Artificial Intelligence Lab Report

Submitted by

Varsha Prasanth(1BM22CS321)
Batch: 3F

Course: Artificial Intelligence
Course Code: 23CS5PCAIN
Sem & Section: 5F

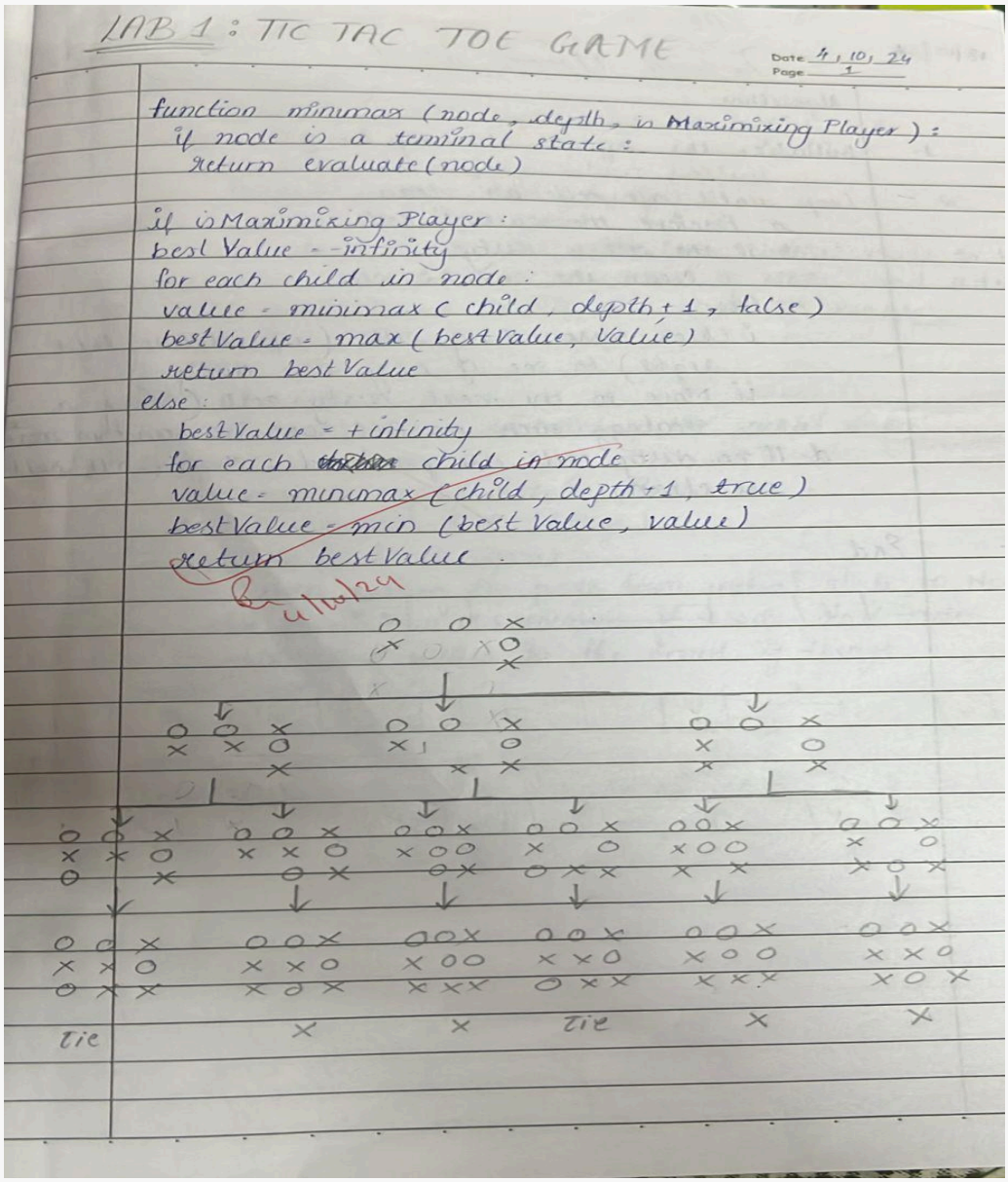
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B. M. S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
2022-2023

Table of contents

Program 1 - Tic Tac toe

Algorithm



Code

```
board = {1: ' ', 2: ' ', 3: ' ',
         4: ' ', 5: ' ', 6: ' ',
         7: ' ', 8: ' ', 9: ' '}


def printBoard(board):
    print(board[1] + '|' + board[2] + '|' + board[3])
    print('-+-+-')
    print(board[4] + '|' + board[5] + '|' + board[6])
    print('-+-+-')
    print(board[7] + '|' + board[8] + '|' + board[9])
    print('\n')


def spaceFree(pos):
    return board[pos] == ' '


def checkWin():
    win_conditions = [
        (1, 2, 3), (4, 5, 6), (7, 8, 9),  # Rows
        (1, 4, 7), (2, 5, 8), (3, 6, 9),  # Columns
        (1, 5, 9), (3, 5, 7)  # Diagonals
    ]
    for a, b, c in win_conditions:
        if board[a] == board[b] == board[c] and board[a] != ' ':
            return True
```

```
    return False


def checkMoveForWin(move):
    win_conditions = [
        (1, 2, 3), (4, 5, 6), (7, 8, 9),
        (1, 4, 7), (2, 5, 8), (3, 6, 9),
        (1, 5, 9), (3, 5, 7)
    ]
    for a, b, c in win_conditions:
        if board[a] == board[b] == move and board[a] != ' ':
            return True
    return False


def checkDraw():
    return all(board[key] != ' ' for key in board.keys())


def insertLetter(letter, position):
    if spaceFree(position):
        board[position] = letter
        printBoard(board)
        if checkDraw():
            print('Draw!')
        elif checkWin():
            if letter == 'X':
                print('Bot wins!')
            else:
                print('You win!')
            return
    else:
```

```
        print('Position taken, please pick a different position.')
        position = int(input('Enter new position: '))
        insertLetter(letter, position)


player = 'O'
bot = 'X'


def playerMove():
    position = int(input('Enter position for O: '))
    insertLetter(player, position)


def compMove():
    bestScore = -1000
    bestMove = 0
    for key in board.keys():
        if board[key] == ' ':
            board[key] = bot
            score = minimax(board, False)
            board[key] = ' '
            if score > bestScore:
                bestScore = score
                bestMove = key
    insertLetter(bot, bestMove)


def minimax(board, isMaximizing):
    if checkMoveForWin(bot):
        return 1
    elif checkMoveForWin(player):
        return -1
```

```
    elif checkDraw():
        return 0

    if isMaximizing:
        bestScore = -1000
        for key in board.keys():
            if board[key] == ' ':
                board[key] = bot
                score = minimax(board, False)
                board[key] = ' '
                bestScore = max(score, bestScore)
        return bestScore
    else:
        bestScore = 1000
        for key in board.keys():
            if board[key] == ' ':
                board[key] = player
                score = minimax(board, True)
                board[key] = ' '
                bestScore = min(score, bestScore)
        return bestScore

while not checkWin() and not checkDraw():
    compMove()
    if checkWin() or checkDraw():
        break
    playerMove()
    name = "Varsha Prasanth"
usn = "1BM22CS321"
```

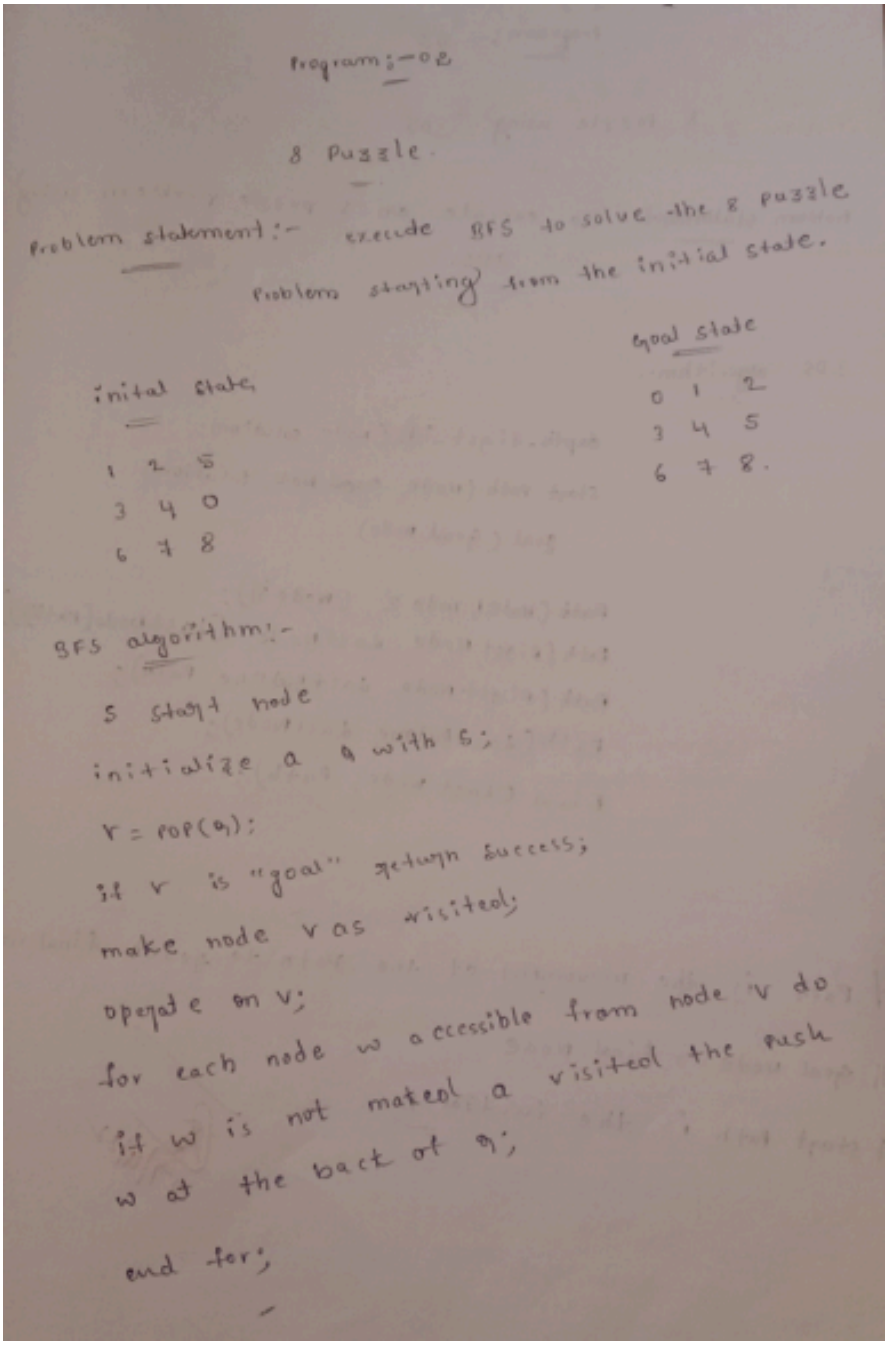
```
print(f"Name: {name}, USN: {usn}")
```

Output Snapshot

```
x| |
-+-+-
 | |
-+-+-
 | |


Enter position for O: 5
x| |
-+-+-
 |O|
-+-+-
 | |


x|x|
-+-+-
 |O|
-+-+-
 | |


Enter position for O: 3
x|x|O
-+-+-
 |O|
-+-+-
 | |


x|x|O
-+-+-
x|O|
-+-+-
 | |


Enter position for O: 7
x|x|O
-+-+-
x|O|
-+-+-
O| |


You win!
Name: Varsha Prasanth, USN: 1BM22CS321
```

Program 2 - 8 Puzzle Using BFS

Algorithm



Program :- 02

8 Puzzle

Problem statement :- execute BFS to solve the 8 puzzle problem starting from the initial state.

inital state

1	2	5
3	4	0
6	7	8

goal state

0	1	2
3	4	5
6	7	8.

BFS algorithm :-

```
s start node
initiwize a q with s;
v = pop(q);
if v is "goal" return success;
make node v as visited;
operate on v;
for each node w accessible from node v do
if w is not mateol a visiteol the push
w at the back of q;
end for;
```

Code

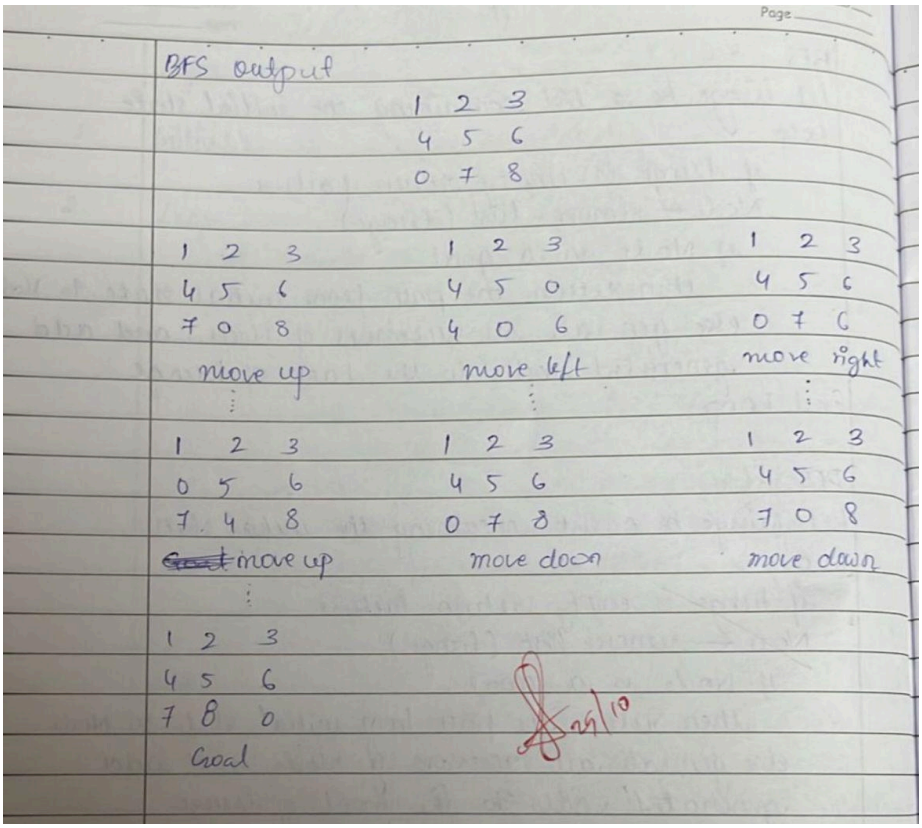
```
from collections import deque

# Define the goal state
GOAL_STATE = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]

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

def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # right, down, left,
up

    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [row[:] for row in state]  # create a copy
            new_state[x][y], new_state[new_x][new_y] =
new_state[new_x][new_y], new_state[x][y]
            neighbors.append(new_state)

    return neighbors

def bfs(start_state):
    start_tuple = tuple(tuple(row) for row in start_state)
    goal_tuple = tuple(tuple(row) for row in GOAL_STATE)

    queue = deque([start_state])  # Use the list representation for the
queue
    visited = {start_tuple: None}

    while queue:
        current_state = queue.popleft()
```

```
        if tuple(tuple(row) for row in current_state) == goal_tuple:
            break

        for neighbor in get_neighbors(current_state):
            neighbor_tuple = tuple(tuple(row) for row in neighbor)
            if neighbor_tuple not in visited:
                visited[neighbor_tuple] = current_state
                queue.append(neighbor)

    # Backtrack to find the solution path
    path = []
    while current_state is not None:
        path.append(current_state)
        current_state = visited[tuple(tuple(row) for row in current_state)]

    return path[::-1]  # Return reversed path

def print_solution(path):
    for state in path:
        for row in state:
            print(row)
        print()

# Example usage
start_state = [[2, 8, 3], [1, 6, 4], [7 ,0 ,5]]
solution_path = bfs(start_state)
print_solution(solution_path)
name = "Varsha Prasanth"
usn = "1BM22CS321"
print(f"Name: {name}, USN: {usn}")
```

Output Snapshot

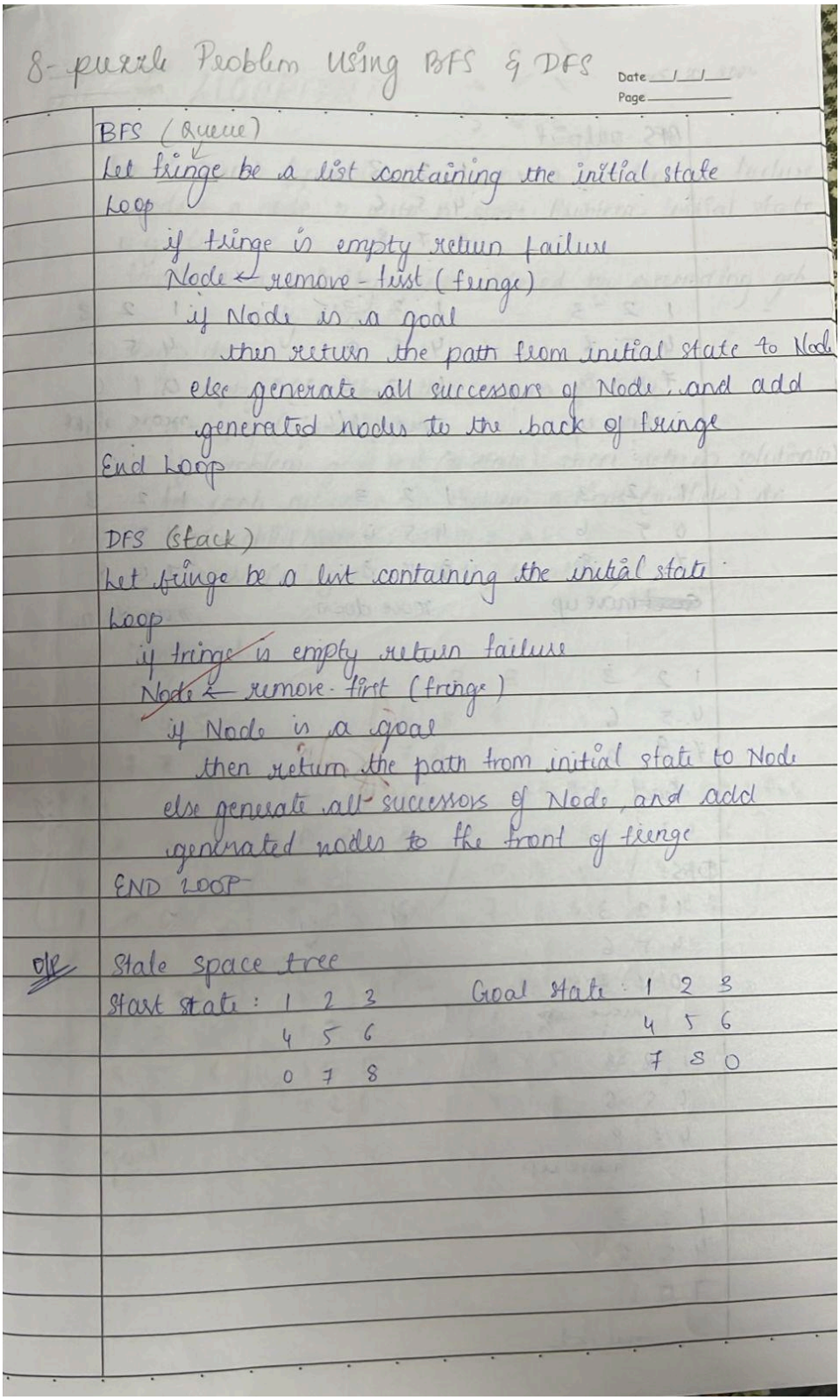
```
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

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

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

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

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

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

Name: Varsha Prasanth, USN: 1BM22CS321
```

State Space Tree



Program 3 - 8 puzzle using DFS

Algorithm



8-puzzle Problem using BFS & DFS

Date / /
Page

BFS (Queue)
Let fringe be a list containing the initial state
Loop
if fringe is empty return failure
Node ← remove-first (fringe)
if Node is a goal
then return the path from initial state to Node
else generate all successors of Node, and add generated nodes to the back of fringe
End Loop

DFS (Stack)
Let fringe be a list containing the initial state
Loop
if fringe is empty return failure
Node ← remove-first (fringe)
if Node is a goal
then return the path from initial state to Node
else generate all successors of Node, and add generated nodes to the front of fringe
END LOOP

O/P
State space tree
Start state : 1 2 3 4 5 6 0 7 8
Goal State : 1 2 3 4 5 6 7 8 0

Code

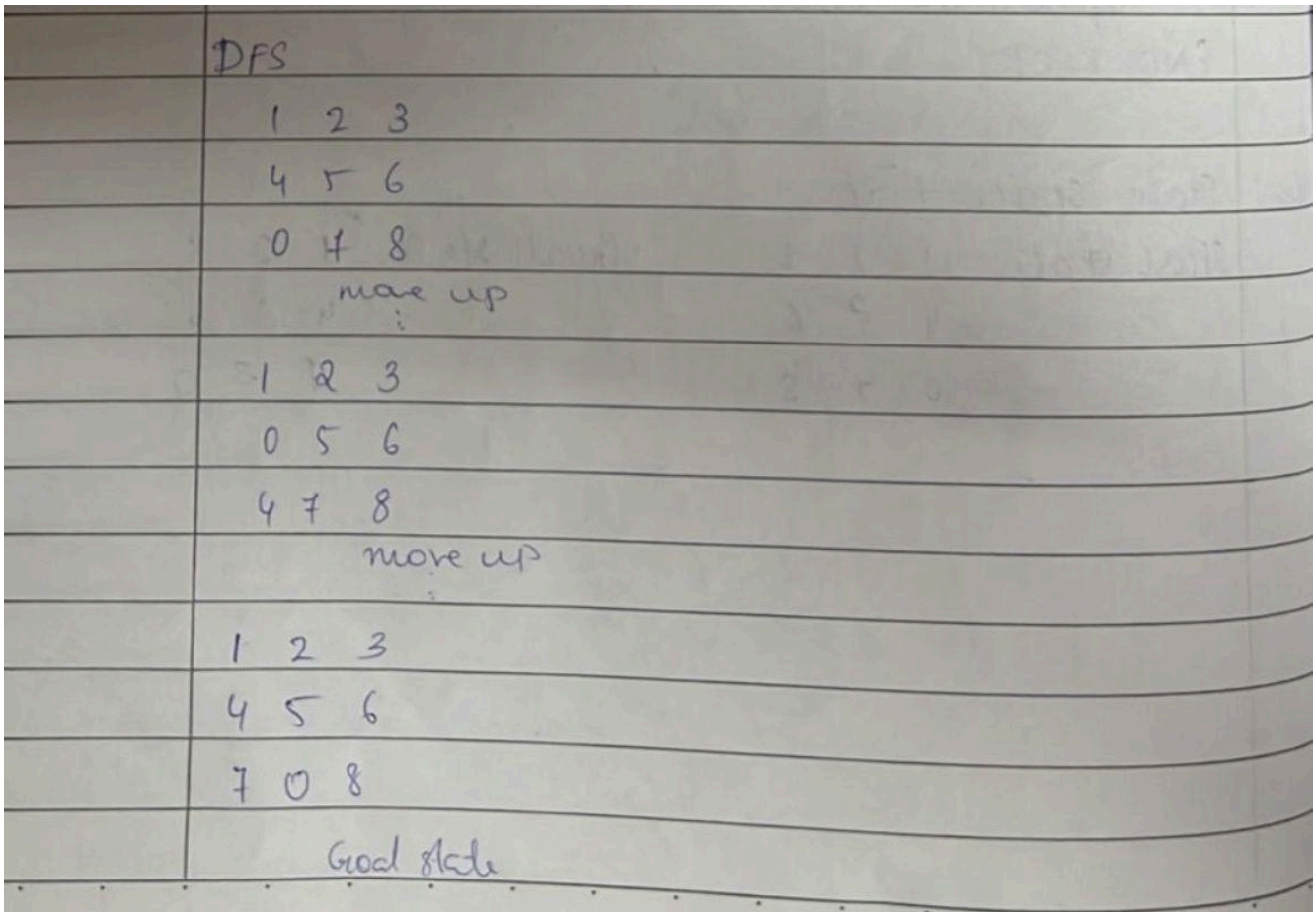
```
def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # right, down, left, up

    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [row[:] for row in state]  # Create a copy
            new_state[x][y], new_state[new_x][new_y] =
new_state[new_x][new_y], new_state[x][y]
            neighbors.append(new_state)

    return neighbors

def iterative_dfs(start_state):
    stack = [start_state]
    visited = set()
    path_map = {tuple(map(tuple, start_state)): None}

    while stack:
        current_state = stack.pop()

        if current_state == GOAL_STATE:
            return reconstruct_path(path_map, current_state)

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

```
        for neighbor in get_neighbors(current_state):
            neighbor_tuple = tuple(map(tuple, neighbor))
            if neighbor_tuple not in visited and neighbor_tuple not in
path_map:
                path_map[neighbor_tuple] = current_state
                stack.append(neighbor)

    return None

def reconstruct_path(path_map, goal_state):
    path = []
    current = goal_state
    while current is not None:
        path.append(current)
        current = path_map[tuple(map(tuple, current))]
    return path[::-1]  # Reverse to get the correct order

def print_solution(path):
    if path:
        for state in path:
            for row in state:
                print(row)
            print()
    else:
        print("No solution found.")

# Example usage
GOAL_STATE = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
start_state = [[1, 2, 3], [4, 0, 6], [7, 5, 8]]

solution_path = iterative_dfs(start_state)
print_solution(solution_path)
name = "Varsha Prasanth"
usn = "1BM22CS321"
print(f"Name: {name}, USN: {usn}")
```

Output Snapshot

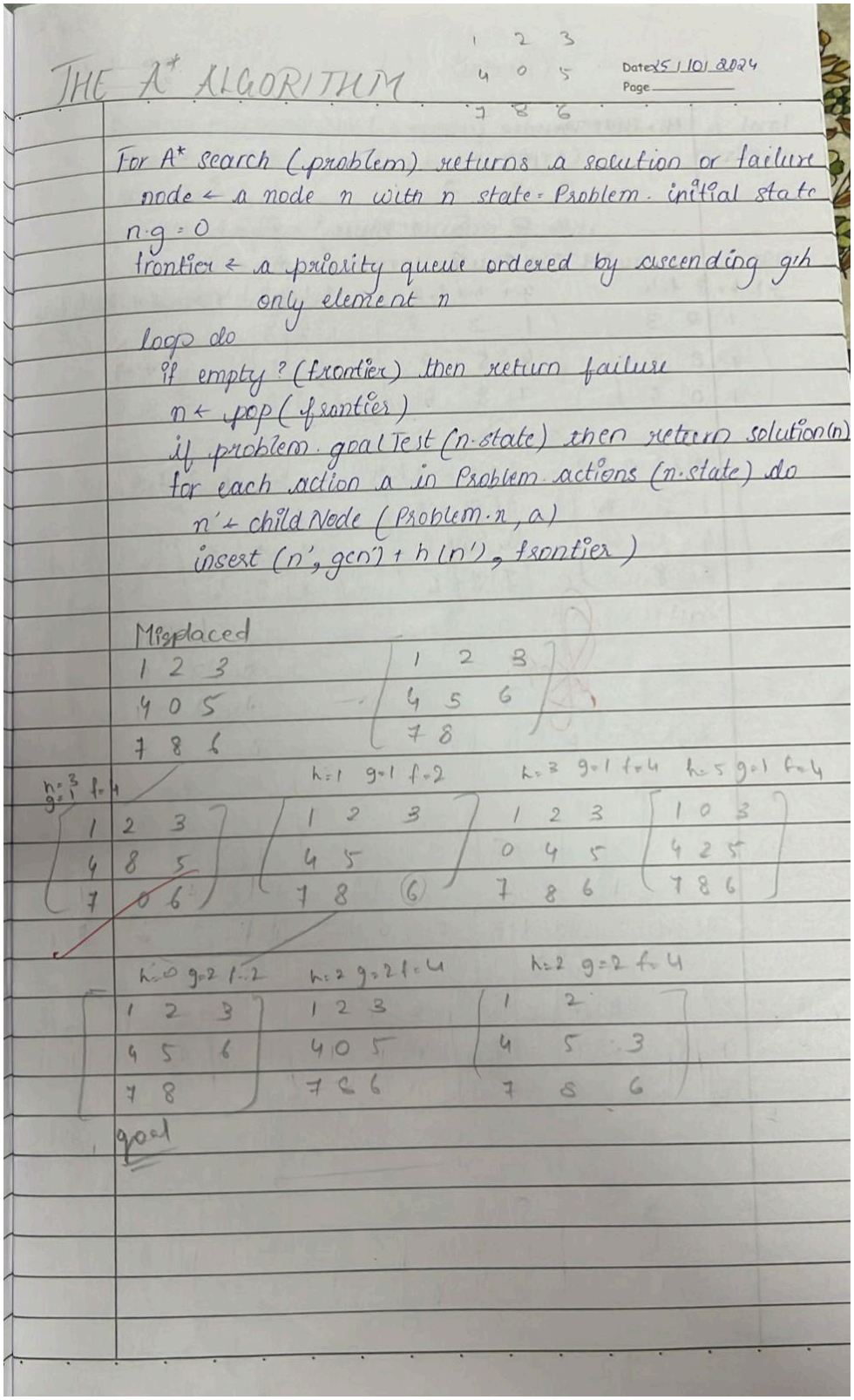
```
[1, 2, 3]
[8, 7, 5]
[0, 4, 6]

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

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

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

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

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

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

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

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

Name: Varsha Prasanth, USN: 1BM22CS321
```

StateSpaceTree



Program 04 - 8 Puzzle Using A*

Algorithm



Code

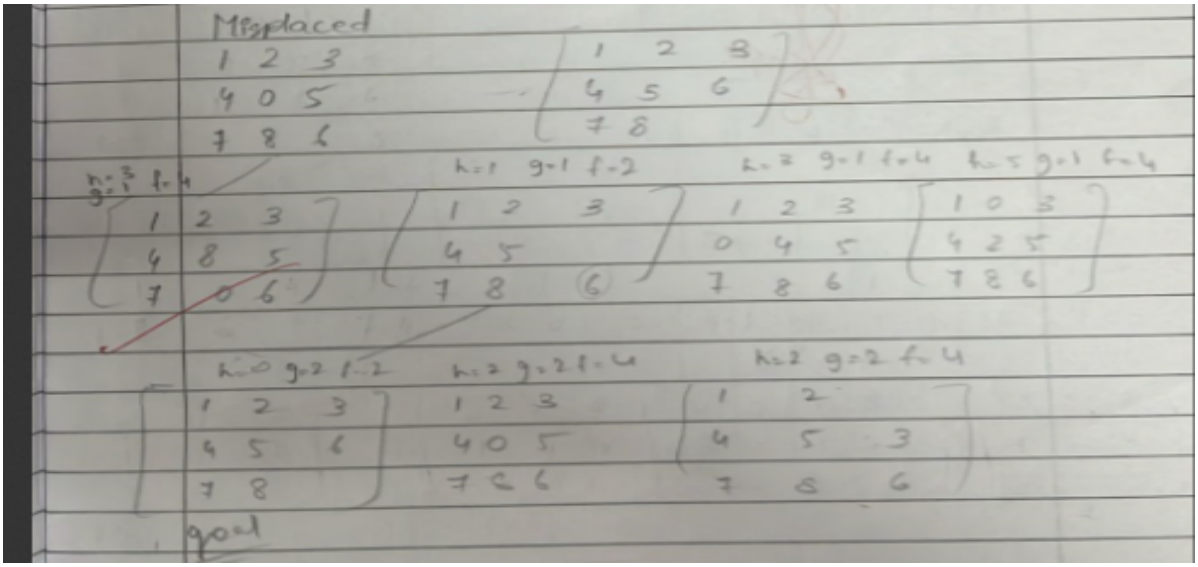
```
import heapq

class PuzzleState:
    def __init__(self, board, depth=0, path=''):
        self.board = board
        self.blank_index = board.index(0)  # Index of the blank tile
        self.depth = depth
        self.path = path  # Path taken to reach this state

    def is_goal(self):
        return self.board == [1, 2, 3, 4, 5, 6, 7, 8, 0]

    def get_possible_moves(self):
        moves = []
        row, col = divmod(self.blank_index, 3)

        if row > 0:  # Up
            moves.append(-3)
        if row < 2:  # Down
            moves.append(3)
        if col > 0:  # Left
            moves.append(-1)
        if col < 2:  # Right
            moves.append(1)

        return moves

    def generate_new_state(self, move):
        new_index = self.blank_index + move
        new_board = self.board[:]
        new_board[self.blank_index], new_board[new_index] =
new_board[new_index], new_board[self.blank_index]
        return PuzzleState(new_board, self.depth + 1, self.path +
str(new_board))
```

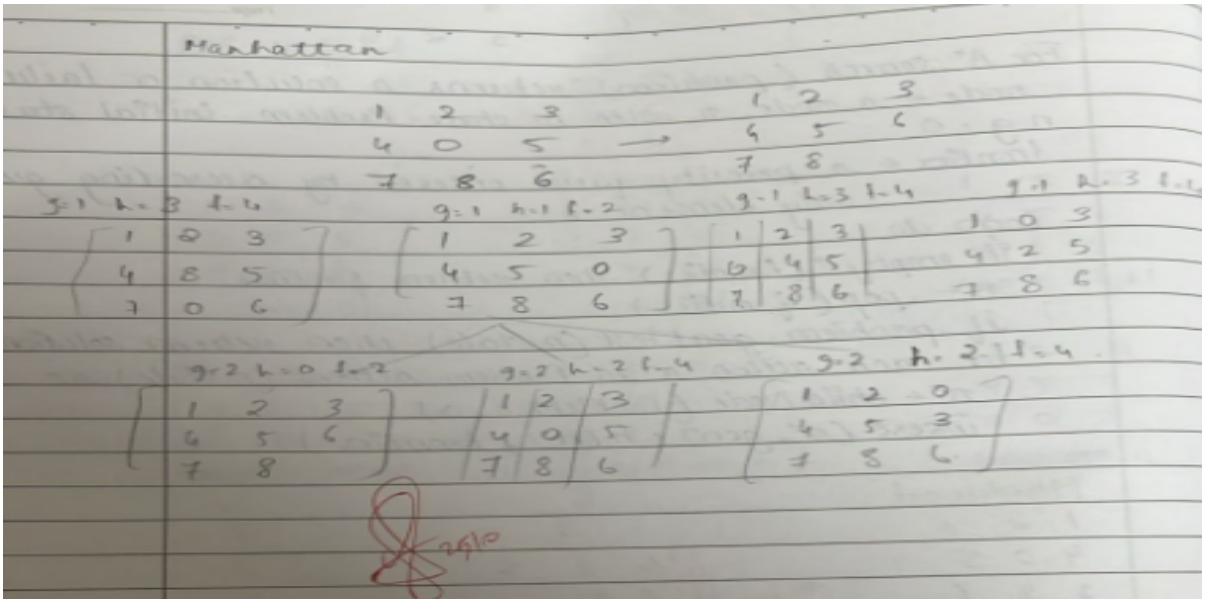
```
    def heuristic_misplaced(self):
        return sum(1 for i in range(9) if self.board[i] != 0 and self.board[i]
!= i + 1)

    def heuristic_manhattan(self):
        distance = 0
        for i in range(9):
            if self.board[i] != 0:
                target_row, target_col = divmod(self.board[i] - 1, 3)
                current_row, current_col = divmod(i, 3)
                distance += abs(target_row - current_row) + abs(target_col -
current_col)
        return distance

    def __lt__(self, other):
        return False  # Needed for priority queue to compare states


def a_star(initial_board, heuristic_type='misplaced'):
    start_state = PuzzleState(initial_board)
    if start_state.is_goal():
        return start_state.path

    # Priority queue for open states
    open_set = []
    heapq.heappush(open_set, (0, start_state))

    # Set to track visited states
    visited = set()

    while open_set:
        current_cost, current_state = heapq.heappop(open_set)

        if current_state.is_goal():
            return current_state.path

        visited.add(tuple(current_state.board))
```

```
        for move in current_state.get_possible_moves():
            new_state = current_state.generate_new_state(move)
            if tuple(new_state.board) in visited:
                continue

            # Calculate h(n) based on selected heuristic
            if heuristic_type == 'misplaced':
                h = new_state.heuristic_misplaced()
            elif heuristic_type == 'manhattan':
                h = new_state.heuristic_manhattan()
            else:
                raise ValueError("Invalid heuristic type")

            # Calculate f(n)
            f = new_state.depth + h

            # Print f(n) for the current state
            print(f"Current State: {new_state.board}, g(n): {new_state.depth},
h(n): {h}, f(n): {f}")

            heapq.heappush(open_set, (f, new_state))

    return None  # No solution found


# Example usage
initial_board = [1, 2, 3, 4, 5, 6, 0, 7, 8]  # 0 represents the blank tile
print("Solution Path (Misplaced Tiles):", a_star(initial_board,
heuristic_type='misplaced'))
print("Solution Path (Manhattan Distance):", a_star(initial_board,
heuristic_type='manhattan'))

name = "Varsha Prasanth"
usn = "1BM22CS321"
print(f"Name: {name}, USN: {usn}")
```

Output Snapshot

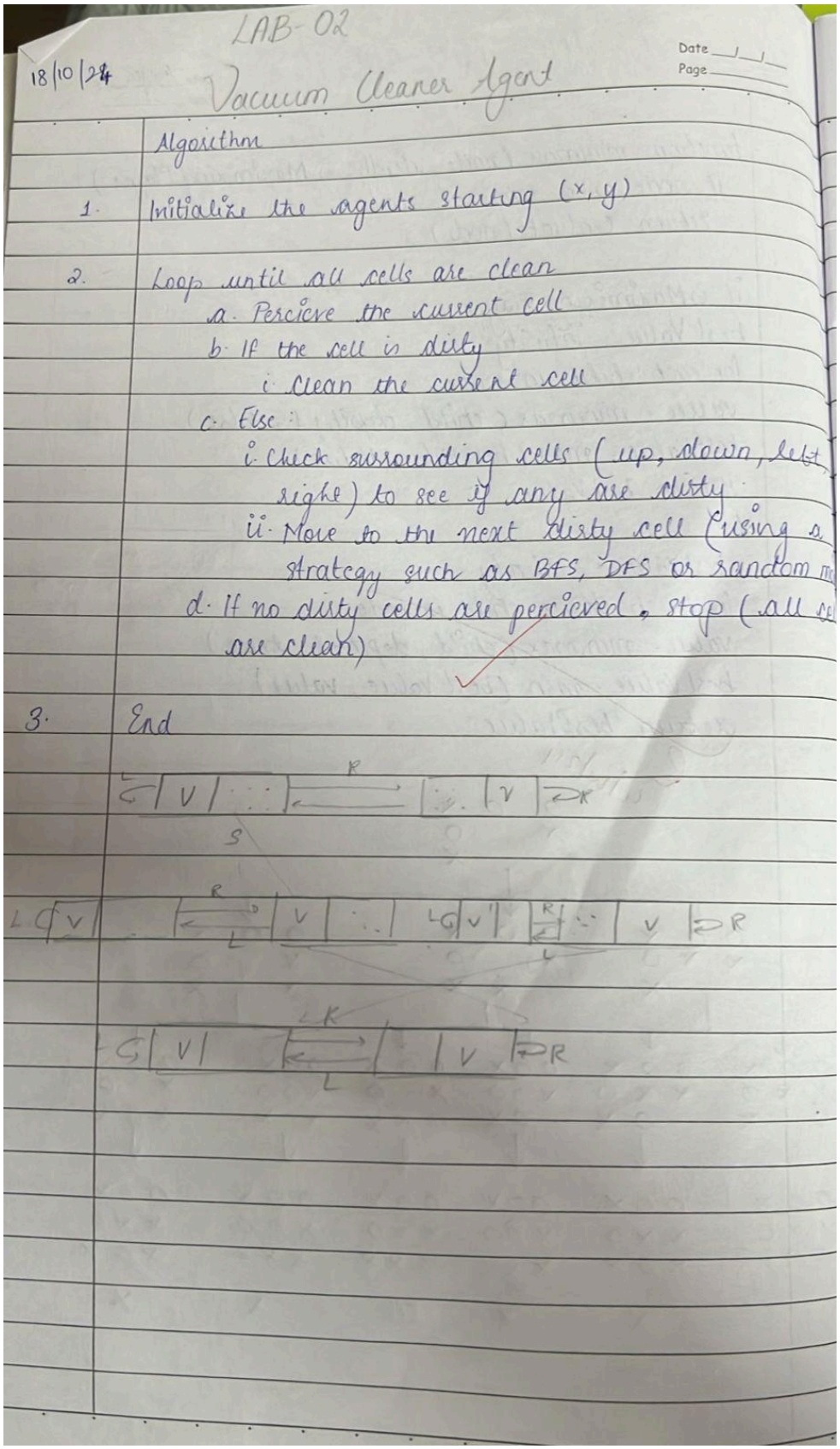
```
Current State: [1, 2, 3, 0, 5, 6, 4, 7, 8], g(n): 1, h(n): 3, f(n): 4
Current State: [1, 2, 3, 4, 5, 6, 7, 0, 8], g(n): 1, h(n): 1, f(n): 2
Current State: [1, 2, 3, 4, 0, 6, 7, 5, 8], g(n): 2, h(n): 2, f(n): 4
Current State: [1, 2, 3, 4, 5, 6, 7, 8, 0], g(n): 2, h(n): 0, f(n): 2
Solution Path (Misplaced Tiles): [1, 2, 3, 4, 5, 6, 7, 0, 8][1, 2, 3, 4, 5, 6, 7, 8, 0]
Current State: [1, 2, 3, 0, 5, 6, 4, 7, 8], g(n): 1, h(n): 3, f(n): 4
Current State: [1, 2, 3, 4, 5, 6, 7, 0, 8], g(n): 1, h(n): 1, f(n): 2
Current State: [1, 2, 3, 4, 0, 6, 7, 5, 8], g(n): 2, h(n): 2, f(n): 4
Current State: [1, 2, 3, 4, 5, 6, 7, 8, 0], g(n): 2, h(n): 0, f(n): 2
Solution Path (Manhattan Distance): [1, 2, 3, 4, 5, 6, 7, 0, 8][1, 2, 3, 4, 5, 6, 7, 8, 0]
Name: Varsha Prasanth, USN: 1BM22CS321
```





Program 5 - Vacuum Cleaner

Algorithm



LAB-02

18/10/24

Vacuum Cleaner Agent

Date __/__/__
Page

Algorithm

1. Initialize the agents starting (x, y)

2. Loop until all cells are clean
   a. Percieve the current cell
   b. If the cell is dirty
      i. Clean the current cell
   c. Else:
      i. Check surrounding cells (up, down, left, right) to see if any are dirty.
      ii. Move to the next dirty cell (using a strategy such as BFS, DFS or random m
   d. If no dirty cells are percieved, stop (all s are clean)

3. End

Code

```
def vacuum_world():
    # Initializing goal_state
    # 0 indicates Clean and 1 indicates Dirty
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter Location of Vacuum: ")  # User input for
location vacuum is placed
    status_input = input("Enter status of " + location_input + " (0 for Clean,
1 for Dirty): ")  # User input if location is dirty or clean
    status_input_complement = input("Enter status of other room (0 for Clean,
1 for Dirty): ")

    print("Initial Location Condition: " + str(goal_state))

    if location_input == 'A':
        # Location A is Dirty.
        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")
            # Suck the dirt and mark it as clean
            goal_state['A'] = '0'
            cost += 1  # Cost for suck
            print("Cost for CLEANING A: " + str(cost))
            print("Location A has been Cleaned.")

            if status_input_complement == '1':
                # If B is Dirty
                print("Location B is Dirty.")
                print("Moving right to Location B.")
                cost += 1  # Cost for moving right
                print("COST for moving RIGHT: " + str(cost))
                # Suck the dirt and mark it as clean
                goal_state['B'] = '0'
                cost += 1  # Cost for suck
                print("COST for SUCK: " + str(cost))
                print("Location B has been Cleaned.")
```

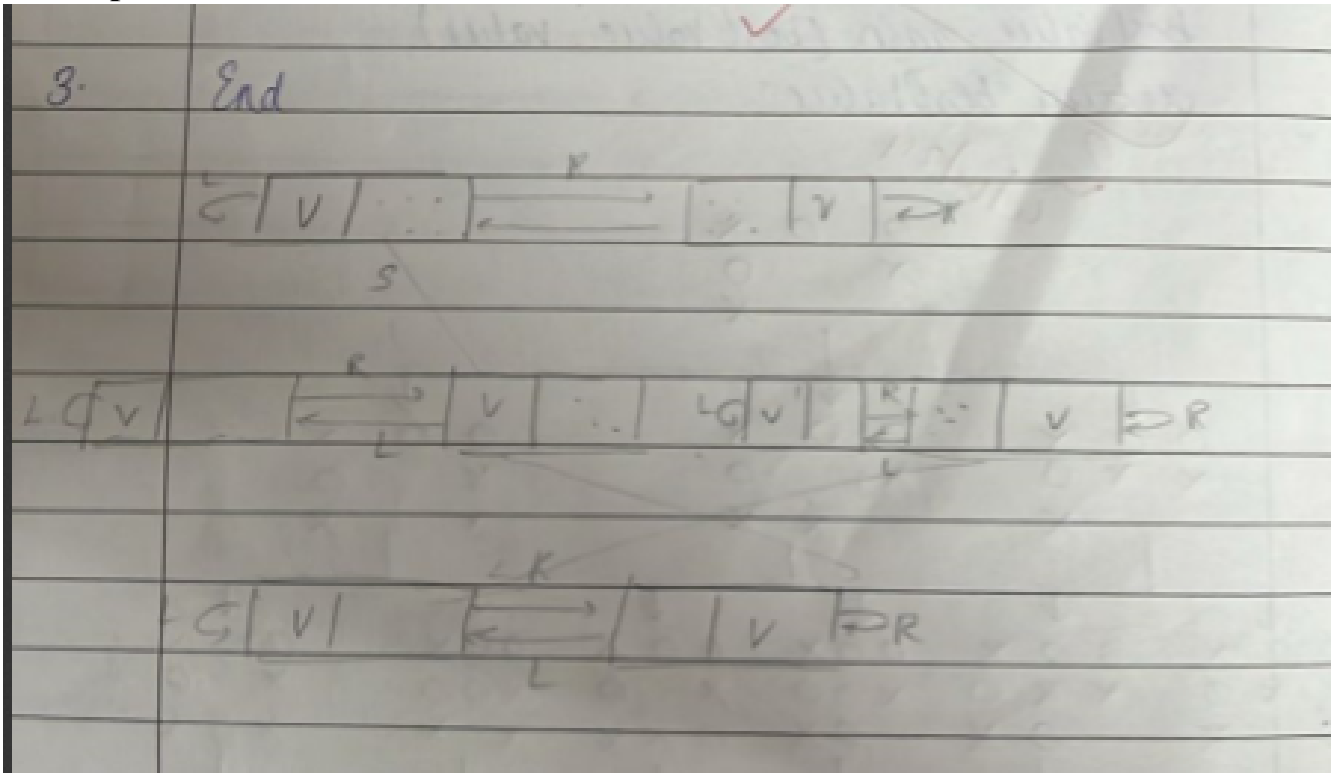
```
            else:
                print("No action needed; Location B is already clean.")
        else:
            print("Location A is already clean.")
            if status_input_complement == '1':  # If B is Dirty
                print("Location B is Dirty.")
                print("Moving RIGHT to Location B.")
                cost += 1  # Cost for moving right
                print("COST for moving RIGHT: " + str(cost))
                # Suck the dirt and mark it as clean
                goal_state['B'] = '0'
                cost += 1  # Cost for suck
                print("COST for SUCK: " + str(cost))
                print("Location B has been Cleaned.")
            else:
                print("No action needed; Location B is already clean.")

    else:  # Vacuum is placed in location B
        print("Vacuum is placed in Location B")
        if status_input == '1':
            print("Location B is Dirty.")
            # Suck the dirt and mark it as clean
            goal_state['B'] = '0'
            cost += 1  # Cost for suck
            print("COST for CLEANING B: " + str(cost))
            print("Location B has been Cleaned.")

            if status_input_complement == '1':  # If A is Dirty
                print("Location A is Dirty.")
                print("Moving LEFT to Location A.")
                cost += 1  # Cost for moving left
                print("COST for moving LEFT: " + str(cost))
                # Suck the dirt and mark it as clean
                goal_state['A'] = '0'
                cost += 1  # Cost for suck
                print("COST for SUCK: " + str(cost))
                print("Location A has been Cleaned.")
            else:
```

```
                print("No action needed; Location A is already clean.")
        else:
            print("Location B is already clean.")
            if status_input_complement == '1':  # If A is Dirty
                print("Location A is Dirty.")
                print("Moving LEFT to Location A.")
                cost += 1  # Cost for moving left
                print("COST for moving LEFT: " + str(cost))
                # Suck the dirt and mark it as clean
                goal_state['A'] = '0'
                cost += 1  # Cost for suck
                print("COST for SUCK: " + str(cost))
                print("Location A has been Cleaned.")
            else:
                print("No action needed; Location A is already clean.")

    # Done cleaning
    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))

# Call the function to run the vacuum world simulation
vacuum_world()
name = "Varsha Prasanth"
usn = "1BM22CS321"
print(f"Name: {name}, USN: {usn}")
```


Output Snapshot

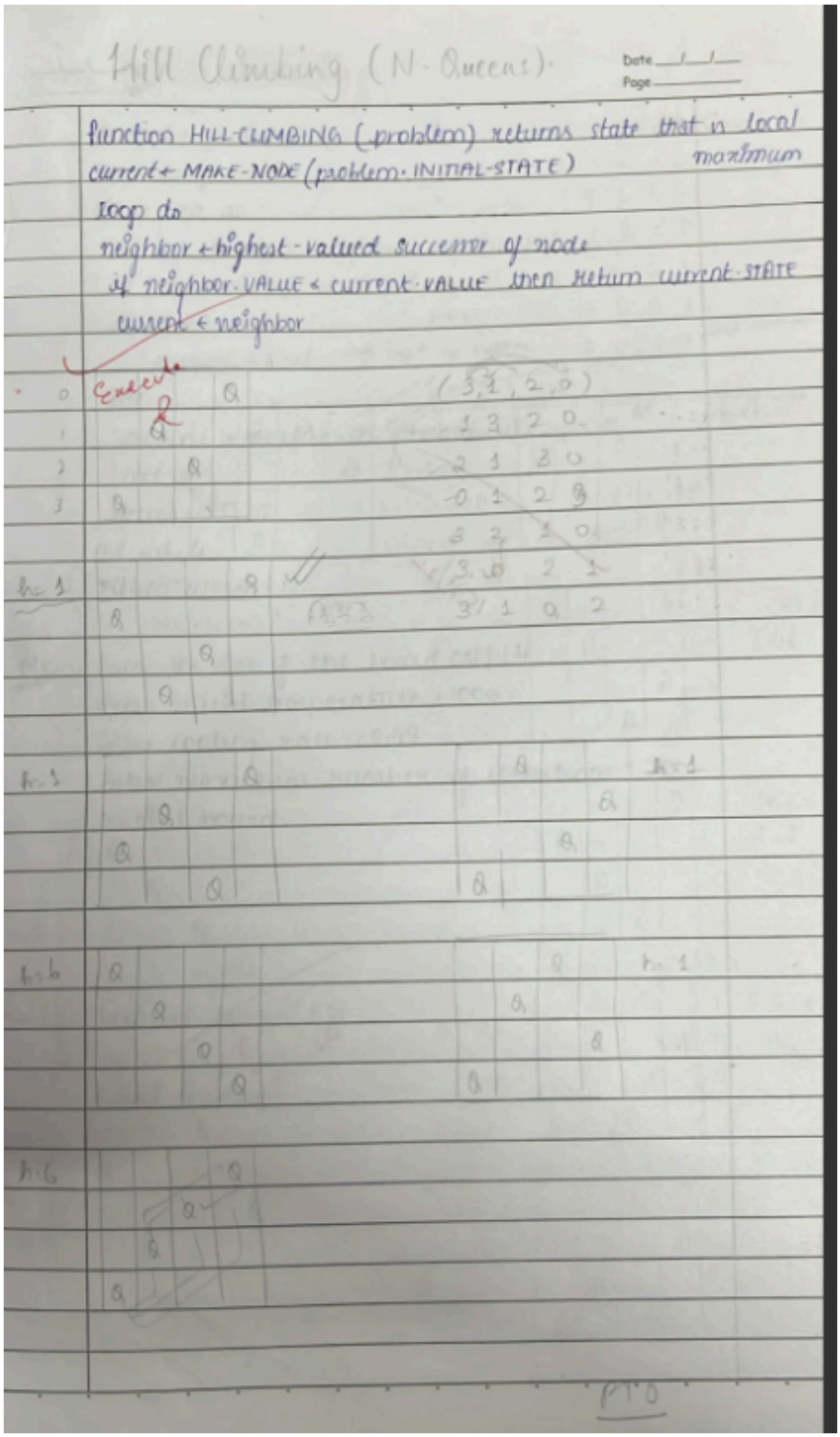
```
Enter Location of Vacuum: A
Enter status of A (0 for Clean, 1 for Dirty): 1
Enter status of other room (0 for Clean, 1 for Dirty): 0
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A: 1
Location A has been Cleaned.
No action needed; Location B is already clean.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 1
Name: Varsha Prasanth, USN: 1BM22CS321
```

State Space Tree



Program-06 Hill Climbing

Algorithm



Code

```
import random

def print_board(board, n):
    """Prints the current state of the board."""
    for row in range(n):
        line = ""
        for col in range(n):
            if board[col] == row:
                line += " Q "
            else:
                line += " . "
        print(line)
    print()

def calculate_conflicts(board, n):
    """Calculates the number of conflicts (attacks) between queens."""
    conflicts = 0
    for i in range(n):
        for j in range(i + 1, n):
            # Check if queens are in the same row or diagonal
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i -
j):
                conflicts += 1
    return conflicts

def get_best_neighbor(board, n):
    """
    Finds the best neighboring board with the fewest conflicts.
    Returns the best board and its conflict count.
    """
    current_conflicts = calculate_conflicts(board, n)
    best_board = board[:]
    best_conflicts = current_conflicts
    neighbors = []

    for col in range(n):
        original_row = board[col]
```

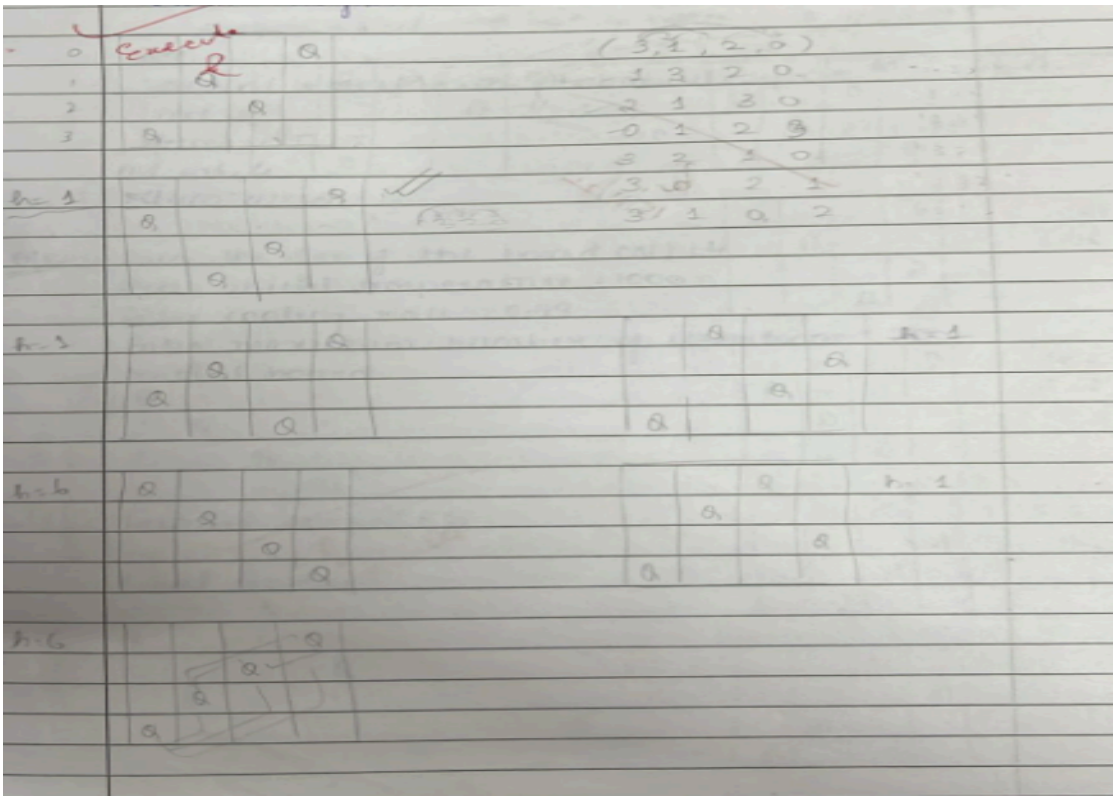
```
        for row in range(n):
            if row == original_row:
                continue
            # Move queen to a new row and calculate conflicts
            board[col] = row
            new_conflicts = calculate_conflicts(board, n)
            neighbors.append((board[:], new_conflicts))
        # Restore the original row before moving to the next column
        board[col] = original_row

    # Sort neighbors by the number of conflicts (ascending)
    neighbors.sort(key=lambda x: x[1])
    if neighbors:
        best_neighbor = neighbors[0]
        if best_neighbor[1] < best_conflicts:
            return best_neighbor
    return board, current_conflicts

def hill_climbing_with_restarts(n, initial_board, max_restarts=100):
    """
    Performs Hill Climbing with random restarts to solve the N-Queens
problem.
    Returns the final board configuration and its conflict count.
    """
    current_board = initial_board[:]
    current_conflicts = calculate_conflicts(current_board, n)

    print("Initial board:")
    print_board(current_board, n)
    print(f"Initial conflicts: {current_conflicts}\n")

    steps = 0
    restarts = 0

    while current_conflicts > 0 and restarts < max_restarts:
        new_board, new_conflicts = get_best_neighbor(current_board, n)

        steps += 1
```

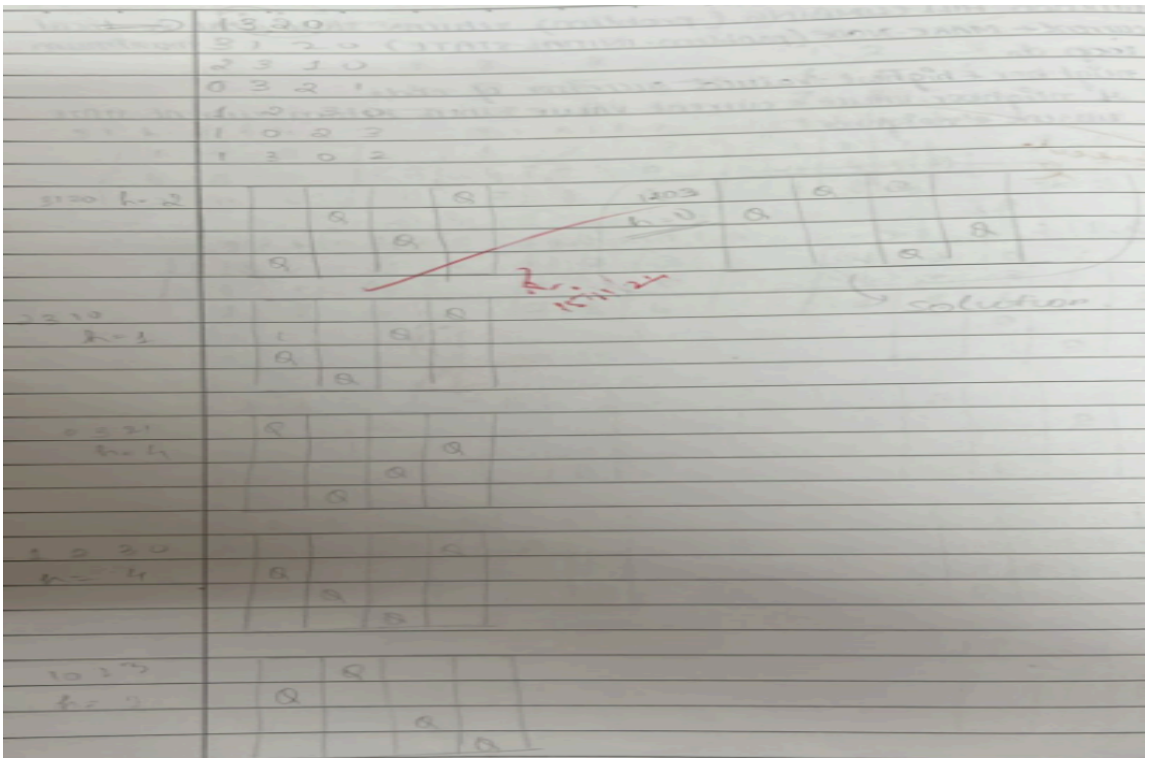
```
        print(f"Step {steps}:")
        print_board(new_board, n)
        print(f"Conflicts: {new_conflicts}\n")

        if new_conflicts < current_conflicts:
            current_board = new_board
            current_conflicts = new_conflicts
        else:
            # If no better neighbor is found, perform a random restart
            restarts += 1
            print(f"Restarting... (Restart number {restarts})\n")
            current_board = [random.randint(0, n-1) for _ in range(n)]
            current_conflicts = calculate_conflicts(current_board, n)
            print("New initial board:")
            print_board(current_board, n)
            print(f"Conflicts: {current_conflicts}\n")

    return current_board, current_conflicts

# Main function
def main():
    n = 4
    print("Enter the initial positions of queens (row numbers from 0 to 3
for each column):")
    initial_board = []
    for i in range(n):
        while True:
            try:
                row = int(input(f"Column {i}: "))
                if 0 <= row < n:
                    initial_board.append(row)
                    break
                else:
                    print(f"Please enter a number between 0 and {n-1}.")
            except ValueError:
                print("Invalid input. Please enter an integer.")

    solution, conflicts = hill_climbing_with_restarts(n, initial_board)
```

```
    print("Final solution:")
    print_board(solution, n)
    if conflicts == 0:
        print("A solution was found with no conflicts!")
    else:
        print(f"No solution was found after {100} restarts. Final number of
conflicts: {conflicts}")

if __name__ == "__main__":
    main()

name = "Varsha Prasanth"
usn = "1BM22CS321"
print(f"Name: {name}, USN: {usn}")
```

OUTPUT

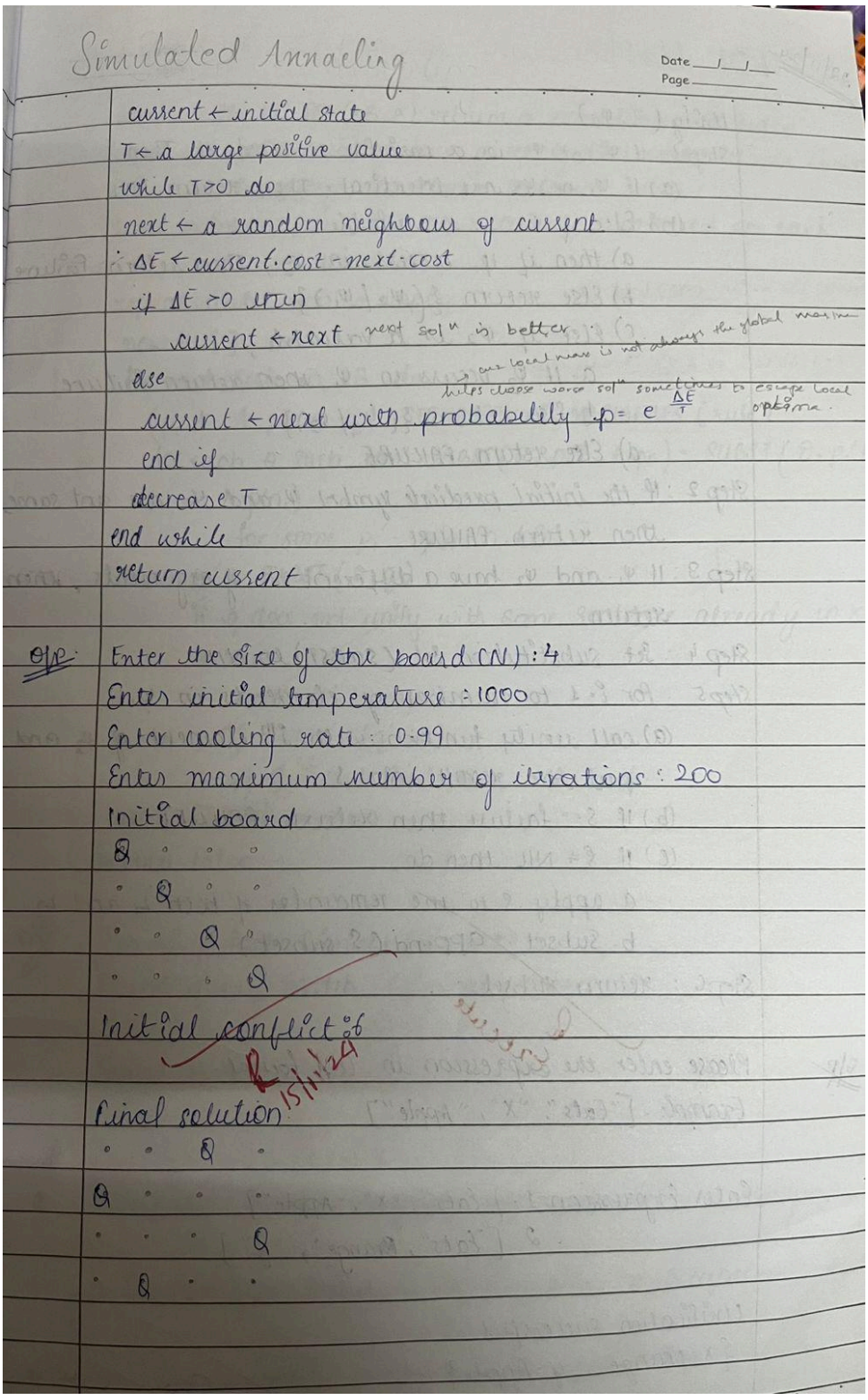
```
Enter the initial positions of queens (row numbers from 0 to 3 for each column):
Column 0: 3
Column 1: 1
Column 2: 2
Column 3: 0
Initial board:
. . . Q
. Q . .
. . Q .
Q . . .

Initial conflicts: 2

    Step 6:
     Q   Q   -   -
     -   -   -   Q
     -   -   -   -
     -   -   Q   -

    Conflicts: 1

    Step 7:
     -   Q   -   -
     -   -   -   Q
     Q   -   -   -
     -   -   Q   -

    Conflicts: 0

    Final solution:
     -   Q   -   -
     -   -   -   Q
     Q   -   -   -
     -   -   Q   -

    A solution was found with no conflicts!
    Name: Varsha Prasanth, USN: 1BM22CS321
```





Program-07 Simulated Annealing

Algorithm



Code

```
import random
import math

def print_board(board, n):
    """Prints the current state of the board."""
    for row in range(n):
        line = ""
        for col in range(n):
            if board[col] == row:
                line += " Q "  # Queen is represented by "Q"
            else:
                line += " . "  # Empty space represented by "."
        print(line)
    print()

def calculate_conflicts(board, n):
    """Calculates the number of conflicts (attacks) between queens."""
    conflicts = 0
    for i in range(n):
        for j in range(i + 1, n):
            # Check if queens are in the same row or diagonal
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i -
j):
                conflicts += 1
    return conflicts

def simulated_annealing(n, initial_temp=1000, cooling_rate=0.995,
max_iterations=10000):
    """Simulated Annealing algorithm to solve N-Queens."""
    # Initial random board configuration (one queen in each column)
    board = [random.randint(0, n - 1) for _ in range(n)]
    current_conflicts = calculate_conflicts(board, n)
    temperature = initial_temp
    iteration = 0

    print("Initial board:")
    print_board(board, n)
    print(f"Initial conflicts: {current_conflicts}\n")
```

```
    while current_conflicts > 0 and iteration < max_iterations:
        # Generate a neighboring state by moving a queen to another row in
its column
        col = random.randint(0, n - 1)
        original_row = board[col]
        new_row = random.randint(0, n - 1)
        while new_row == original_row:
            new_row = random.randint(0, n - 1)  # Ensure we are moving the
queen to a new row
        board[col] = new_row

        # Calculate the number of conflicts in the new configuration
        new_conflicts = calculate_conflicts(board, n)

        # If the new state has fewer conflicts, accept it.
        # If the new state has more conflicts, accept it with a certain
probability.
        if new_conflicts < current_conflicts or random.random() <
math.exp((current_conflicts - new_conflicts) / temperature):
            current_conflicts = new_conflicts
        else:
            # If no improvement, revert the move
            board[col] = original_row

        # Reduce the temperature according to the cooling schedule
        temperature *= cooling_rate

        iteration += 1

        # Display progress
        if iteration % 1000 == 0:
            print(f"Iteration {iteration}: Conflicts = {current_conflicts},
Temperature = {temperature}")
            print_board(board, n)

    return board, current_conflicts

def main():
    # Input dynamic parameters
    print("Welcome to the N-Queens Problem Solver using Simulated
Annealing!")
    n = int(input("Enter the size of the board (N): "))
```

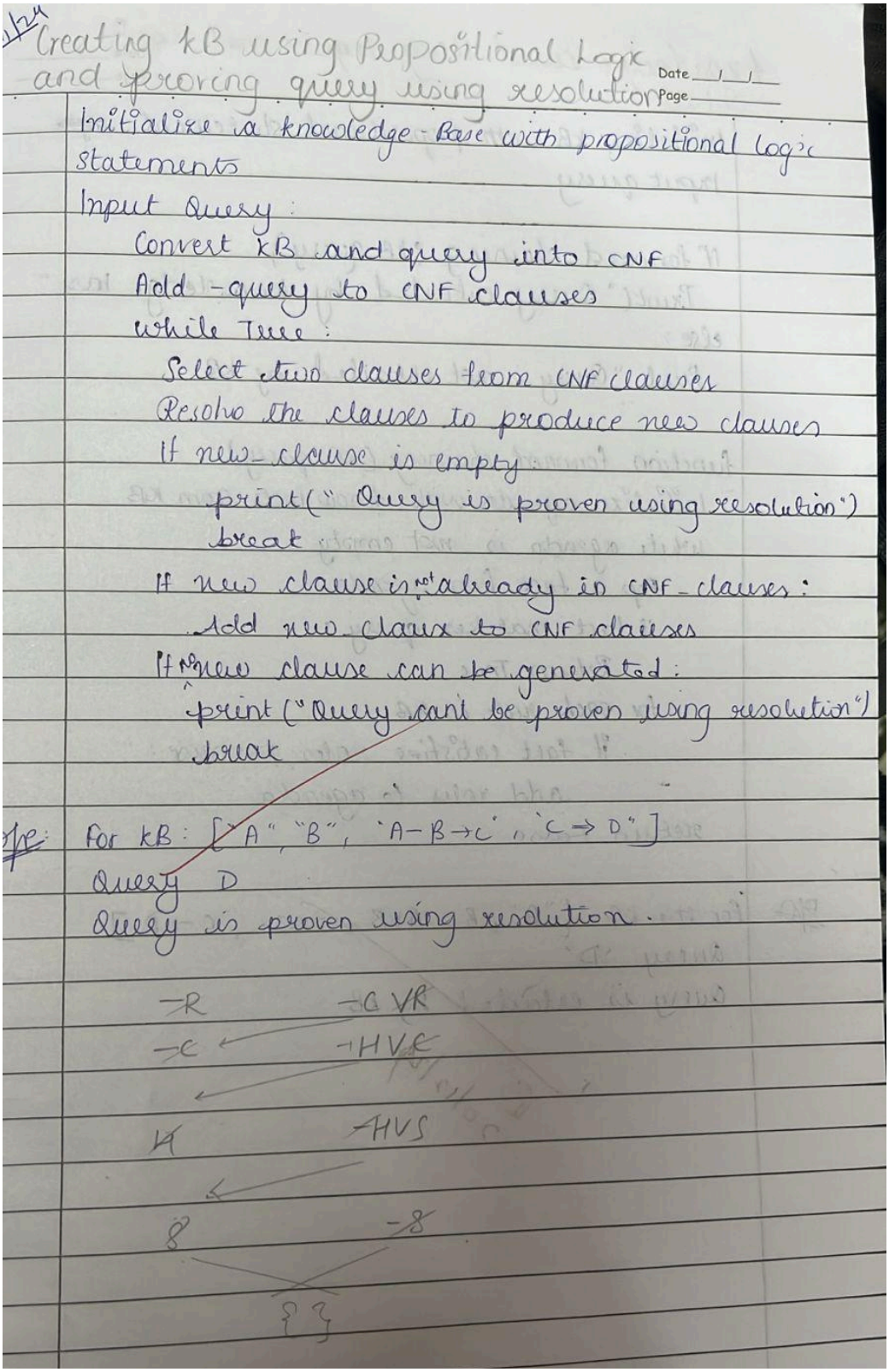
```
    initial_temp = float(input("Enter the initial temperature (e.g., 1000):
"))
    cooling_rate = float(input("Enter the cooling rate (e.g., 0.995): "))
    max_iterations = int(input("Enter the maximum number of iterations
(e.g., 10000): "))

    solution, conflicts = simulated_annealing(n, initial_temp,
cooling_rate, max_iterations)

    print("Final solution:")
    print_board(solution, n)
    if conflicts == 0:
        print("A solution was found with no conflicts!")
    else:
        print(f"No solution was found after {max_iterations} iterations.
Final number of conflicts: {conflicts}")

if __name__ == "__main__":
    main()
name = "Varsha Prasanth"
usn = "1BM22CS321"
print(f"Name: {name}, USN: {usn}")
```

OUTPUT

```
Welcome to the N-Queens Problem Solver using Simulated Annealing!
Enter the size of the board (N): 4
Enter the initial temperature (e.g., 1000): 1000
Enter the cooling rate (e.g., 0.995): 0.995
Enter the maximum number of iterations (e.g., 10000): 200
Initial board:
 .  .  .  .
 .  .  .  Q
 Q  .  Q  .
 .  Q  .  .

Initial conflicts: 5

Final solution:
 .  Q  .  .
 .  .  .  Q
 Q  .  .  .
 .  .  Q  .

A solution was found with no conflicts!
Name: Varsha Prasanth, USN: 1BM22CS321
```

Program-08 KnowledgeBase - Resolution

Algorithm



Code

```
# Function to negate a literal
def negate(literal):
    """Negate a literal."""
    if isinstance(literal, tuple) and literal[0] == "not":
        # If the literal is already negated, return the positive form
        return literal[1]
    else:
        # Otherwise, return the negated form
        return ("not", literal)

# Function to resolve two clauses
def resolve(clause1, clause2):
    """Return the resolvent of two clauses."""
    resolvents = set()
    for literal1 in clause1:
        for literal2 in clause2:
            if literal1 == negate(literal2):
                resolvent = (clause1 - {literal1}) | (clause2 - {literal2})
                print(f" Resolving literal: {literal1} with {literal2}")
                print(f" Resulting Resolvent: {resolvent}")
                resolvents.add(frozenset(resolvent))
    return resolvents

# Function to perform resolution on the KB and query with detailed output
def resolution_algorithm(KB, query):
    """Perform the resolution algorithm to check if the query can be
proven."""
    print("\n--- Step-by-Step Resolution Process ---")
    # Add the negation of the query to the knowledge base
    negated_query = negate(query)
    KB.append(frozenset([negated_query]))
    print(f"Negated Query Added to KB: {negated_query}")

    # Initialize the set of clauses to process
    clauses = set(KB)

    step = 1
    while True:
        new_clauses = set()
        print(f"\nStep {step}: Resolving Clauses")
        for c1 in clauses:
```

```
            for c2 in clauses:
                if c1 != c2:
                    print(f" Resolving clauses: {c1} and {c2}")
                    resolvent = resolve(c1, c2)

                    for res in resolvent:
                        if frozenset([]) in resolvent:
                            print("\nEmpty clause derived! The query is
provable.")
                            return True  # Empty clause found, contradiction,
query is provable

                        new_clauses.add(res)

        if new_clauses.issubset(clauses):
            print("\nNo new clauses can be derived. The query is not
provable.")
            return False  # No new clauses, query is not provable

        clauses.update(new_clauses)
        step += 1

# Knowledge Base (KB) from the image facts
KB = [
    frozenset([("not", "food(x)"), ("likes", "John", "x")]),  # 1
    frozenset([("food", "Apple")]),  # 2
    frozenset([("food", "vegetables")]),  # 3
    frozenset([("not", "eats(y, z)"), ("killed", "y"), ("food", "z")]),  # 4
    frozenset([("eats", "Anil", "Peanuts")]),  # 5
    frozenset([("alive", "Anil")]),  # 6
    frozenset([("not", "eats(Anil, w)"), ("eats", "Harry", "w")]),  # 7
    frozenset([("killed", "g"), ("alive", "g")]),  # 8
    frozenset([("not", "alive(k)"), ("not", "killed(k)")]),  # 9
    frozenset([("likes", "John", "Peanuts")])  # 10
]

# Query to prove
query = ("likes", "John", "Peanuts")

# Perform resolution to check if the query is provable
result = resolution_algorithm(KB, query)
if result:
    print("\nQuery is provable.")
```

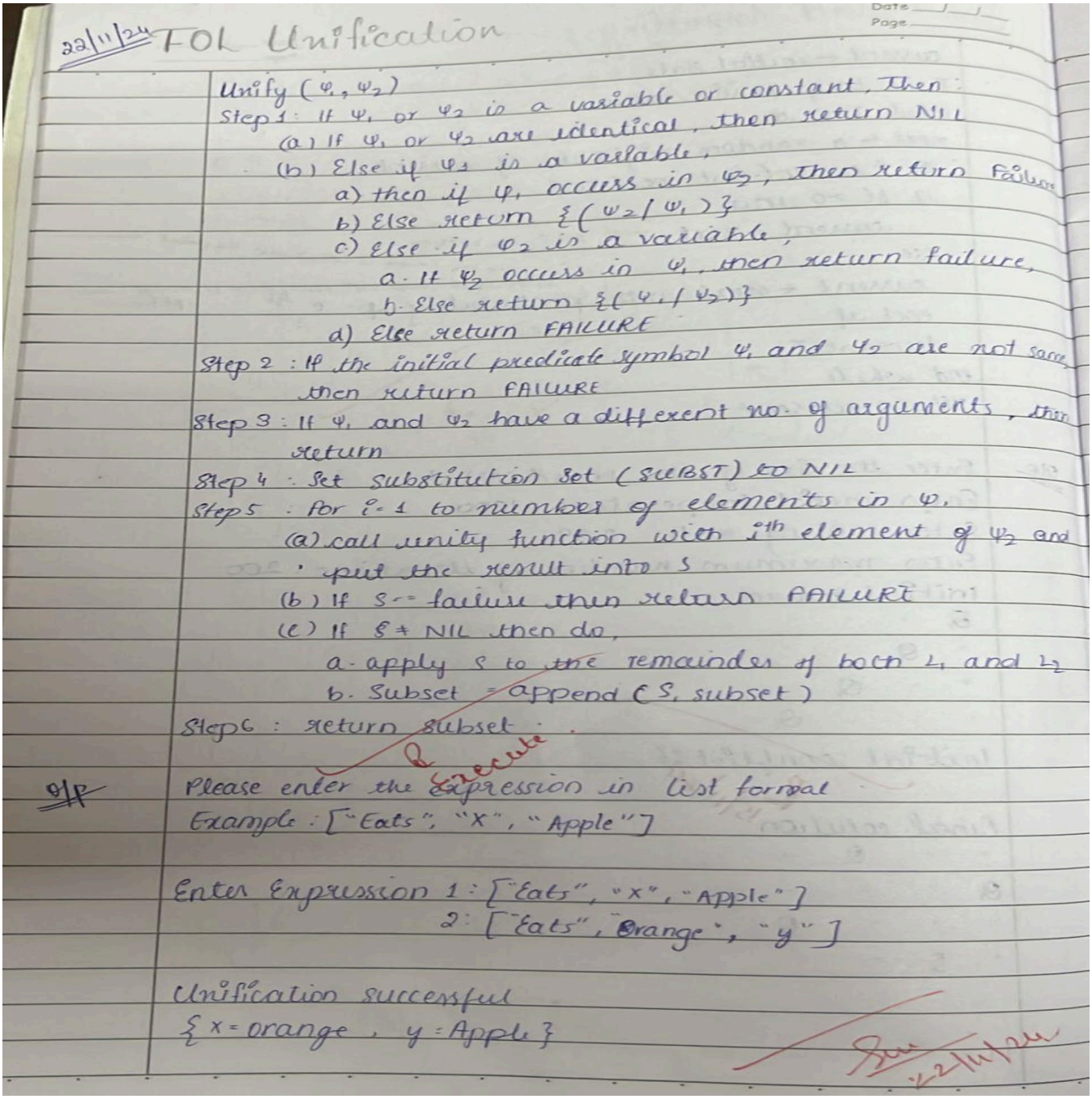
```
else:
    print("\nQuery is not provable.")
name = "Varsha Prasanth"
usn = "1BM22CS321"
print(f"Name: {name}, USN: {usn}")
```

Output Snapshot

```
Resolving clauses: frozenset({('eats', 'Anil', 'Peanuts')}) and frozenset({('not', 'food(x)'), ('likes', 'John', 'x')})
Resolving clauses: frozenset({('killed', 'g'), ('alive', 'g')}) and frozenset({('alive', 'Anil')})
Resolving clauses: frozenset({('killed', 'g'), ('alive', 'g')}) and frozenset({('eats', 'Anil', 'Peanuts')})
Resolving clauses: frozenset({('killed', 'g'), ('alive', 'g')}) and frozenset({('food', 'Apple')})
Resolving clauses: frozenset({('killed', 'g'), ('alive', 'g')}) and frozenset({('not', ('likes', 'John', 'Peanuts'))})
Resolving clauses: frozenset({('killed', 'g'), ('alive', 'g')}) and frozenset({('eats', 'Harry', 'w'), ('not', 'eats(Anil, w)')})
Resolving clauses: frozenset({('killed', 'g'), ('alive', 'g')}) and frozenset({('food', 'vegetables')})
Resolving clauses: frozenset({('killed', 'g'), ('alive', 'g')}) and frozenset({('not', 'killed(k)'), ('not', 'alive(k)')})
Resolving clauses: frozenset({('killed', 'g'), ('alive', 'g')}) and frozenset({('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y')})
Resolving clauses: frozenset({('killed', 'g'), ('alive', 'g')}) and frozenset({('likes', 'John', 'Peanuts')})
Resolving clauses: frozenset({('killed', 'g'), ('alive', 'g')}) and frozenset({('not', 'food(x)'), ('likes', 'John', 'x')})
Resolving clauses: frozenset({('food', 'Apple')}) and frozenset({('alive', 'Anil')})
Resolving clauses: frozenset({('food', 'Apple')}) and frozenset({('eats', 'Anil', 'Peanuts')})
Resolving clauses: frozenset({('food', 'Apple')}) and frozenset({('killed', 'g'), ('alive', 'g')})
Resolving clauses: frozenset({('food', 'Apple')}) and frozenset({('not', ('likes', 'John', 'Peanuts'))})
Resolving clauses: frozenset({('food', 'Apple')}) and frozenset({('eats', 'Harry', 'w'), ('not', 'eats(Anil, w)')})
Resolving clauses: frozenset({('food', 'Apple')}) and frozenset({('food', 'vegetables')})
Resolving clauses: frozenset({('food', 'Apple')}) and frozenset({('not', 'killed(k)'), ('not', 'alive(k)')})
Resolving clauses: frozenset({('food', 'Apple')}) and frozenset({('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y')})
Resolving clauses: frozenset({('food', 'Apple')}) and frozenset({('likes', 'John', 'Peanuts')})
Resolving clauses: frozenset({('food', 'Apple')}) and frozenset({('not', 'food(x)'), ('likes', 'John', 'x')})
Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('alive', 'Anil')})
Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('eats', 'Anil', 'Peanuts')})
Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('killed', 'g'), ('alive', 'g')})
Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('food', 'Apple')})
Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('eats', 'Harry', 'w'), ('not', 'eats(Anil, w)')})
Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('food', 'vegetables')})
Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('not', 'killed(k)'), ('not', 'alive(k)')})
Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y')})
Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('likes', 'John', 'Peanuts')})
Resolving literal: ('not', ('likes', 'John', 'Peanuts')) with ('likes', 'John', 'Peanuts')
Resulting Resolvent: frozenset()

Empty clause derived! The query is provable.

Query is provable.
Name: Varsha Prasanth, USN: 1BM22CS321
```

Program-09 Unification in first order logic

Algorithm



22/11/24 FOL Unification

Unify (Ψ_1, Ψ_2)

Step 1: If Ψ_1 or Ψ_2 is a variable or constant, 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 Ψ_1 and Ψ_2 are not same, then return FAILURE

Step 3: If Ψ_1 and Ψ_2 have a different no. of arguments, then return

Step 4: Set substitution set (SUBST) to NIL

Step 5: For i = 1 to number of elements in Ψ_1.
- (a) call unify function with i^{th} element of Ψ_1 and Ψ_2 & put the result into S
- (b) If S == failure then return FAILURE
- (c) If S ≠ NIL then do,
  - a. apply S to the remainder of both L_1 and L_2
  - b. Subset = append (S, subset)

Step 6: return subset.

Execute

O/P

Please enter the Expression in list format
Example: ["Eats", "X", "Apple"]

Enter Expression 1: ["Eats", "x", "Apple"]
2: ["Eats", "Orange", "y"]

Unification successful
{x = orange, y = Apple}

22/11/24

Code

```
import ast

from typing import Union, List, Dict, Tuple

from collections import deque




# Define Term Classes

class Term:

    def substitute(self, subs: Dict[str, 'Term']) -> 'Term':

        raise NotImplementedError




    def occurs(self, var: 'Variable') -> bool:

        raise NotImplementedError
```

```
    def __eq__(self, other):

        raise NotImplementedError



    def __str__(self):

        raise NotImplementedError




class Variable(Term):

    def __init__(self, name: str):

        self.name = name



    def substitute(self, subs: Dict[str, Term]) -> Term:

        if self.name in subs:
```

```
            return subs[self.name].substitute(subs)

        return self


    def occurs(self, var: 'Variable') -> bool:

        return self.name == var.name


    def __eq__(self, other):

        return isinstance(other, Variable) and self.name == other.name


    def __str__(self):

        return self.name


class Constant(Term):
```

```
    def __init__(self, name: str):

        self.name = name




    def substitute(self, subs: Dict[str, Term]) -> Term:

        return self




    def occurs(self, var: 'Variable') -> bool:

        return False




    def __eq__(self, other):

        return isinstance(other, Constant) and self.name == other.name




    def __str__(self):
```

```
        return self.name




class Function(Term):

    def __init__(self, name: str, args: List[Term]):

        self.name = name

        self.args = args




    def substitute(self, subs: Dict[str, Term]) -> Term:

        substituted_args = [arg.substitute(subs) for arg in self.args]

        return Function(self.name, substituted_args)




    def occurs(self, var: 'Variable') -> bool:

        return any(arg.occurs(var) for arg in self.args)
```

```
    def __eq__(self, other):

        return (

            isinstance(other, Function) and

            self.name == other.name and

            len(self.args) == len(other.args) and

            all(a == b for a, b in zip(self.args, other.args))

        )




    def __str__(self):

        return f"{self.name}({', '.join(str(arg) for arg in self.args)})"




def parse_expression(expr: List) -> Term:
```

```
if not isinstance(expr, list) or not expr:

    raise ValueError("Expression must be a non-empty list")




func_name = expr[0]

args = expr[1:]




parsed_args = []

for arg in args:

    if isinstance(arg, list):

        parsed_args.append(parse_expression(arg))

    elif isinstance(arg, str):

        if arg[0].islower():

            parsed_args.append(Variable(arg))
```

```
            elif arg[0].isupper():
                parsed_args.append(Constant(arg))
            else:
                raise ValueError(f"Invalid argument format: {arg}")
        else:
            raise ValueError(f"Unsupported argument type: {arg}")

    return Function(func_name, parsed_args)


def unify_terms(term1: Term, term2: Term) -> Union[Dict[str, Term], str]:
    # Initialize substitution set S
    S: Dict[str, Term] = {}
```

```
# Initialize the equation list

equations: deque[Tuple[Term, Term]] = deque()

equations.append((term1, term2))




while equations:

    s, t = equations.popleft()

    s = s.substitute(S)

    t = t.substitute(S)



    if s == t:

        continue

    elif isinstance(s, Variable):

        if t.occurs(s):
```

```
            return "FAILURE"

        S[s.name] = t

        # Apply the substitution to existing substitutions

        for var in S:

            S[var] = S[var].substitute({s.name: t})

    elif isinstance(t, Variable):

        if s.occurs(t):

            return "FAILURE"

        S[t.name] = s

        for var in S:

            S[var] = S[var].substitute({t.name: s})

    elif isinstance(s, Function) and isinstance(t, Function):

        if s.name != t.name or len(s.args) != len(t.args):
```

```
                return "FAILURE"

            for s_arg, t_arg in zip(s.args, t.args):

                equations.append((s_arg, t_arg))

        elif isinstance(s, Constant) and isinstance(t, Constant):

            if s.name != t.name:

                return "FAILURE"

            # else, they are equal; continue

        else:

            return "FAILURE"



    return S



def format_substitution(S: Dict[str, Term]) -> str:
```

```
    if not S:

        return "{}"

    return "{ " + ", ".join(f"{var} = {term}" for var, term in S.items()) + "
}"




def unify(expression1: List, expression2: List) -> Union[Dict[str, Term],
str]:

    try:

        term1 = parse_expression(expression1)

        term2 = parse_expression(expression2)

    except ValueError as e:

        return f"FAILURE: {e}"




    result = unify_terms(term1, term2)
```

```
    return result



def main():

    print("=== Unification Algorithm ===\n")

    print("Please enter the expressions in list format.")

    print("Example: [\"Eats\", \"x\", \"Apple\"]\n")



    try:

        expr1_input = input("Enter Expression 1: ")

        expression1 = ast.literal_eval(expr1_input)

        if not isinstance(expression1, list):

            raise ValueError("Expression must be a list.")
```

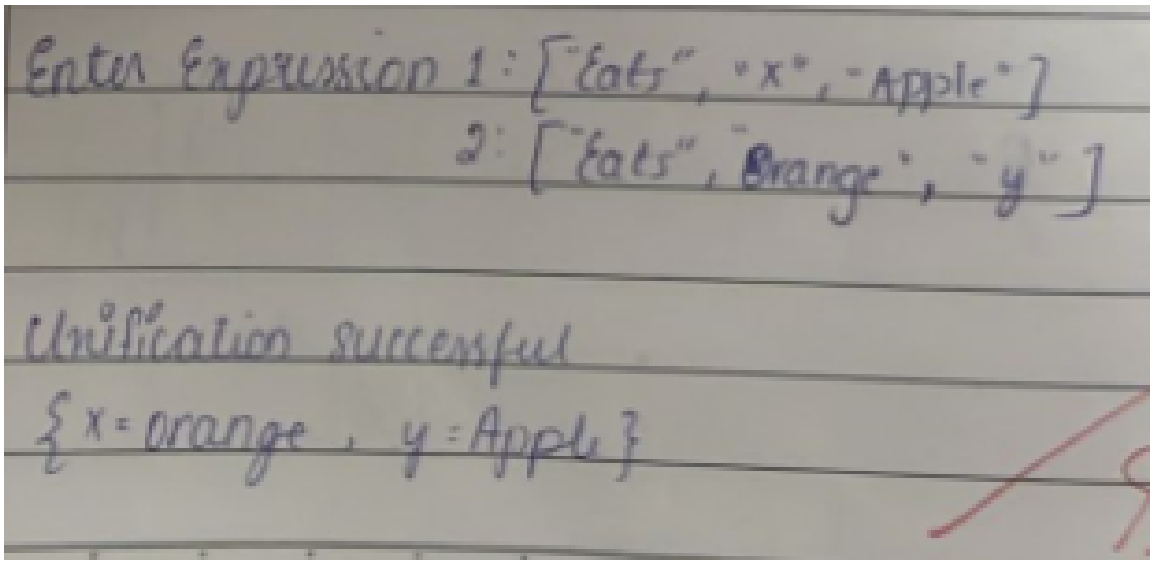
```
        expr2_input = input("Enter Expression 2: ")

        expression2 = ast.literal_eval(expr2_input)

        if not isinstance(expression2, list):

            raise ValueError("Expression must be a list.")




    except (SyntaxError, ValueError) as e:

        print(f"Invalid input format: {e}")

        return



    result = unify(expression1, expression2)



    if isinstance(result, str):

        print("\nUnification Result:")
```

```
        print(result)

    else:

        print("\nUnification Successful:")

        print(format_substitution(result))




if __name__ == "__main__":

    main()



name = "Varsha Prasanth"

usn = "1BM22CS321"

print(f"Name: {name}, USN: {usn}")
```

Output Snapshot

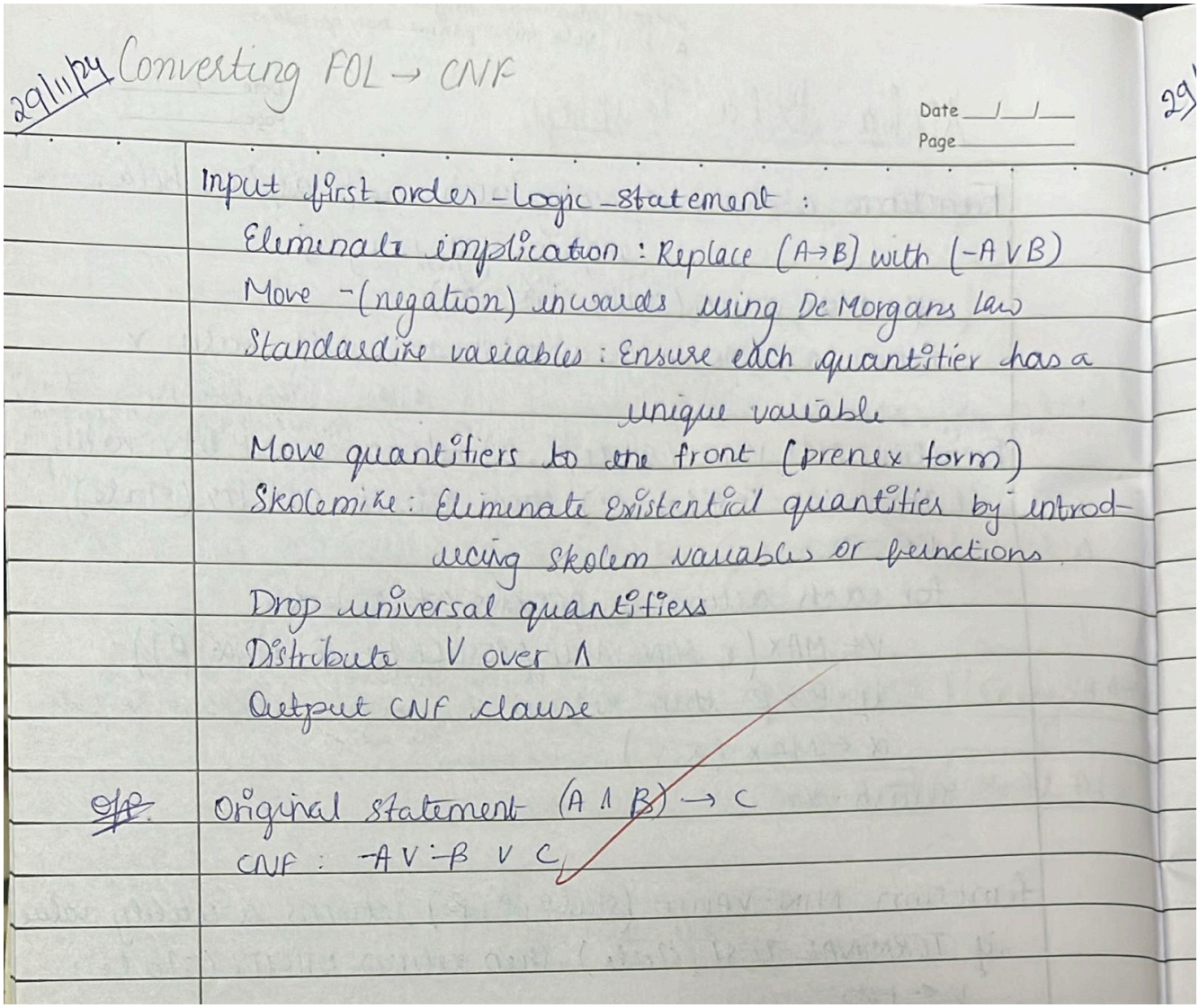
```
=== Unification Algorithm ===

Please enter the expressions in list format.
Example: ["Eats", "x", "Apple"]

Enter Expression 1:  ["Eats", "x", "Apple"]
Enter Expression 2:  ["Eats", "orange", "y"]

Unification Successful:
{ x = orange, y = Apple }
Name: Varsha Prasanth, USN: 1BM22CS321
```

STATE SPACE TREE



Program-10 First Order Logic to Conjunctive Normal Form

Algorithm



29/11/24 Converting FOL → CNF

Date __/__/__
Page

Input first order logic statement :
Eliminate implication : Replace (A→B) with (¬A ∨ B)
Move ¬(negation) inwards using De Morgans law
Standardize variables : Ensure each quantifier has a unique variable
Move quantifiers to the front (prenex form)
Skolemize : Eliminate Existential quantities by introducing skolem variables or functions
Drop universal quantifiers
Distribute ∨ over ∧
Output CNF clause

Original statement (A ∧ B) → C
CNF : ¬A ∨ ¬B ∨ C

Code

```
from sympy.logic.boolalg import Or, And, Not, Implies, Equivalent
from sympy import symbols


def eliminate_implications(expr):
    """Eliminate implications and equivalences."""
    if isinstance(expr, Implies):
        return Or(Not(eliminate_implications(expr.args[0])),
                  eliminate_implications(expr.args[1]))
    elif isinstance(expr, Equivalent):
        left = eliminate_implications(expr.args[0])
        right = eliminate_implications(expr.args[1])
        return And(Or(Not(left), right), Or(Not(right), left))
    elif expr.is_Atom:
        return expr
    else:
        return expr.func(*[eliminate_implications(arg) for arg in expr.args])


def push_negations(expr):
    """Push negations inward using De Morgan's laws."""
    if expr.is_Not:
        arg = expr.args[0]
        if isinstance(arg, And):
            return Or(*[push_negations(Not(sub_arg)) for sub_arg in arg.args])
        elif isinstance(arg, Or):
            return And(*[push_negations(Not(sub_arg)) for sub_arg in
arg.args])
        elif isinstance(arg, Not):
            return push_negations(arg.args[0])
        else:
            return Not(push_negations(arg))
    elif expr.is_Atom:
        return expr
    else:
        return expr.func(*[push_negations(arg) for arg in expr.args])
```

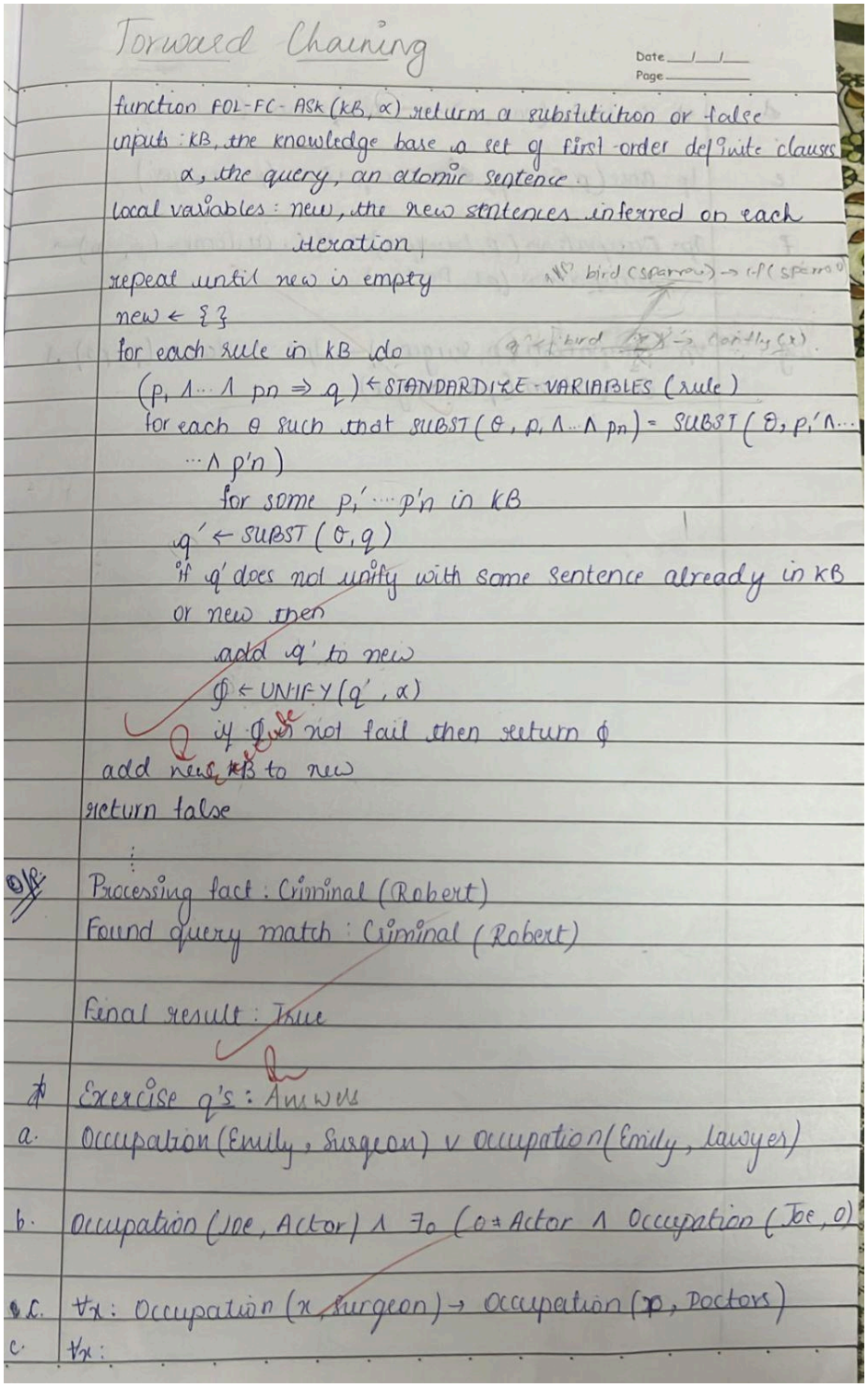
```
def distribute_ands(expr):
    """Distribute AND over OR to obtain CNF."""
    if isinstance(expr, Or):
        and_args = [arg for arg in expr.args if isinstance(arg, And)]
        if and_args:
            first_and = and_args[0]
            rest = [arg for arg in expr.args if arg != first_and]
            return And(*[distribute_ands(Or(arg, *rest)) for arg in
first_and.args])
    elif isinstance(expr, And) or expr.is_Atom or expr.is_Not:
        return expr
    return expr.func(*[distribute_ands(arg) for arg in expr.args])

def to_cnf(expr):
    """Convert the given logical expression to CNF."""
    expr = eliminate_implications(expr)
    expr = push_negations(expr)
    expr = distribute_ands(expr)
    return expr

# Example usage:
A, B, C = symbols('A B C')
fol_expr = Implies(A, Or(B, Not(C)))  # Example FOL expression
cnf_expr = to_cnf(fol_expr)
print("FOL expression:", fol_expr)
print("CNF expression:", cnf_expr)
name = "Varsha Prasanth"
usn = "1BM22CS321"
print(f"Name: {name}, USN: {usn}")
```

Output Snapshot

```
FOL expression: Implies(A, B | ~C)
CNF expression: B | ~A | ~C
Name: Varsha Prasanth, USN: 1BM22CS321
```

Program-11 Forward Reasoning

Algorithm



Forward Chaining

Date / /
Page

function FOL-FC-ASK(KB, α) returns a substitution or false
inputs: 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
(p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-VARIABLES(rule)
for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)
for some p'1, ... p'n in KB
q' ← SUBST(θ, q)
if q' does not unify with some sentence already in KB or new then
add q' to new
φ ← UNIFY(q', α)
if φ is not fail then return φ
add new to KB
return false

Processing fact: Criminal(Robert)
Found query match: Criminal(Robert)

Final result: True

Exercise q's: Answers

a. Occupation(Emily, Surgeon) ∨ Occupation(Emily, Lawyer)

b. Occupation(Joe, Actor) ∧ ∃o (o = Actor ∧ Occupation(Joe, o))

c. ∀x: Occupation(x, Surgeon) → Occupation(x, Doctors)

c. ∀x:

Code

```
def fol_fc_ask(KB, query):
    """
    Implements the Forward Chaining algorithm.
    :param KB: The knowledge base, a list of first-order definite clauses.
    :param query: The query, an atomic sentence.
    :return: True if the query can be proven, otherwise False.
    """
    inferred = set()  # Keep track of inferred facts
    agenda = [fact for fact in KB if not fact.get('premises')]  # Initial
facts
    rules = [rule for rule in KB if rule.get('premises')]  # Rules with
premises

    # Debugging output: Initial agenda and inferred facts
    print(f"Initial agenda: {[fact['conclusion'] for fact in agenda]}")
    print(f"Initial inferred: {inferred}")

    while agenda:
        fact = agenda.pop(0)
        print(f"\nProcessing fact: {fact['conclusion']}")

        # Check if this fact matches the query
        if fact['conclusion'] == query:
            print(f"Found query match: {fact['conclusion']}")
            return True

        # Infer new facts if this fact hasn't been inferred before
        if fact['conclusion'] not in inferred:
            inferred.add(fact['conclusion'])
            print(f"Inferred facts: {inferred}")

            # Process rules that match this fact as a premise
            for rule in rules:
                if fact['conclusion'] in rule['premises']:
                    print(f"Rule premise satisfied: {rule['premises']} ->
{rule['conclusion']}")
                    rule['premises'].remove(fact['conclusion'])  # Remove
satisfied premise
                    if not rule['premises']:  # All premises satisfied
                        new_fact = {'conclusion': rule['conclusion']}
                        agenda.append(new_fact)  # Add new fact to agenda
```

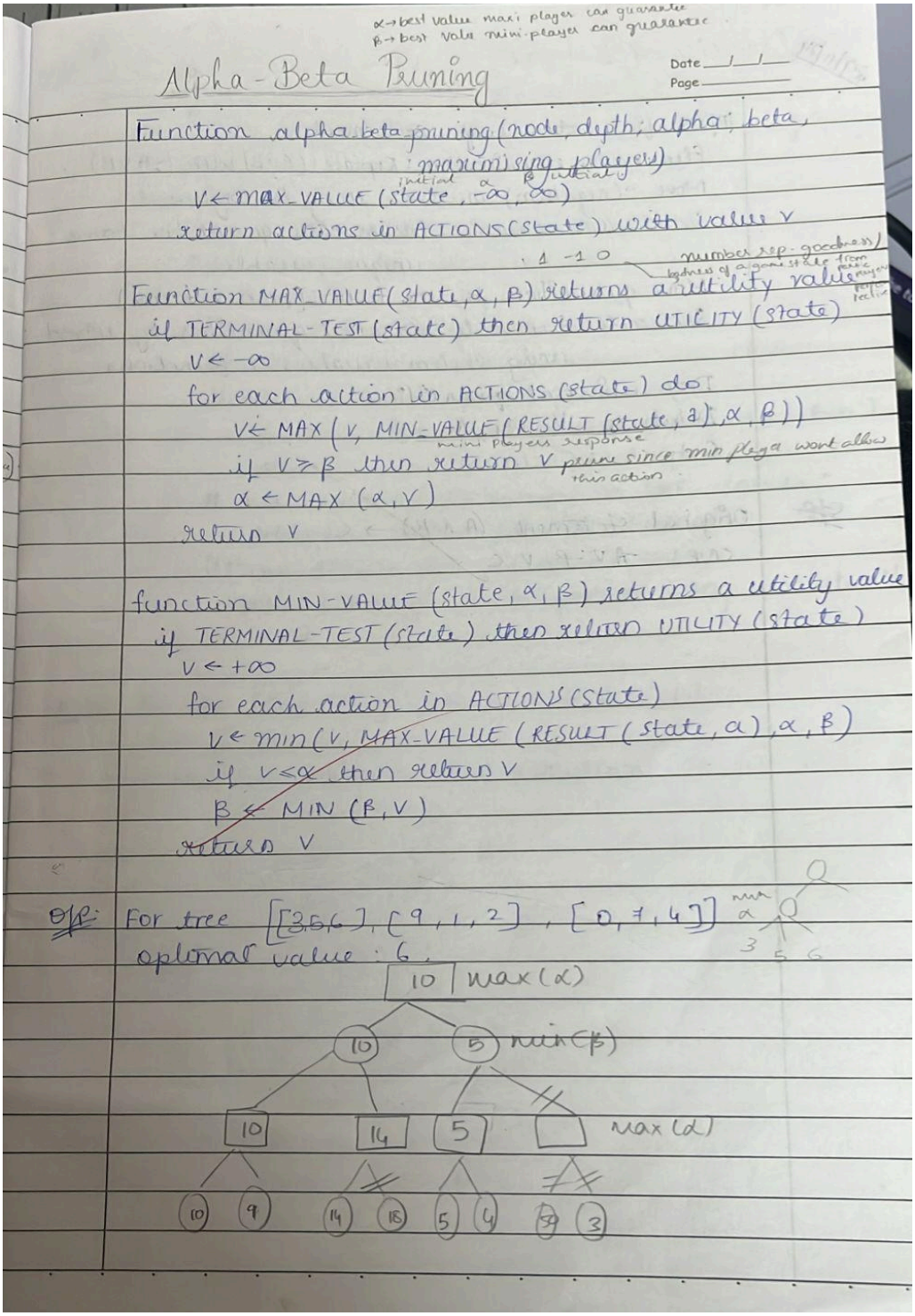
```
                    print(f"New fact inferred: {new_fact['conclusion']}")

        # Debugging output after each iteration
        print(f"Current agenda: {[fact['conclusion'] for fact in agenda]}")
        print(f"Current inferred: {inferred}")

    # If the loop finishes without finding the query
    print(f"Query {query} not found.")
    return False


# Example Knowledge Base
KB = [
    {'premises': [], 'conclusion': 'American(Robert)'},
    {'premises': [], 'conclusion': 'Missile(T1)'},
    {'premises': [], 'conclusion': 'Owns(A, T1)'},
    {'premises': [], 'conclusion': 'Enemy(A, America)'},
    {'premises': ['Missile(T1)'], 'conclusion': 'Weapon(T1)'},
    {'premises': ['American(Robert)', 'Weapon(T1)', 'Sells(Robert, T1, A)',
'Hostile(A)'], 'conclusion': 'Criminal(Robert)'},
    {'premises': ['Owns(A, T1)', 'Enemy(A, America)'], 'conclusion':
'Hostile(A)'},
    {'premises': [], 'conclusion': 'Sells(Robert, T1, A)'}
]

# Query
query = 'Criminal(Robert)'

# Run the algorithm
result = fol_fc_ask(KB, query)
print("\nFinal Result:", result)
name = "Varsha Prasanth"
usn = "1BM22CS321"
print(f"Name: {name}, USN: {usn}")
```

OutputSnapshot

```
Initial agenda: ['American(Robert)', 'Missile(T1)', 'Owns(A, T1)', 'Enemy(A, America)', 'Sells(Robert, T1, A)']
Initial inferred: set()

Processing fact: American(Robert)
Inferred facts: {'American(Robert)'}
Rule premise satisfied: ['American(Robert)', 'Weapon(T1)', 'Sells(Robert, T1, A)', 'Hostile(A)'] -> Criminal(Robert)
Current agenda: ['Missile(T1)', 'Owns(A, T1)', 'Enemy(A, America)', 'Sells(Robert, T1, A)']
Current inferred: {'American(Robert)'}

Processing fact: Missile(T1)
Inferred facts: {'American(Robert)', 'Missile(T1)'}
Rule premise satisfied: ['Missile(T1)'] -> Weapon(T1)
New fact inferred: Weapon(T1)
Current agenda: ['Owns(A, T1)', 'Enemy(A, America)', 'Sells(Robert, T1, A)', 'Weapon(T1)']
Current inferred: {'American(Robert)', 'Missile(T1)'}

Processing fact: Owns(A, T1)
Inferred facts: {'Owns(A, T1)', 'American(Robert)', 'Missile(T1)'}
Rule premise satisfied: ['Owns(A, T1)', 'Enemy(A, America)'] -> Hostile(A)
Current agenda: ['Enemy(A, America)', 'Sells(Robert, T1, A)', 'Weapon(T1)']
Current inferred: {'Owns(A, T1)', 'American(Robert)', 'Missile(T1)'}

Processing fact: Enemy(A, America)
Inferred facts: {'Owns(A, T1)', 'American(Robert)', 'Missile(T1)', 'Enemy(A, America)'}
Rule premise satisfied: ['Enemy(A, America)'] -> Hostile(A)
New fact inferred: Hostile(A)
Current agenda: ['Sells(Robert, T1, A)', 'Weapon(T1)', 'Hostile(A)']
Current inferred: {'Owns(A, T1)', 'American(Robert)', 'Missile(T1)', 'Enemy(A, America)'}

Processing fact: Sells(Robert, T1, A)
Inferred facts: {'American(Robert)', 'Missile(T1)', 'Owns(A, T1)', 'Sells(Robert, T1, A)', 'Enemy(A, America)'}
Rule premise satisfied: ['Weapon(T1)', 'Sells(Robert, T1, A)', 'Hostile(A)'] -> Criminal(Robert)
Current agenda: ['Weapon(T1)', 'Hostile(A)']
Current inferred: {'American(Robert)', 'Missile(T1)', 'Owns(A, T1)', 'Sells(Robert, T1, A)', 'Enemy(A, America)'}

Processing fact: Weapon(T1)
Inferred facts: {'American(Robert)', 'Missile(T1)', 'Owns(A, T1)', 'Sells(Robert, T1, A)', 'Weapon(T1)', 'Enemy(A, America)'}
Rule premise satisfied: ['Weapon(T1)', 'Hostile(A)'] -> Criminal(Robert)
Current agenda: ['Hostile(A)']
Current inferred: {'American(Robert)', 'Missile(T1)', 'Owns(A, T1)', 'Sells(Robert, T1, A)', 'Weapon(T1)', 'Enemy(A, America)'}

Processing fact: Hostile(A)
Inferred facts: {'American(Robert)', 'Missile(T1)', 'Owns(A, T1)', 'Hostile(A)', 'Sells(Robert, T1, A)', 'Weapon(T1)', 'Enemy(A, America)'}
Rule premise satisfied: ['Hostile(A)'] -> Criminal(Robert)
New fact inferred: Criminal(Robert)
Current agenda: ['Criminal(Robert)']
Current inferred: {'American(Robert)', 'Missile(T1)', 'Owns(A, T1)', 'Hostile(A)', 'Sells(Robert, T1, A)', 'Weapon(T1)', 'Enemy(A, America)'}

Processing fact: Criminal(Robert)
Found query match: Criminal(Robert)

Final Result: True
Name: Varsha Prasanth, USN: 1BM22CS321
```

Program-11 Alpha - Beta Pruning

Algorithm



Code

```
import math
```

```
# Alpha-Beta Pruning Algorithm
def alpha_beta_search(depth, index, is_max, values, alpha, beta,
target_depth):
    """Recursive function for Alpha-Beta Pruning."""
    # Base case: If the target depth is reached, return the leaf node value
    if depth == target_depth:
        return values[index]

    if is_max:
        # Maximizer's turn
        best = -math.inf
        for i in range(2):
            val = alpha_beta_search(depth + 1, index * 2 + i, False, values,
alpha, beta, target_depth)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                break  # Prune remaining branches
        return best
    else:
        # Minimizer's turn
        best = math.inf
        for i in range(2):
            val = alpha_beta_search(depth + 1, index * 2 + i, True, values,
alpha, beta, target_depth)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                break  # Prune remaining branches
        return best

def main():
    # User Input: Values of leaf nodes
    print("Enter the values of leaf nodes separated by spaces:")
    values = list(map(int, input().split()))

    # Calculate depth of the game tree
    target_depth = math.log2(len(values))
    if target_depth != int(target_depth):
        print("Error: The number of leaf nodes must be a power of 2.")
        return
    target_depth = int(target_depth)
```

```
    # Run Alpha-Beta Pruning
    result = alpha_beta_search(0, 0, True, values, -math.inf, math.inf,
target_depth)

    # Display the result
    print(f"The optimal value determined by Alpha-Beta Pruning is: {result}")

if __name__ == "__main__":
    main()
    name = "Varsha Prasanth"
    usn = "1BM22CS321"
    print(f"Name: {name}, USN: {usn}")
```

OUTPUT

```
Enter the values of leaf nodes separated by spaces:
10 9 14 18 5 4 50 3
The optimal value determined by Alpha-Beta Pruning is: 10
Name: Varsha Prasanth, USN: 1BM22CS321
```

STATE SPACE TREE