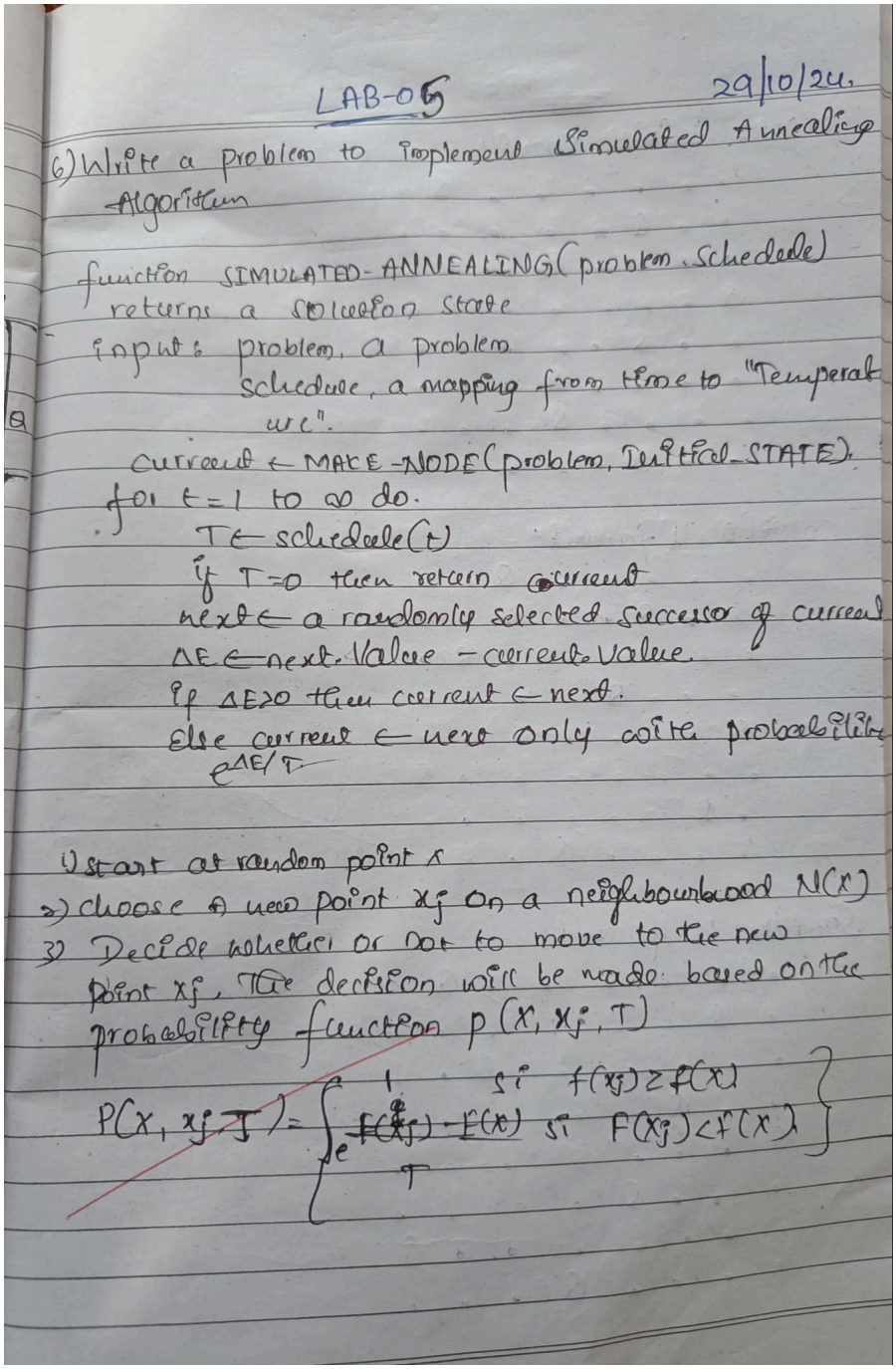
Output :

```
Enter the number of queens (N): 5
Enter the initial state (row numbers for each column, space-separated): 1 0 1 0 1
Initial board configuration:
. Q . Q .
Q . Q . Q
. . . . .
. . . . .
. . . . .
Cost: 8

Iteration 1:
. Q . Q .
Q . . . Q
. . Q . .
. . . . .
. . . . .
Cost: 4

Iteration 2:
. Q . Q .
. . . . Q
Q . Q . .
. . . . .
. . . . .
Cost: 3
```

```
Iteration 3:
. Q . . .
. . . . Q
Q . Q . .
. . . . .
. . . Q .
Cost: 1

Iteration 4:
. Q . . .
. . . . Q
. . Q . .
Q . . . .
. . . Q .
Cost: 0

Final board configuration with cost 0:
. Q . . .
. . . . Q
. . Q . .
Q . . . .
. . . Q .
Cost: 0
```

Program 5:
Simulated Annealing to Solve 8-Queens problem

Algorithm :



LAB-05 29/10/24.

6) Write a problem to Implement Simulated Annealing Algorithm

function SIMULATED-ANNEALING(problem, schedule)
returns a solution state
inputs: problem, a problem
schedule, a mapping from time to "Temperature".
current ← MAKE-NODE(problem, INITIAL-STATE).
for t=1 to ∞ do.
T ← schedule(t)
if T=0 then return current
next ← a randomly selected successor of current
ΔE ← next.Value - current.Value
If ΔE≥0 then current ← next.
Else current ← next only with probability $e^{\Delta E/T}$

1) Start at random point x
2) Choose a new point x_j on a neighbourhood N(x)
3) Decide whether or not to move to the new point x_j, The decision will be made based on the probability function p(x, x_j, T)

$$P(x, x_j, T) = \begin{cases} 1 & \text{si } f(x_j) \geq f(x) \\ e^{\frac{f(x_j) - f(x)}{T}} & \text{si } f(x_j) < f(x) \end{cases}$$

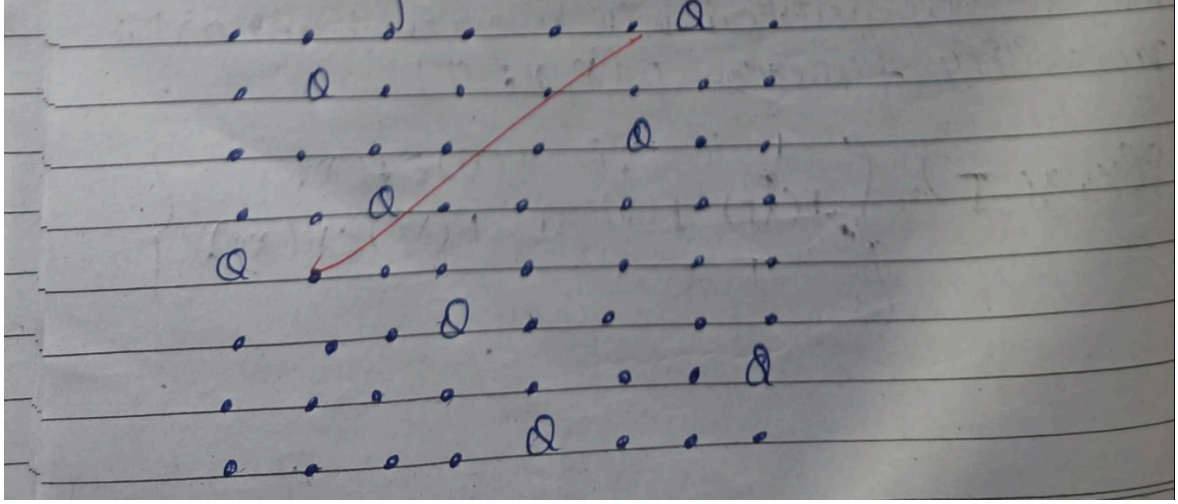
Output: Initial Board:

. . . . . Q .
. . Q . . . . .
. . . . Q . . .
Q . . . .
. . Q . . . . .
. . . . Q . . .
. . . . . Q . .
. . Q . . . . .

Solution 1 found:

. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. . . Q . . . .
Q . . . .
. . . . Q
. Q . . . . . .
. . . . Q . . .

Solution 2 found:



Travelling salesman problem using Stimulated Annealing.

Cities = [(0,0), (1,5), (5,1), (10,10), (10,5), (6,7), (3,8), (8,3), (2,6)]

Best tour: [(0,0), (1,5), (2,6), (3,8), (6,5), (10,10), (10,5), (8,3), (5,1)]

Best distance = 33

29/10/24

Code :

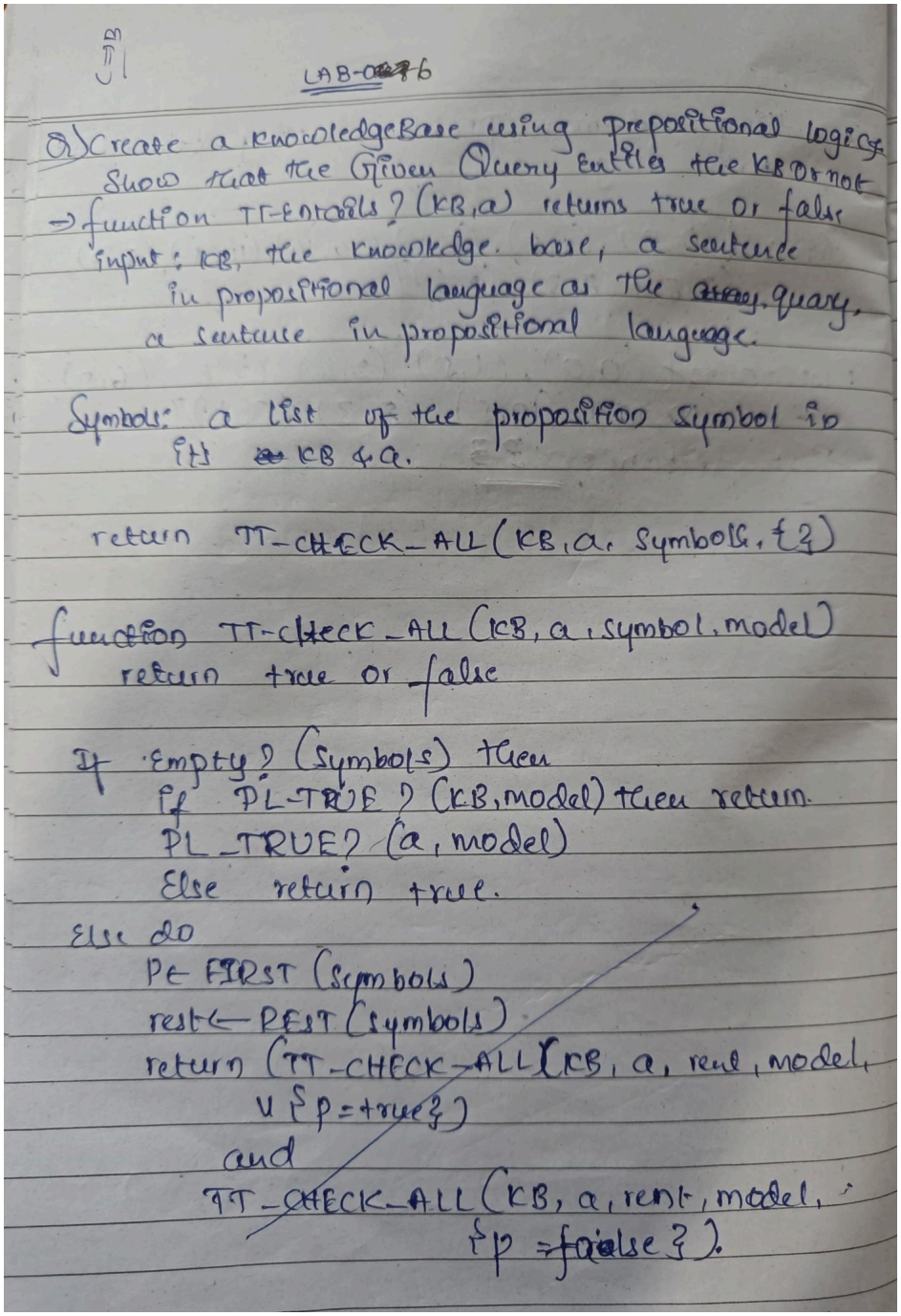
```
import mlrose_hiive as mlrose
import numpy as np

def queens_max(position):
    no_attack_on_j = 0
    queen_not_attacking = 0
    for i in range(len(position) - 1):
        no_attack_on_j = 0
        for j in range(i + 1, len(position)):
            if (position[j] != position[i]) and (position[j] != position[i] + (j - i)) and
(position[j] != position[i] - (j - i)):
                no_attack_on_j += 1
        if no_attack_on_j == len(position) - 1 - i:
            queen_not_attacking += 1
    if queen_not_attacking == 7:
        queen_not_attacking += 1
    return queen_not_attacking

def print_board(position):
    size = len(position)
    board = np.full((size, size), '.')
    for row, col in enumerate(position):
        board[row, col] = 'Q'
    print('\n'.join([' '.join(row) for row in board]))

objective = mlrose.CustomFitness(queens_max)
problem = mlrose.DiscreteOpt(length=8, fitness_fn=objective, maximize=True,
max_val=8)
T = mlrose.ExpDecay()

initial_position = np.array([4, 6, 1, 5, 2, 0, 3, 7])
best_position, best_objective, fitness_curve = mlrose.simulated_annealing(
    problem=problem, schedule=T, max_attempts=500, init_state=initial_position
)

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

Output :

```
The best position found is: [2 5 7 0 3 6 4 1]
The number of queens that are not attacking each other is: 8.0
Board representation:
. . Q . . . . .
. . . . . Q . .
. . . . . . . Q
Q . . . . . . .
. . . Q . . . .
. . . . . Q .
. . . . Q . . .
. Q . . . . . .
```

Program 6 :
Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

Algorithm :



Truth Table

$\alpha = A \vee B$

$KB = (A \vee C) \wedge (B \vee \neg C)$

A	B	C	$A \vee C$	$B \vee \neg C$	KB	α
false	false	false	false	true	false	false
false	false	true	true	false	false	false
false	true	false	false	true	false	true
false	true	true	true	true	true	true
true	false	false	true	true	true	true
true	false	true	true	false	false	true
true	true	false	true	true	true	true
true	true	true	true	true	true	true

8×8

$2 \mid 11 \mid 2^4$

Code :

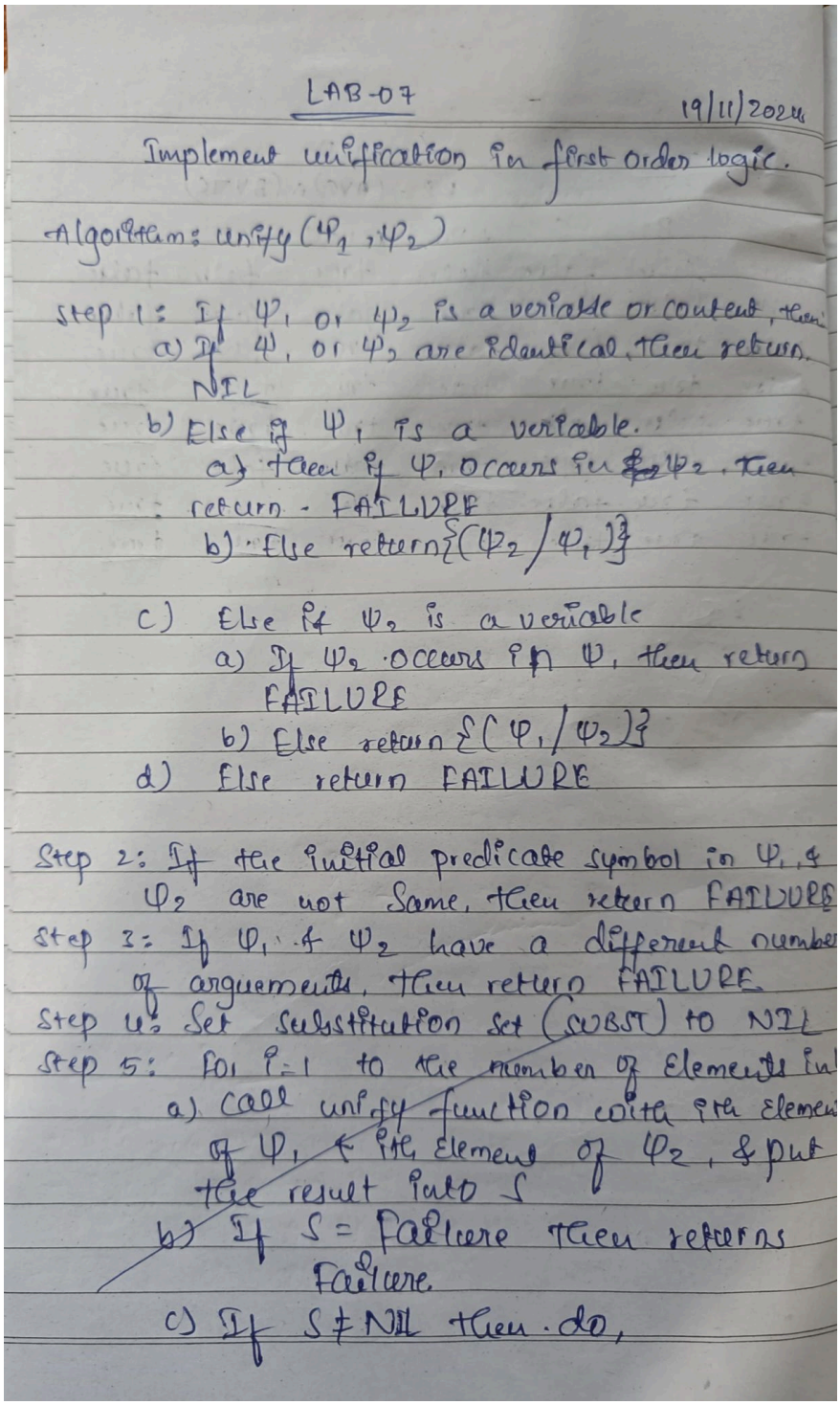
```
import pandas as pd

truth_values = [
    (False, False, False),
    (False, False, True),
    (False, True, False),
    (False, True, True),
    (True, False, False),
    (True, False, True),
    (True, True, False),
    (True, True, True)
]

table = pd.DataFrame(truth_values, columns=["A", "B", "C"])
table["A or C"] = table["A"] | table["C"]  # A ∨ C
table["B or not C"] = table["B"] | ~table["C"]  # B ∨ ¬C
table["KB"] = table["A or C"] & table["B or not C"]
table["Alpha (α)"] = table["A"] | table["B"]

def highlight_rows(row):
    if row["KB"] and row["Alpha (α)"]:
        return ['background-color: green'] * len(row)
    else:
        return [''] * len(row)

styled_table = table.style.apply(highlight_rows, axis=1)
styled_table
```

Output :

	A	B	C	A or C	B or not C	KB	Alpha (α)
0	False	False	False	False	True	False	False
1	False	False	True	True	False	False	False
2	False	True	False	False	True	False	True
3	False	True	True	True	True	True	True
4	True	False	False	True	True	True	True
5	True	False	True	True	False	False	True
6	True	True	False	True	True	True	True
7	True	True	True	True	True	True	True

Program 7 :
Implement unification in first order logic

Algorithm :



LAB-07 19/11/2024

Implement unification in first order logic.

Algorithm: unify(Ψ_1, Ψ_2)

Step 1: If Ψ_1 or Ψ_2 is a variable or content, then
a) If Ψ_1 or Ψ_2 are identical, then return NIL
b) Else if Ψ_1 is a variable.
a) then if Ψ_1 occurs in Ψ_2, then return FAILURE
b) Else return $\{(\Psi_2 / \Psi_1)\}$
c) Else if Ψ_2 is a variable
a) If Ψ_2 occurs in Ψ_1, then return FAILURE
b) Else return $\{(\Psi_1 / \Psi_2)\}$
d) Else return FAILURE

Step 2: If the initial predicate symbol in Ψ_1 & Ψ_2 are not same, then return FAILURE
Step 3: If Ψ_1 & Ψ_2 have a different number of arguments, then return FAILURE
Step 4: Set substitution set (SUBST) to NIL
Step 5: For i=1 to the number of elements in
a) call unify function with ith element of Ψ_1 & ith element of Ψ_2, & put the result into S
b) If S = failure then returns Failure.
c) If S ≠ NIL then do,

a) Apply S to the remainder of both L1 & L2
b) SUBST = Append (S, SUBST).

Step 6: Return SUBST.

Ex: P(x, h(y))
P(a, f(z))

Output: unification successful.

~~Unify (L1, L2)~~

2) Expression 1 : P(b, x, f(g(z))
Expression 2 : P(z, f(y), f(y)).

Result : unification successful.

3) Expression 1 : P(b, x, f(g(z))
Expression 2 : Q(b, x, f(g(z))

Result : unification failed.

19/1/24

Code :

```
import re

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

def unify_var(var, x, subst):
    """Handles unification of a variable with another term."""
    if var in subst:  # If var is already substituted
        return unify(subst[var], x, subst)
    elif isinstance(x, (list, tuple)) and tuple(x) in subst:  # Handle compound
expressions
        return unify(var, subst[tuple(x)], subst)
    elif occurs_check(var, x):  # Check for circular references
        return "FAILURE"
    else:
        # Add the substitution to the set (convert list to tuple for hashability)
        subst[var] = tuple(x) if isinstance(x, list) else x
        return subst

def unify(x, y, subst=None):
    """
    Unifies two expressions x and y and returns the substitution set if they can be
unified.
    Returns 'FAILURE' if unification is not possible.
    """
    if subst is None:
        subst = {}  # Initialize an empty substitution set
    if x == y:  # If x and y are identical
        return subst
    elif isinstance(x, str) and x.islower():  # If x is a variable
        return unify_var(x, y, subst)
    elif isinstance(y, str) and y.islower():  # If y is a variable
        return unify_var(y, x, subst)
    elif isinstance(x, list) and isinstance(y, list):  # If x and y are compound
expressions (lists)
        if len(x) != len(y):  # Step 3: Different number of arguments
            return "FAILURE"
        if x[0] != y[0]:  # Step 2: Check if the predicate symbols (the first element)
```

```
match
            return "FAILURE"
        for xi, yi in zip(x[1:], y[1:]):  # Step 5: Recursively unify each argument
            subst = unify(xi, yi, subst)
            if subst == "FAILURE":
                return "FAILURE"
        return subst
    else:  # If x and y are different constants or non-unifiable structures
        return "FAILURE"

def unify_and_check(expr1, expr2):
    """
    Attempts to unify two expressions and returns a tuple:
    (is_unified: bool, substitutions: dict or None)
    """
    result = unify(expr1, expr2)
    if result == "FAILURE":
        return False, None
    return True, result

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

def parse_input(input_str):
    """Parses a string input into a structure that can be processed by the unification
algorithm."""
    input_str = input_str.replace(" ", "")  # Remove spaces
    def parse_term(term):
        if '(' in term:
            match = re.match(r'([a-zA-Z0-9_]+)\((.*)\)', term)
            if match:
                predicate = match.group(1)
                arguments_str = match.group(2)
                arguments = [parse_term(arg.strip()) for arg in arguments_str.split(',')]
                return [predicate] + arguments
        return term
    return parse_term(input_str)

def main():
    while True:
```

```
        expr1_input = input("Enter the first expression (e.g., p(x, f(y))): ")
        expr2_input = input("Enter the second expression (e.g., p(a, f(z))): ")
        expr1 = parse_input(expr1_input)
        expr2 = parse_input(expr2_input)
        is_unified, result = unify_and_check(expr1, expr2)
        display_result(expr1, expr2, is_unified, result)
        another_test = input("Do you want to test another pair of expressions?
(yes/no): ").strip().lower()
        if another_test != 'yes':
            break

if __name__ == "__main__":
    main()
```

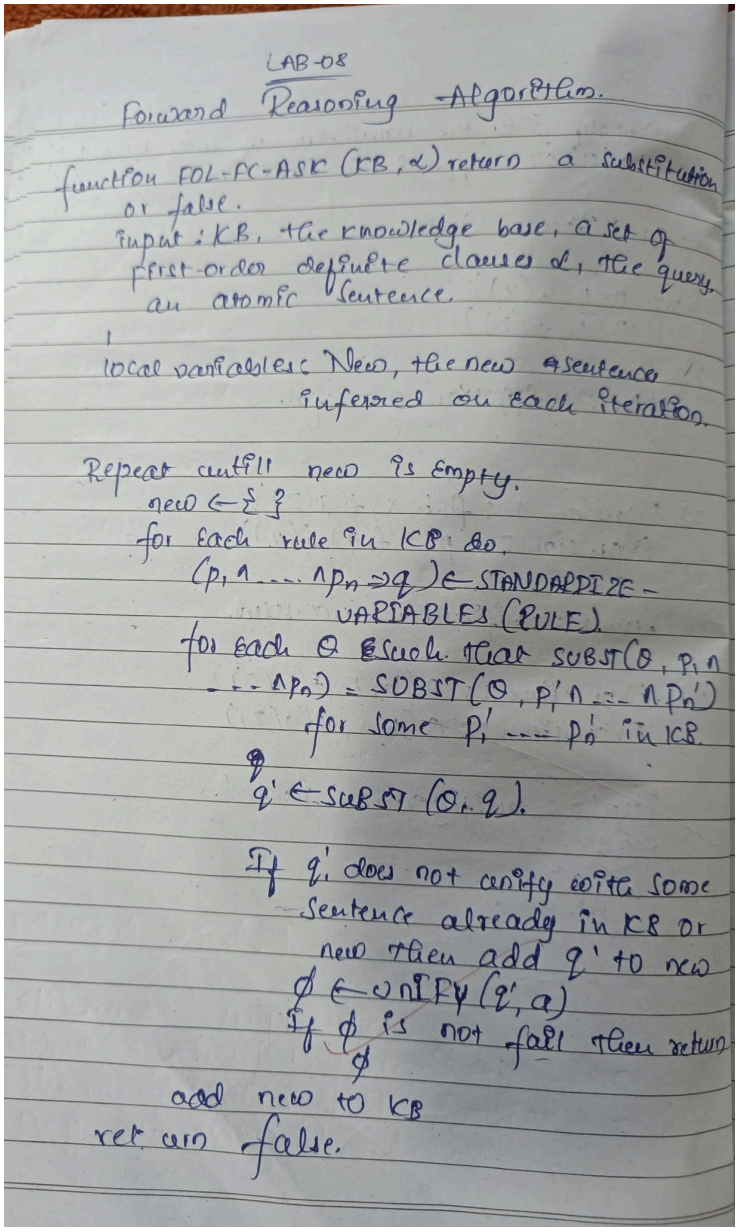
Output :

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

Program 8:

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

Algorithm :



LAB-08

Forward Reasoning Algorithm.

function FOL-FC-ASK (KB, α) return a substitution or false.

input : KB, the knowledge base, a set of first-order definite clauses α, the query, an atomic sentence.

local variables: New, the new sentences inferred on each iteration.

Repeat until new is empty.

new ← { }

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(θ, $p_1 \wedge \dots \wedge p_n$) = SUBST(θ, $p_1' \wedge \dots \wedge p_n'$)

for some $p_1' \dots p_n'$ in KB.

$q' \leftarrow$ SUBST (θ, q).

If q' does not unify with some sentence already in 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.

Ex: Parent (John, mary)
Parent (mary, anna)

→ Parent (x, y) ∧ parent (y, z) → grandparent (x, z)

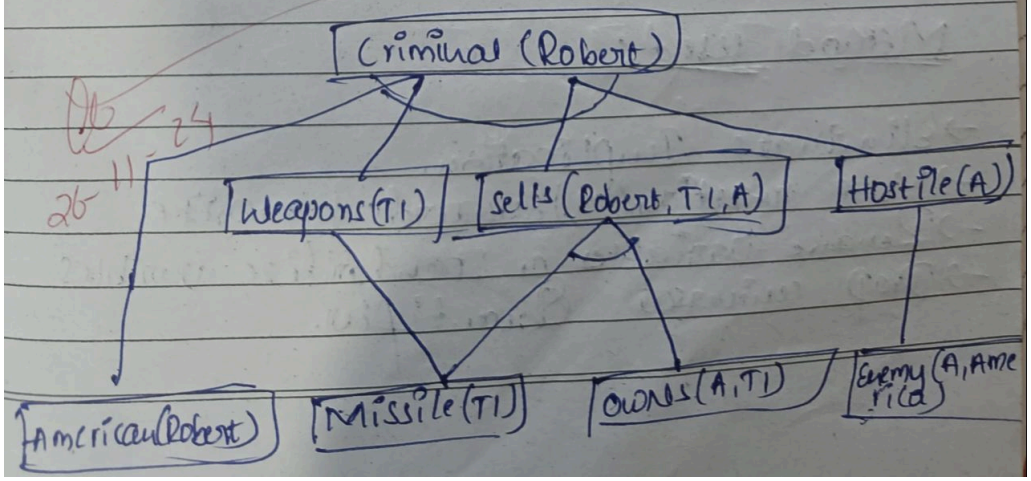
Ex: Human (socrates)
∀x (Human (x) → Mortal (x))

Query: Mortal (socrates).

output:

~~Query ('c'):~~
KB: [{"A": "x"}, {"B": "x"}, {"C": "x"}]
{"A": "John"},
{"B": "John"}
]
Query = {"C": "John"}

output:
{'c': "John"} is TRUE with substitution {'c': 'John'}.



Code :

```
facts = {
    "American(Robert)": True,
    "Missile(T1)": True,
    "Enemy(A, America)": True,
    "Owns(A, T1)": True,
    "Hostile(A)": False,
    "Weapon(T1)": False,
    "Sells(Robert, T1, A)": False,
    "Criminal(Robert)": False,
}

rules = [
    ("American(Robert) and Weapon(T1) and Sells(Robert, T1, A) and Hostile(A)",
"Criminal(Robert)"),
    ("Owns(A, T1) and Missile(T1)", "Weapon(T1)"),
    ("Missile(T1) and Owns(A, T1)", "Sells(Robert, T1, A)"),
    ("Enemy(A, America)", "Hostile(A)"),
]

def check_fact(fact):
    return facts.get(fact, False)

def parse_condition(condition):
    return condition.split(" and ")

def forward_reasoning():
    new_inferences = True
    while new_inferences:
        new_inferences = False
        for condition, conclusion in rules:
            condition_facts = parse_condition(condition)
            if all(check_fact(fact) for fact in condition_facts):
                if not check_fact(conclusion):
                    facts[conclusion] = True
                    new_inferences = True
                    print(f"Inferred: {conclusion}")

def print_inferred_facts():
    forward_reasoning()
    print("\nFinal Inferred Facts:")
    for fact, value in facts.items():
        print(f"{fact} is {'TRUE' if value else 'FALSE'}")

print_inferred_facts()
```

Output :

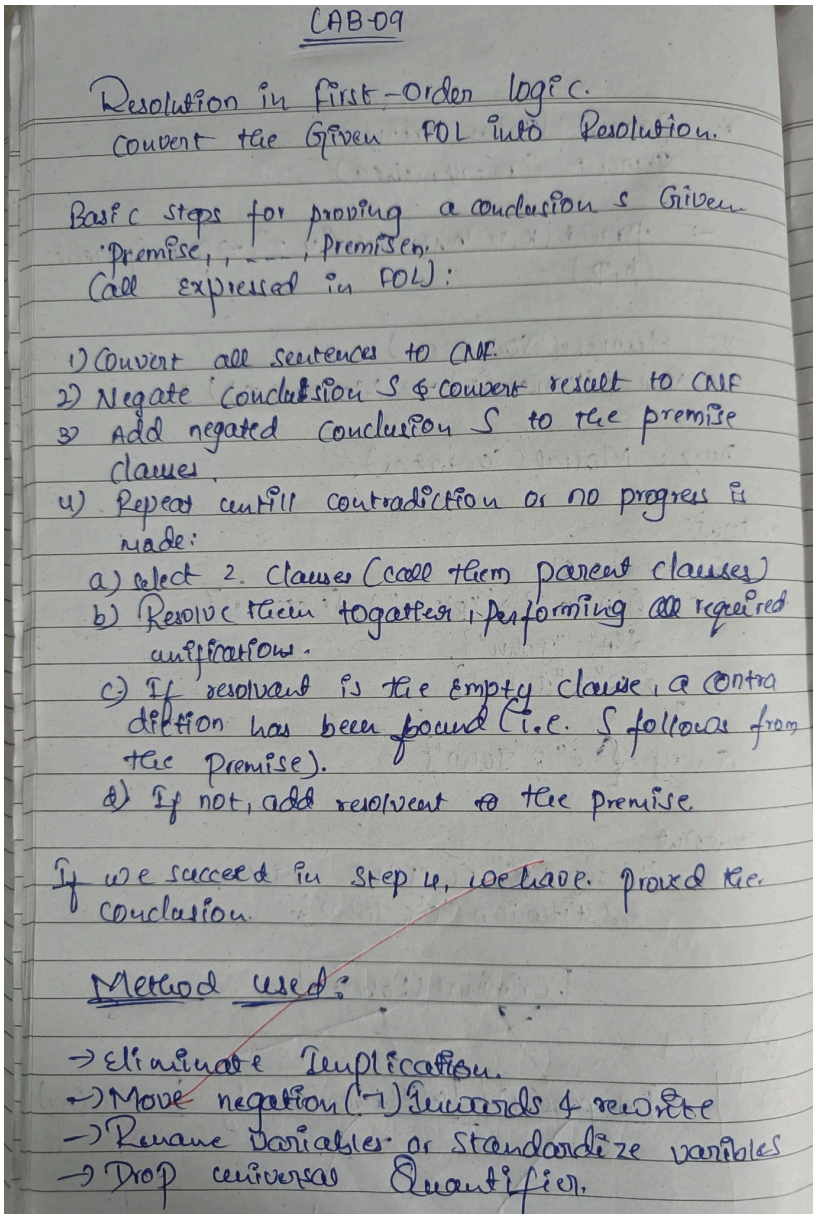
```
Inferred: Weapon(T1)
Inferred: Sells(Robert, T1, A)
Inferred: Hostile(A)
Inferred: Criminal(Robert)

Final Inferred Facts:
American(Robert) is TRUE
Missile(T1) is TRUE
Enemy(A, America) is TRUE
Owns(A, T1) is TRUE
Hostile(A) is TRUE
Weapon(T1) is TRUE
Sells(Robert, T1, A) is TRUE
Criminal(Robert) is TRUE
```

```
Inferred: Weapon(T1)
Inferred: Sells(Robert, T1, A)

Final Inferred Facts:
American(Robert) is TRUE
Missile(T1) is TRUE
Enemy(A, America) is FALSE
Owns(A, T1) is TRUE
Hostile(A) is FALSE
Weapon(T1) is TRUE
Sells(Robert, T1, A) is TRUE
Criminal(Robert) is FALSE
```

Program 9 :

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

Algorithm :



LAB-09

Resolution in first-order logic.
Convert the Given FOL into Resolution.

Basic steps for proving a conclusion S Given Premise$_1$, ..., Premise$_n$.
(All expressed in FOL):

1) Convert all sentences to CNF.
2) Negate conclusion S & 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 premise).
   d) If not, add resolvent to the premise.

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

Method used:

→ Eliminate Implications.
→ Move negation (¬) inwards & rewrite
→ Rename variables or Standardize variables
→ Drop universal Quantifier.

Representation in FOL.

a) John likes all kind of food — a) $\forall x : food(x) \rightarrow likes(John, x)$

b) Apple & vegetables are food — b) $food(apple) \wedge food(vegetables)$

c) Anything Anyone eats & not killed is food — c) $\forall x \forall y. eats(x, y) \wedge \neg killed(x) \rightarrow food(y)$

d) Anil Eats ~~Everything~~ Peanuts & still alive — d) $eats(Anil, Peanuts) \wedge alive(Anil)$

e) Harry Eats Everything that Anil Eats — e) $\forall x : eats(Anil, x) \rightarrow eats(Harry, x)$

f) Anyone who is alive implies not killed — f) ~~$\forall x : alive(x) \rightarrow \neg killed(x)$~~ $\forall x : \neg killed(x) \rightarrow alive(x)$

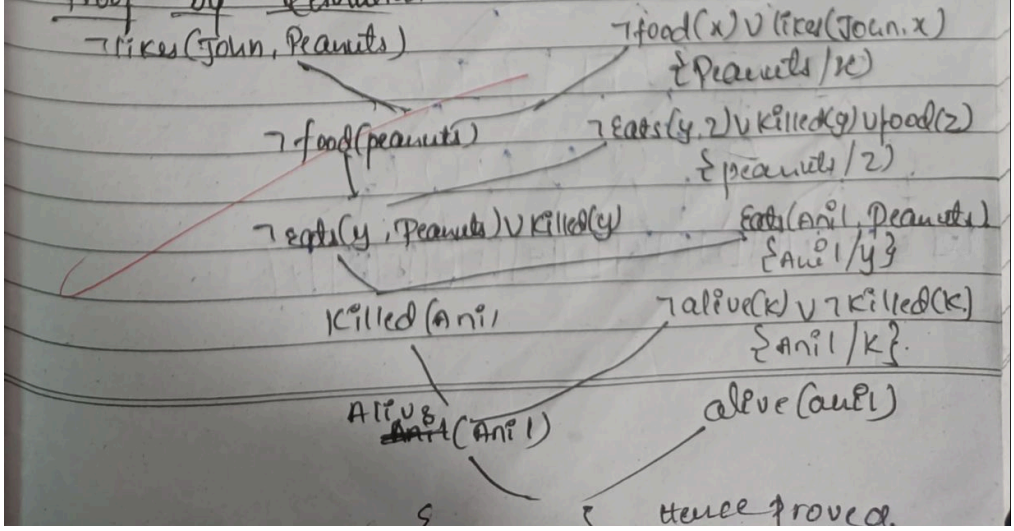
g) Anyone who is not killed implies alive — g) $\forall x : Alive(x) \rightarrow \neg killed(x)$

Prove by resolution that: h) likes(John, Peanuts)

h) John likes peanuts

Output: Does John likes peanuts? yes, proven by resolution

Proof by Resolution



Code :

```
from sympy import symbols, And, Or, Not, Implies, to_cnf

# Define constants (entities in the problem)
John, Anil, Harry, Apple, Vegetables, Peanuts, x, y = symbols('John Anil Harry Apple Vegetables
Peanuts x y')

# Define predicates as symbols (this works as a workaround)
Food = symbols('Food')
Eats = symbols('Eats')
Likes = symbols('Likes')
Alive = symbols('Alive')
Killed = symbols('Killed')

# Knowledge Base (Premises) in First-Order Logic
premises = [
    # 1. John likes all kinds of food: Food(x) → Likes(John, x)
    Implies(Food, Likes),

    # 2. Apples and vegetables are food: Food(Apple) ∧ Food(Vegetables)
    And(Food, Food),

    # 3. Anything anyone eats and is not killed is food: (Eats(y, x) ∧ ¬Killed(y)) → Food(x)
    Implies(And(Eats, Not(Killed)), Food),

    # 4. Anil eats peanuts and is still alive: Eats(Anil, Peanuts) ∧ Alive(Anil)
    And(Eats, Alive),

    # 5. Harry eats everything that Anil eats: Eats(Anil, x) → Eats(Harry, x)
    Implies(Eats, Eats),

    # 6. Anyone who is alive implies not killed: Alive(x) → ¬Killed(x)
    Implies(Alive, Not(Killed)),

    # 7. Anyone who is not killed implies alive: ¬Killed(x) → Alive(x)
    Implies(Not(Killed), Alive),
]

# Negated conclusion to prove: ¬Likes(John, Peanuts)
negated_conclusion = Not(Likes)

# Convert all premises and the negated conclusion to Conjunctive Normal Form (CNF)
cnf_clauses = [to_cnf(premise, simplify=True) for premise in premises]
cnf_clauses.append(to_cnf(negated_conclusion, simplify=True))
```

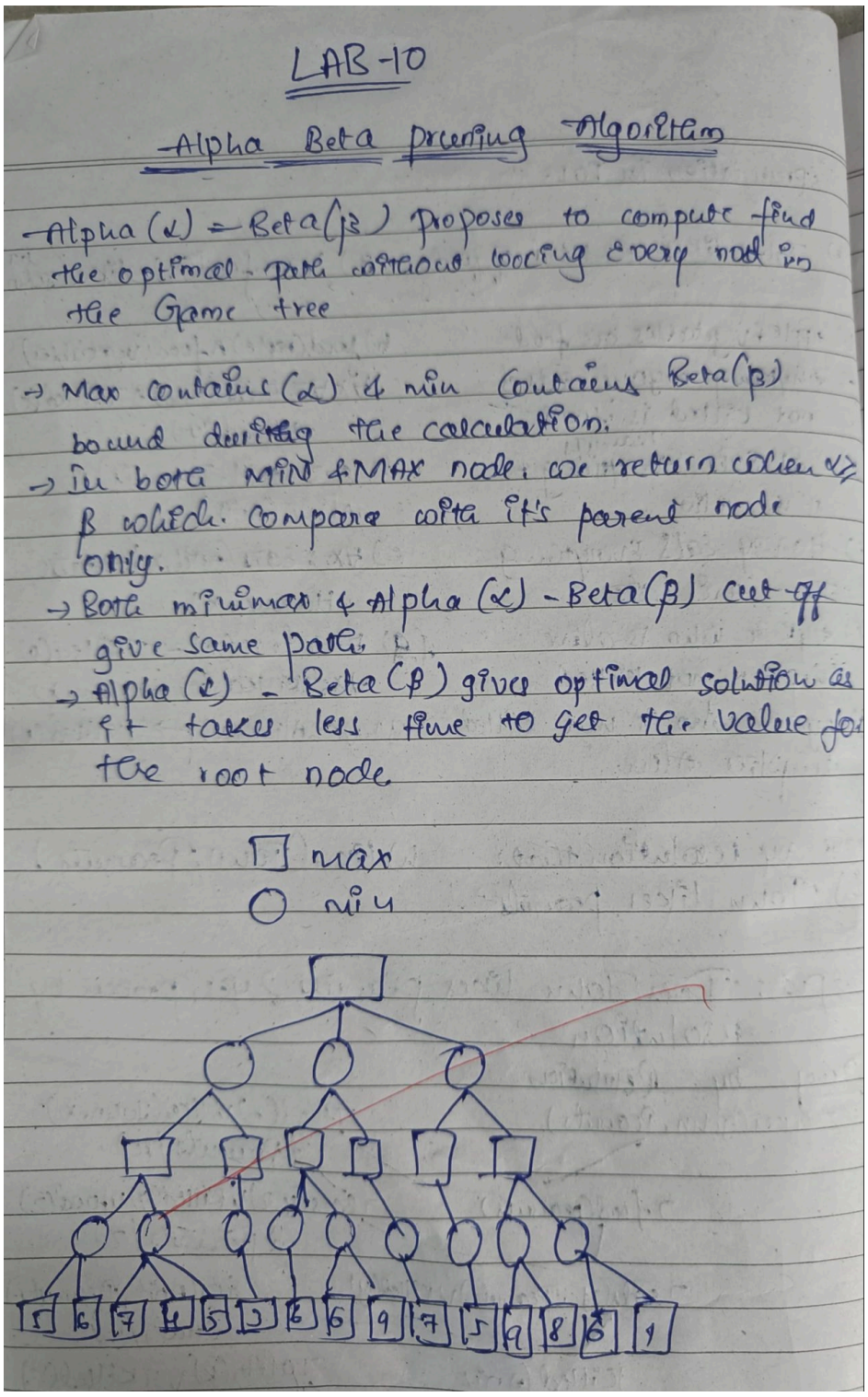
```
# Function to resolve two clauses
def resolve(clause1, clause2):
    """
    Resolve two CNF clauses to produce resolvents.
    """
    clause1_literals = clause1.args if isinstance(clause1, Or) else [clause1]
    clause2_literals = clause2.args if isinstance(clause2, Or) else [clause2]
    resolvents = []
    for literal in clause1_literals:
        if Not(literal) in clause2_literals:
            # Remove the literal and its negation and combine the rest
            new_clause = Or(
                *[l for l in clause1_literals if l != literal],
                *[l for l in clause2_literals if l != Not(literal)]
            ).simplify()
            resolvents.append(new_clause)
    return resolvents

# Function to perform resolution on the set of CNF clauses
def resolution(cnf_clauses):
    """
    Perform resolution on CNF clauses to check for a contradiction.
    """
    clauses = set(cnf_clauses)
    new_clauses = set()
    while True:
        clause_list = list(clauses)
        for i in range(len(clause_list)):
            for j in range(i + 1, len(clause_list)):
                resolvents = resolve(clause_list[i], clause_list[j])
                if False in resolvents:  # Empty clause found
                    return True  # Contradiction found; proof succeeded
                new_clauses.update(resolvents)
        if new_clauses.issubset(clauses):  # No new information
            return False  # No contradiction; proof failed
        clauses.update(new_clauses)

# Perform resolution to check if the conclusion follows
result = resolution(cnf_clauses)
print("Does John like peanuts? ", "Yes, proven by resolution." if result else "No, cannot be
proven.")
```

OutPut :

```
Does John like peanuts?  Yes, proven by resolution.
```

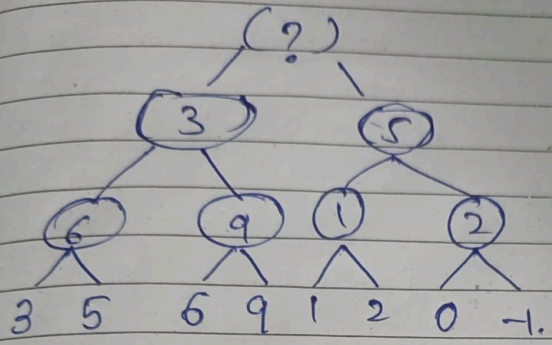
Program 10:
Implement Alpha-Beta Pruning.

Algorithm :

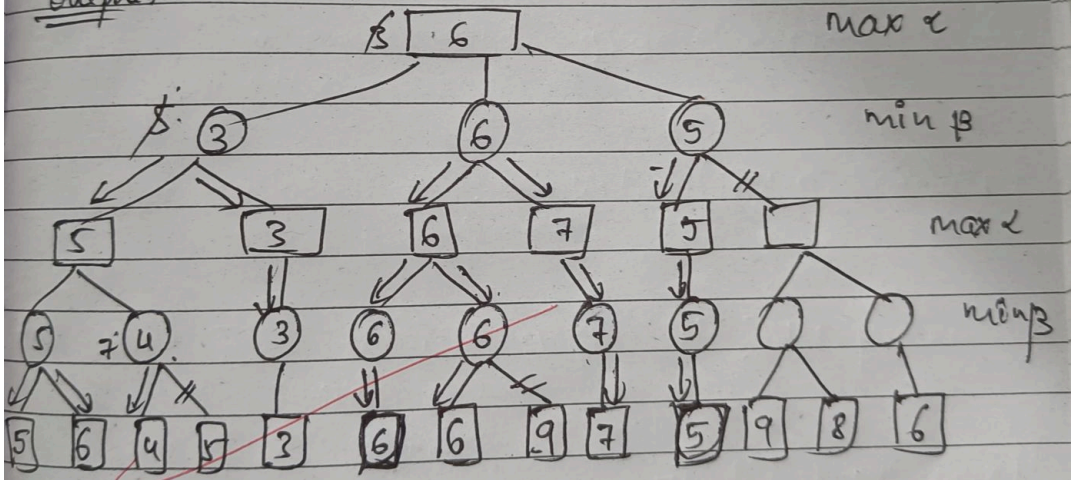


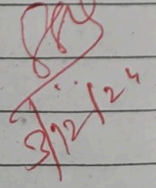
Output: The optimal value is : 5

Tree Structure



Output:





Code :

```
# Python3 program to demonstrate
# working of Alpha-Beta Pruning with detailed step output

# Initial values of Alpha and Beta
MAX, MIN = 1000, -1000

# Returns optimal value for the current player
def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):
    # Terminating condition: leaf node is reached
    if depth == 3:
        print(f"Leaf node reached: Depth={depth}, NodeIndex={nodeIndex},
Value={values[nodeIndex]}")
        return values[nodeIndex]

    if maximizingPlayer:
        best = MIN
        print(f"Maximizer: Depth={depth}, NodeIndex={nodeIndex}, Alpha={alpha},
Beta={beta}")

        # Recur for left and right children
        for i in range(2):
            val = minimax(depth + 1, nodeIndex * 2 + i, False, values, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)
            print(f"Maximizer updated: Depth={depth}, NodeIndex={nodeIndex}, Best={best},
Alpha={alpha}, Beta={beta}")

            # Alpha Beta Pruning
            if beta <= alpha:
                print(f"Maximizer Pruned: Depth={depth}, NodeIndex={nodeIndex}, Alpha={alpha},
Beta={beta}")
                break
        return best
    else:
        best = MAX
        print(f"Minimizer: Depth={depth}, NodeIndex={nodeIndex}, Alpha={alpha}, Beta={beta}")

        # Recur for left and right children
        for i in range(2):
            val = minimax(depth + 1, nodeIndex * 2 + i, True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)
            print(f"Minimizer updated: Depth={depth}, NodeIndex={nodeIndex}, Best={best},
Alpha={alpha}, Beta={beta}")
```

```
            # Alpha Beta Pruning
            if beta <= alpha:
                print(f"Minimizer Pruned: Depth={depth}, NodeIndex={nodeIndex}, Alpha={alpha},
Beta={beta}")
                break
        return best

# Driver Code
if __name__ == "__main__":
    values = [3, 5, 6, 9, 1, 2, 0, -1]  # Leaf node values
    print("Starting Alpha-Beta Pruning...")
    optimal_value = minimax(0, 0, True, values, MIN, MAX)
    print(f"\nThe optimal value is: {optimal_value}")
```

Output :

```
Starting Alpha-Beta Pruning...
Maximizer: Depth=0, NodeIndex=0, Alpha=-1000, Beta=1000
Minimizer: Depth=1, NodeIndex=0, Alpha=-1000, Beta=1000
Maximizer: Depth=2, NodeIndex=0, Alpha=-1000, Beta=1000
Leaf node reached: Depth=3, NodeIndex=0, Value=3
Maximizer updated: Depth=2, NodeIndex=0, Best=3, Alpha=3, Beta=1000
Leaf node reached: Depth=3, NodeIndex=1, Value=5
Maximizer updated: Depth=2, NodeIndex=0, Best=5, Alpha=5, Beta=1000
Minimizer updated: Depth=1, NodeIndex=0, Best=5, Alpha=-1000, Beta=5
Maximizer: Depth=2, NodeIndex=1, Alpha=-1000, Beta=5
Leaf node reached: Depth=3, NodeIndex=2, Value=6
Maximizer updated: Depth=2, NodeIndex=1, Best=6, Alpha=6, Beta=5
Maximizer Pruned: Depth=2, NodeIndex=1, Alpha=6, Beta=5
Minimizer updated: Depth=1, NodeIndex=0, Best=5, Alpha=-1000, Beta=5
Maximizer updated: Depth=0, NodeIndex=0, Best=5, Alpha=5, Beta=1000
Minimizer: Depth=1, NodeIndex=1, Alpha=5, Beta=1000
Maximizer: Depth=2, NodeIndex=2, Alpha=5, Beta=1000
Leaf node reached: Depth=3, NodeIndex=4, Value=1
Maximizer updated: Depth=2, NodeIndex=2, Best=1, Alpha=5, Beta=1000
Leaf node reached: Depth=3, NodeIndex=5, Value=2
Maximizer updated: Depth=2, NodeIndex=2, Best=2, Alpha=5, Beta=1000
Minimizer updated: Depth=1, NodeIndex=1, Best=2, Alpha=5, Beta=2
Minimizer Pruned: Depth=1, NodeIndex=1, Alpha=5, Beta=2
Maximizer updated: Depth=0, NodeIndex=0, Best=5, Alpha=5, Beta=1000

The optimal value is: 5
```