VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Tanush Prajwal S (1BM22CS304)

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 **Tanush Prajwal S (1BM22CS304),** who is a 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.

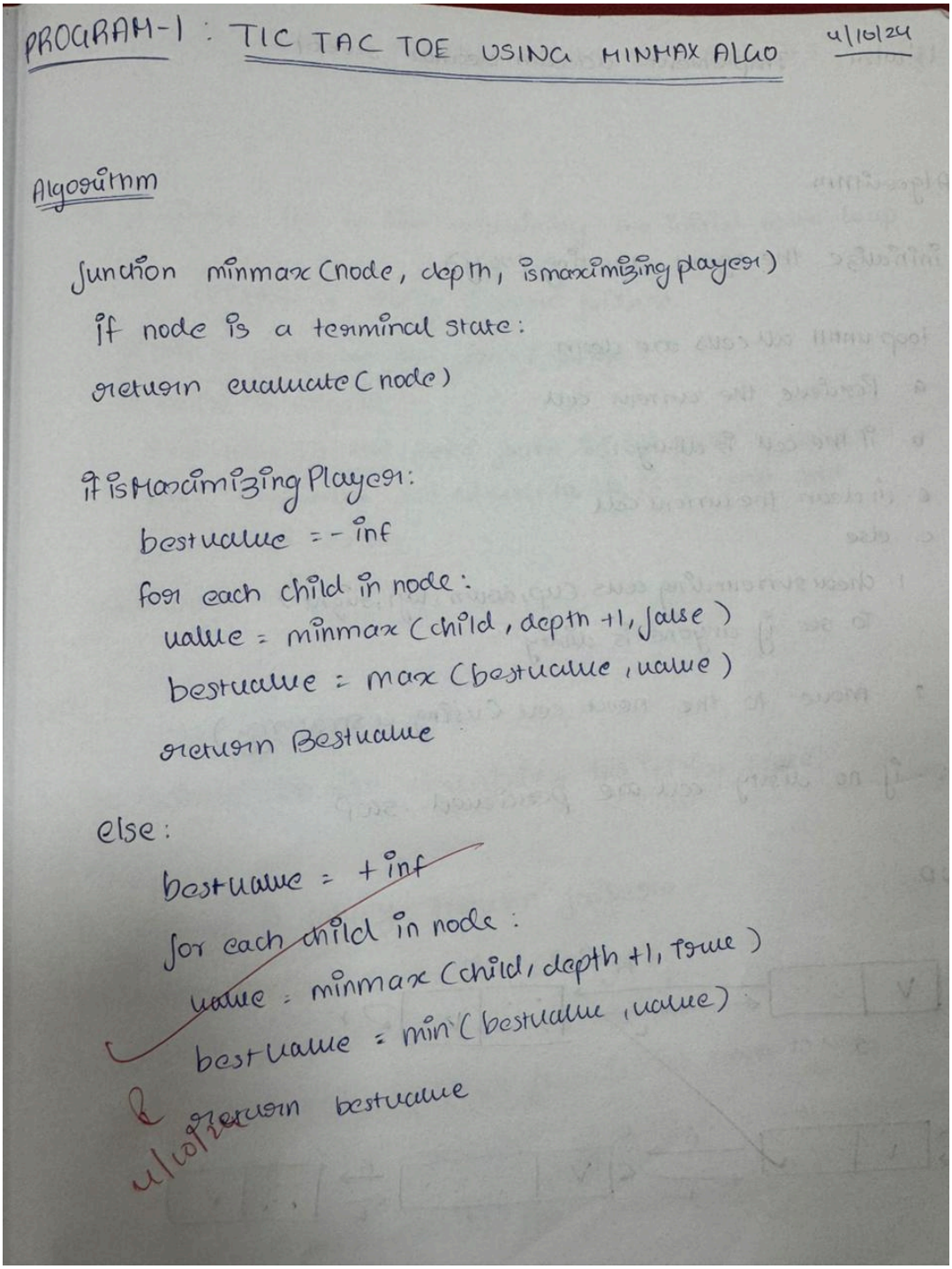
Prof. Rashmi H Associate Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Github Link: https://github.com/tanush082008/AI_LAB_Programs.git

Table of contents

Program 1 - Tic Tac toe

Algorithm



PROGRAM-1 : TIC TAC TOE USING MINMAX ALGO 4/10/24

Algorithm

function minmax(node, depth, ismaximizing player)
if node is a terminal state:
return evaluate(node)

if isMaximizing Player:
bestvalue = -inf
for each child in node:
value = minmax(child, depth+1, false)
bestvalue = max(bestvalue, value)
return Bestvalue

else:
bestvalue = +inf
for each child in node:
value = minmax(child, depth+1, true)
bestvalue = min(bestvalue, value)
return bestvalue

Code

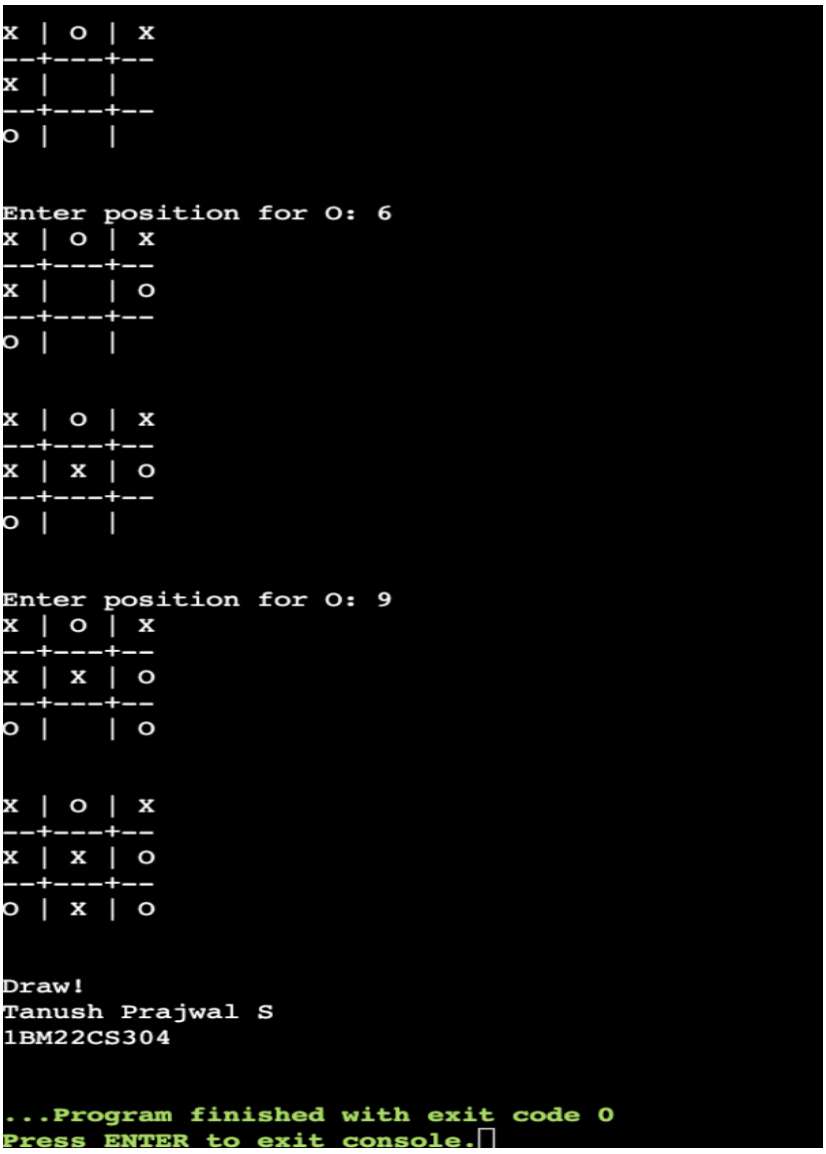
```
board = {1: ' ', 2: ' ', 3: ' ',
        4: ' ', 5: ' ', 6: ' ',
        7: ' ', 8: ' ', 9: ' '}


def printBoard(board):
    print(board[1] + '|' + board[2] + '|' + board[3])
    print('-+-+-')
    print(board[4] + '|' + board[5] + '|' + board[6])2
    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.')
```

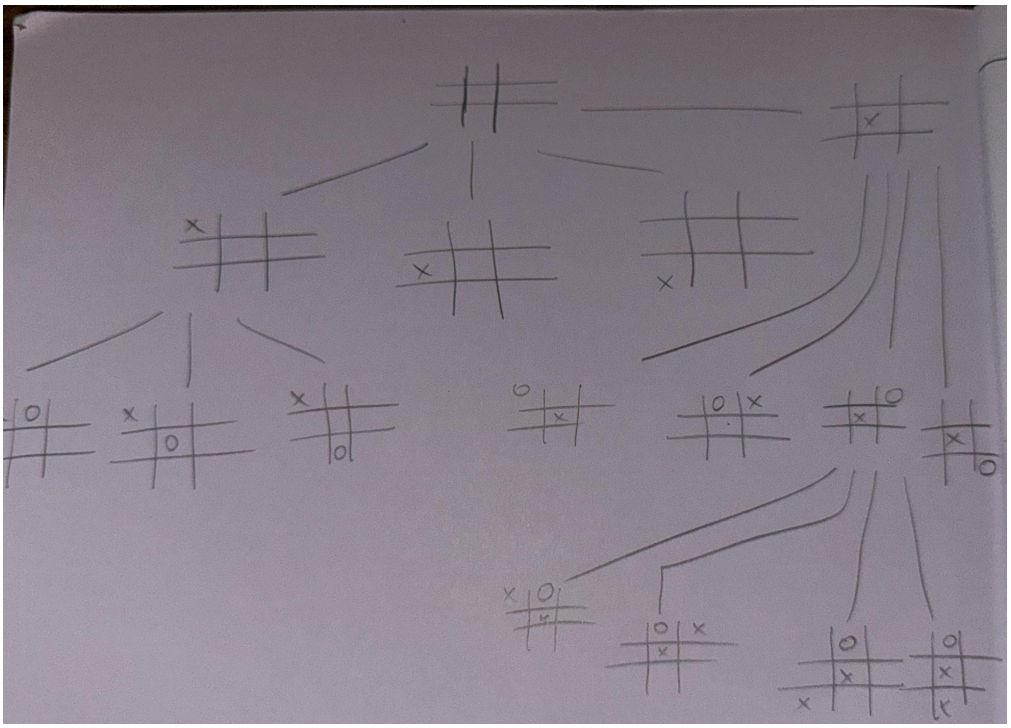
```
        position = int(input('Enter new position: '))
        insertLetter(letter, position)


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

print("Tanush Prajwal S")
print("1BM22CS304\n")

while not checkWin() and not checkDraw():
    compMove()
    if checkWin() or checkDraw():
        break
    playerMove()
```

Output Snapshot

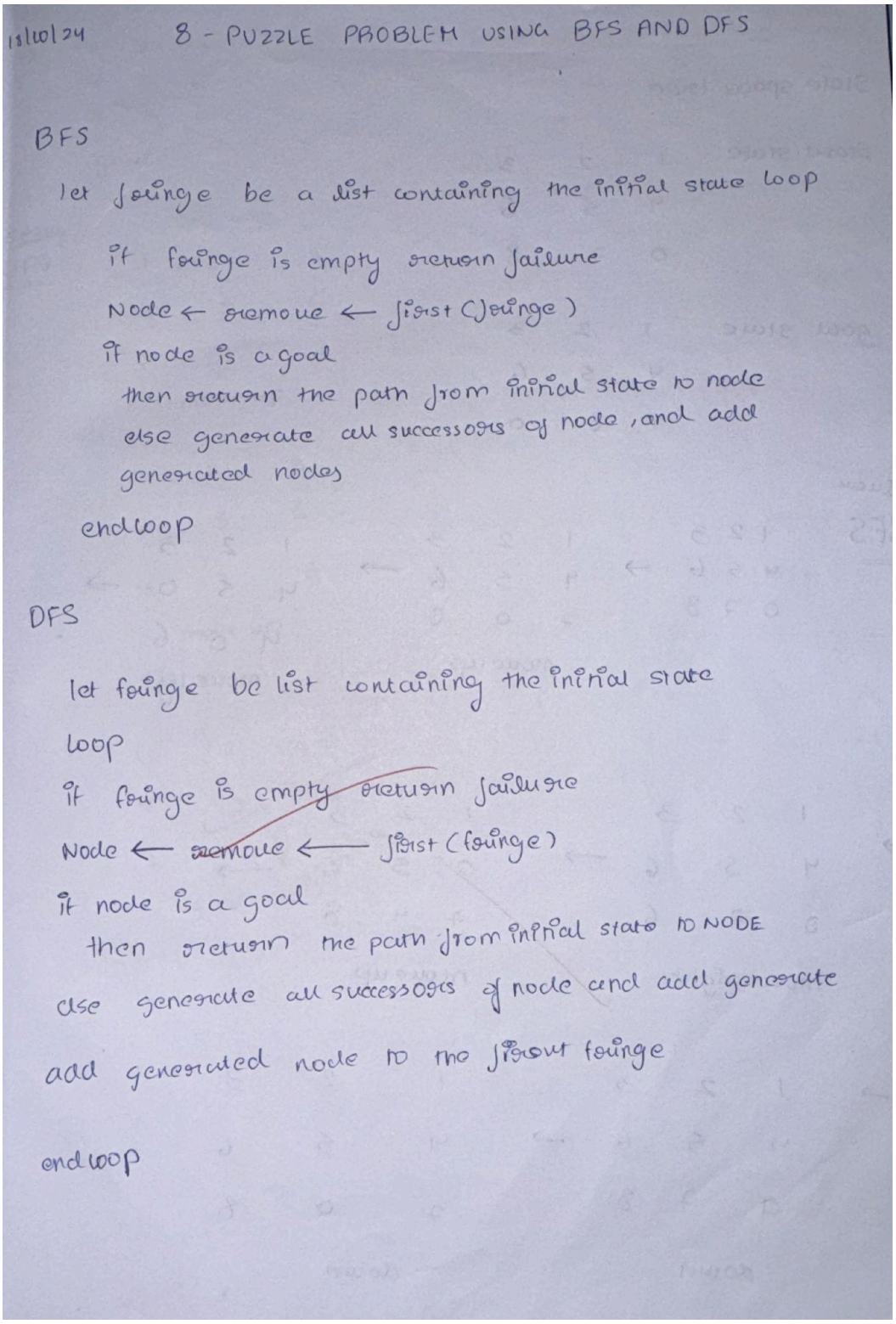


State Space Tree



Program 2 - 8 Puzzle Using BFS

Algorithm



15/10/24 8 - PUZZLE PROBLEM USING BFS AND DFS

BFS

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

endloop

DFS

let fringe be 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 generate

add generated node to the front fringe

endloop

Code

```
#BFS
from collections import deque

class PuzzleState:
    def __init__(self, board, zero_position, path=[]):
        self.board = board
        self.zero_position = zero_position
        self.path = path

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

    def get_possible_moves(self):
        moves = []
        row, col = self.zero_position
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down, Left, Up

        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_board = self.board[:]
                # Swap zero with the adjacent tile
                new_board[row * 3 + col], new_board[new_row * 3 + new_col] = new_board[new_row
* 3 + new_col], new_board[row * 3 + col]
                moves.append(PuzzleState(new_board, (new_row, new_col), self.path + [new_board]))
        return moves

def bfs(initial_state):
    queue = deque([initial_state])
    visited = set()

    while queue:
        current_state = queue.popleft()
        # Show the current board
```

```
        print("Current Board State:")
        print_board(current_state.board)
        print()

        if current_state.is_goal():
            return current_state.path
        visited.add(tuple(current_state.board))

        for next_state in current_state.get_possible_moves():
            if tuple(next_state.board) not in visited:
                queue.append(next_state)

    return None

def print_board(board):
    for i in range(3):
        print(board[i * 3:i * 3 + 3])

def main():
    print("Enter the initial state of the 8-puzzle (use 0 for the blank tile, e.g., '1 2 3 4 5 6 7 8 0'): ")
    user_input = input()
    initial_board = list(map(int, user_input.split()))

    if len(initial_board) != 9 or set(initial_board) != set(range(9)):
        print("Invalid input! Please enter 9 numbers from 0 to 8.")
        return

    zero_position = initial_board.index(0)
    initial_state = PuzzleState(initial_board, (zero_position // 3, zero_position % 3))
    solution_path = bfs(initial_state)

    if solution_path is None:
        print("No solution found.")
    else:
        print("Solution found in", len(solution_path), "steps.")
        for step in solution_path:
            print_board(step)
            print()

if __name__ == "__main__":
    main()

print("-------------")
```

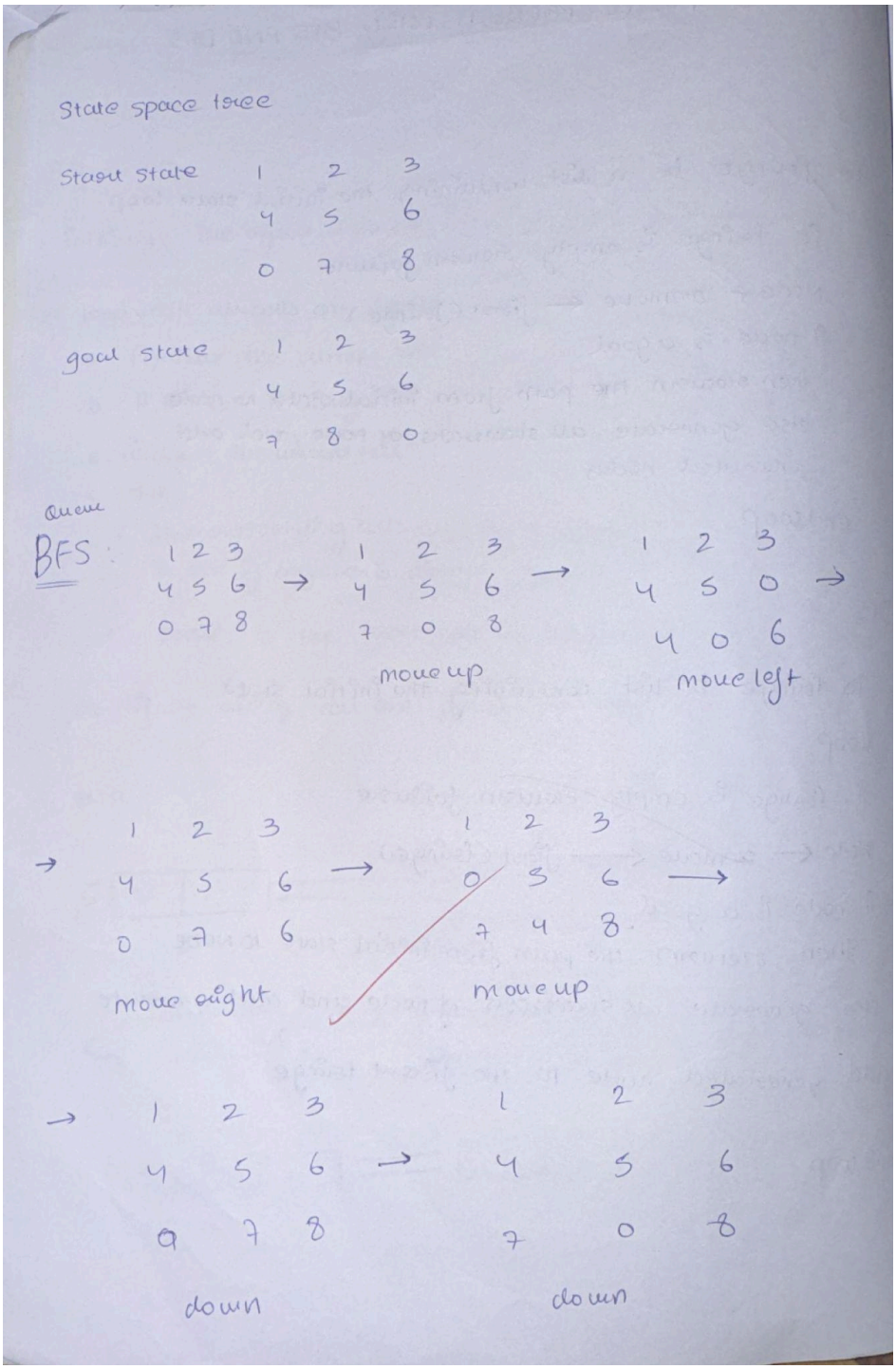
```
print("Tanush Prajwal S")
print("1BM22CS304")
```

Output Snapshot

Output Clear

```
Enter the initial state of the 8-puzzle (use 0 for the blank tile, e.g., '1
    2 3 4 5 6 7 8 0'):
1 2 3 4 5 6 0 7 8
Solution found in 2 steps:
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]

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

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

Tanush Prajwal S
1BM22CS304

=== Code Execution Successful ===
```

State Space Tree



8 puzzle using DFS

Code

```
from collections import deque

print("Tanush Prajwal S")
print("1BM22CS304")
print("--------")

def get_user_input(prompt):
    board = []
    print(prompt)
    for i in range(3):
        row = list(map(int, input(f"Enter row {i + 1} (space-separated numbers, use 0 for empty space):
").split()))
        board.append(row)
    return board

def is_solvable(board):
    flattened_board = [tile for row in board for tile in row if tile != 0]
    inversions = 0
    for i in range(len(flattened_board)):
        for j in range(i + 1, len(flattened_board)):
            if flattened_board[i] > flattened_board[j]:
                inversions += 1
    return inversions % 2 == 0

class PuzzleState:
    def __init__(self, board, moves=0, previous=None):
```

```
        self.board = board
        self.empty_tile = self.find_empty_tile()
        self.moves = moves
        self.previous = previous

    def find_empty_tile(self):
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return (i, j)

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

    def get_possible_moves(self):
        row, col = self.empty_tile
        possible_moves = []
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]  # down, up, right, left
        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                # Make the move
                new_board = [row[:] for row in self.board]  # Deep copy
                new_board[row][col], new_board[new_row][new_col] = new_board[new_row][new_col],
new_board[row][col]
                possible_moves.append(PuzzleState(new_board, self.moves + 1, self))
        return possible_moves

def dfs(initial_state, goal_state):
    stack = [initial_state]
    visited = set()
    while stack:
        current_state = stack.pop()
        # If we find the goal, return the state
        if current_state.is_goal(goal_state):
            return current_state
        # Convert board to a tuple for the visited set
```

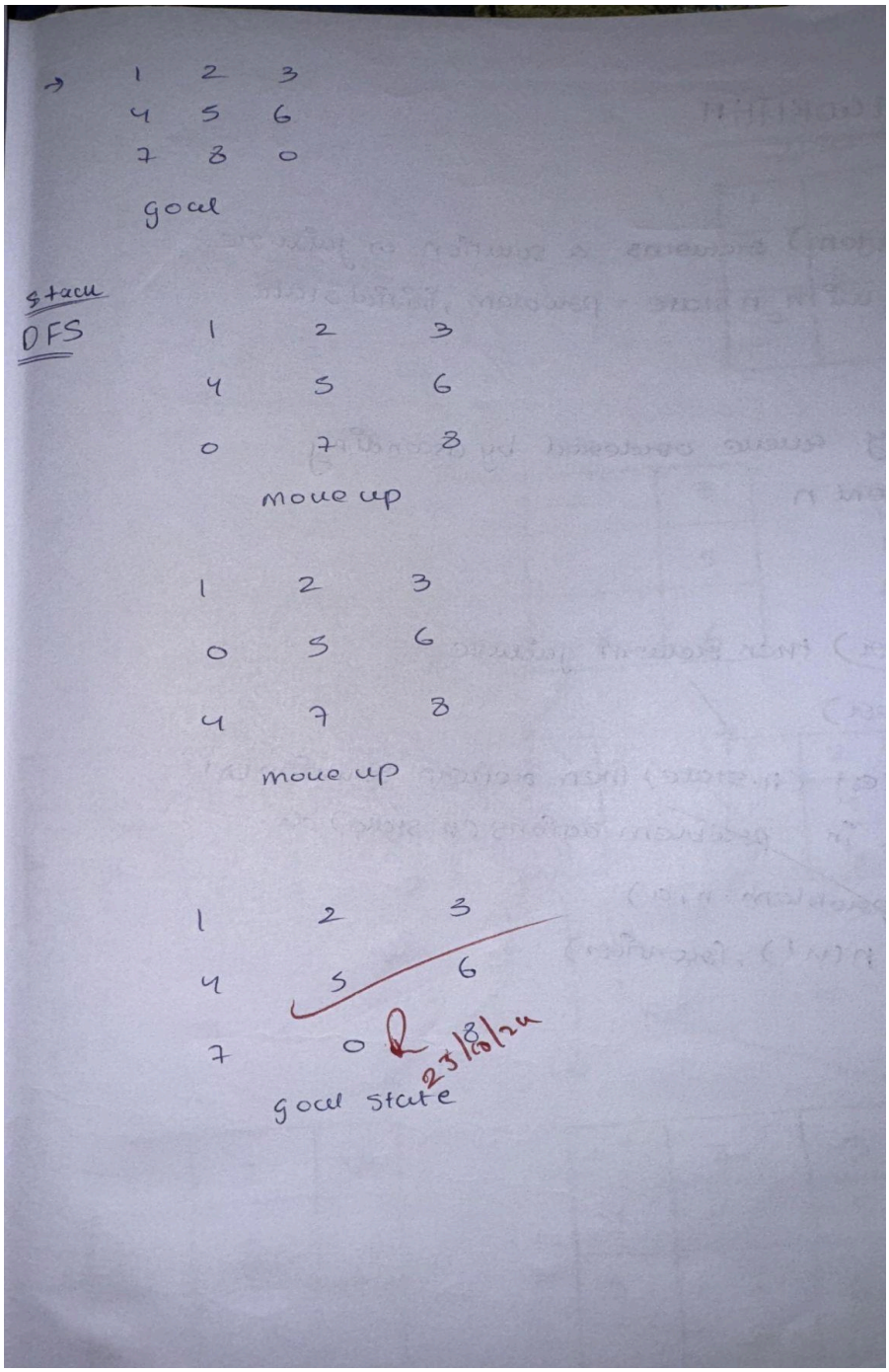
```
        state_tuple = tuple(tuple(row) for row in current_state.board)
        # If we've already visited this state, skip it
        if state_tuple not in visited:
            visited.add(state_tuple)
            for next_state in current_state.get_possible_moves():
                stack.append(next_state)
    return None  # No solution found

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

if __name__ == "__main__":
    # Get user input for initial and goal states
    initial_board = get_user_input("Enter the initial state of the puzzle:")
    goal_board = get_user_input("Enter the goal state of the puzzle:")

    if is_solvable(initial_board):
        initial_state = PuzzleState(initial_board)
        solution = dfs(initial_state, goal_board)
        if solution:
            print("Solution found in", solution.moves, "moves:")
            print_solution(solution)
        else:
            print("No solution found.")
    else:
        print("This puzzle is unsolvable.")
```

Output Snapshot

Output Clear

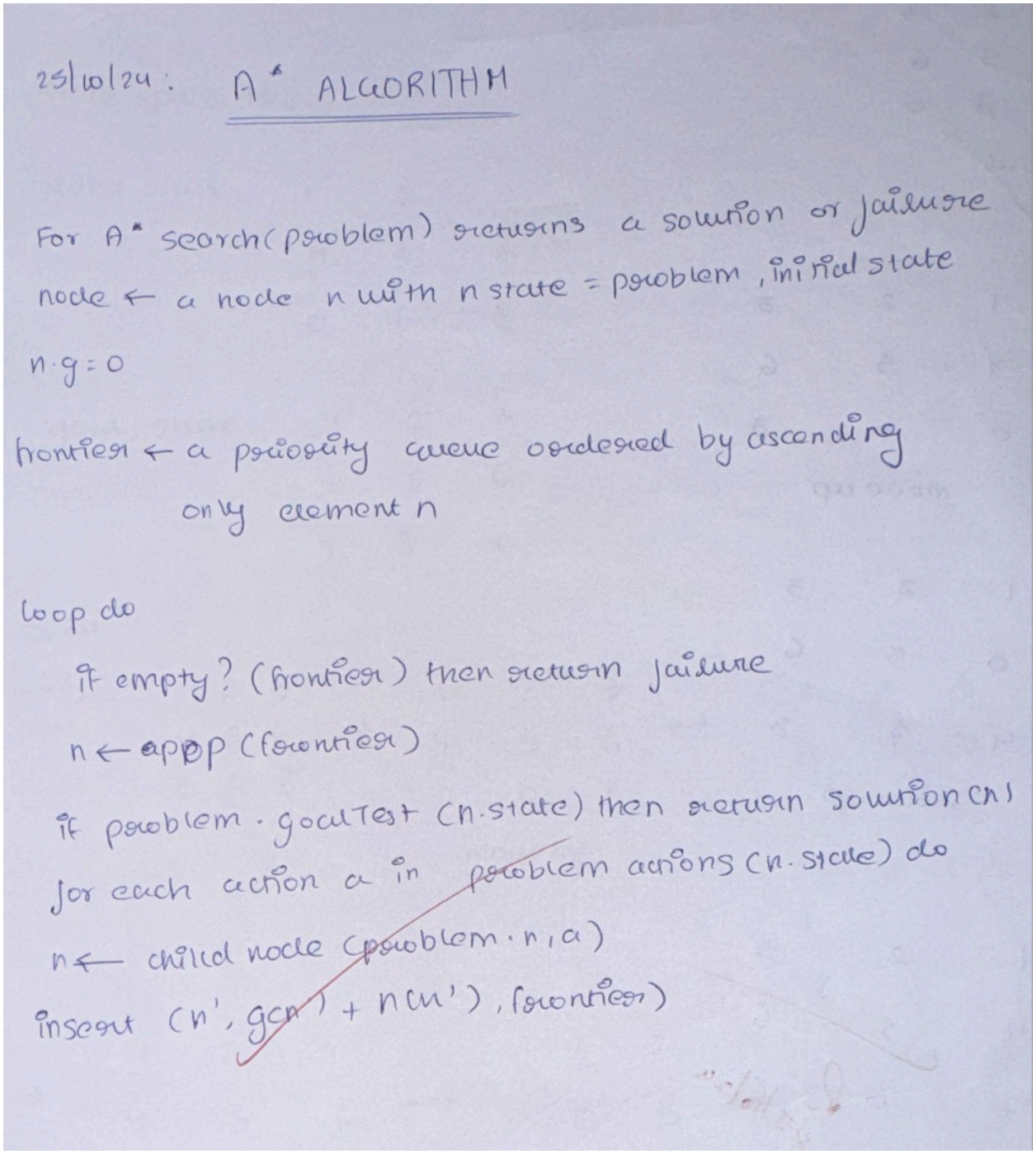
```
Tanush Prajwal S
1BM22CS304
-----------
Enter the initial state of the puzzle:
Enter row 1 (space-separated numbers, use 0 for empty space): 1 2 3
Enter row 2 (space-separated numbers, use 0 for empty space): 4 0 5
Enter row 3 (space-separated numbers, use 0 for empty space): 7 8 6
Enter the goal state of the puzzle:
Enter row 1 (space-separated numbers, use 0 for empty space): 1 2 3
Enter row 2 (space-separated numbers, use 0 for empty space): 4 5 6
Enter row 3 (space-separated numbers, use 0 for empty space): 7 8 0
Solution found in 30 moves:
[1, 2, 3]
[4, 0, 5]
[7, 8, 6]

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

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

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

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

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

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

StateSpaceTree



Program 03 - 8 Puzzle Using A*

Algorithm



Code

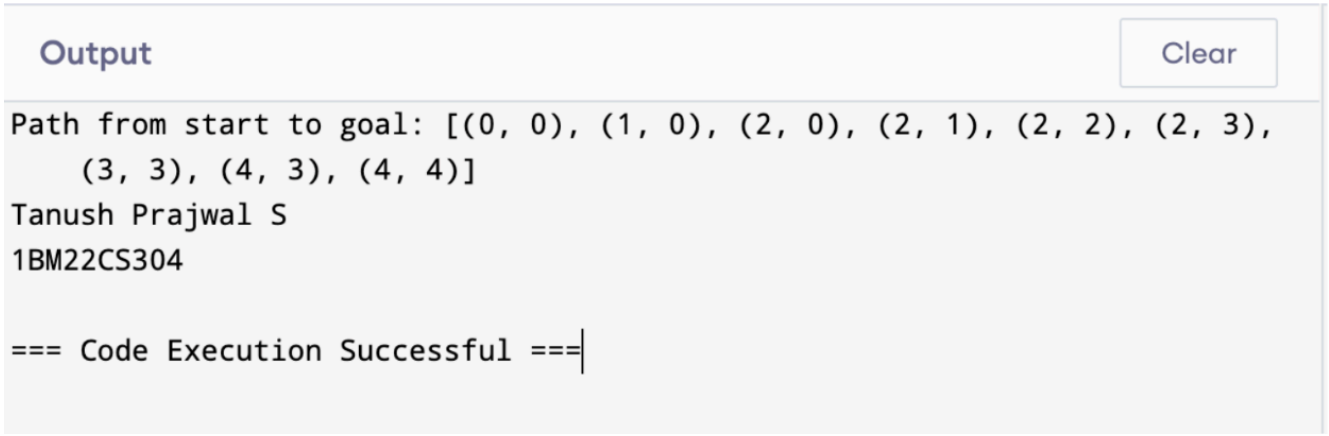
MANHATTAN DISTANCE

```
#Manhattan Distance
import heap
class Node:
    def __init__(self, position, parent=None):
        self.position = position
        self.parent = parent
        self.g = 0  # Cost from start to this node
        self.h = 0  # Heuristic cost from this node to target
        self.f = 0  # Total cost

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

def heuristic(a, b):
    # Manhattan distance
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def astar(start, goal, grid):
    open_list = []
    closed_list = set()
    start_node = Node(start)
    goal_node = Node(goal)
    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)
```

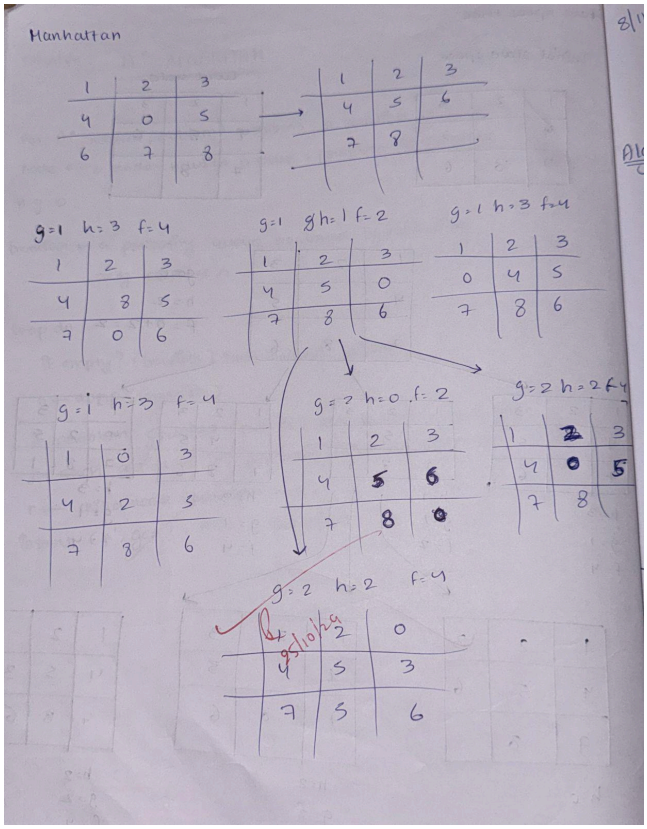
```
    closed_list.add(current_node.position)

    # Goal check
    if current_node.position == goal:
        path = []
        while current_node:
            path.append(current_node.position)
            current_node = current_node.parent
        return path[::-1]  # Return reversed path

    # Generate neighbors
    neighbors = [
        (current_node.position[0] + dx, current_node.position[1] + dy)
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]
    ]

    for next_position in neighbors:
        # Check if within bounds and not a wall (assuming 0 is free space)
        if (0 <= next_position[0] < len(grid) and
            0 <= next_position[1] < len(grid[0]) and
            grid[next_position[0]][next_position[1]] == 0):

            if next_position in closed_list:
                continue

            neighbor_node = Node(next_position, current_node)
            neighbor_node.g = current_node.g + 1
            neighbor_node.h = heuristic(next_position, goal)
            neighbor_node.f = neighbor_node.g + neighbor_node.h

            # Check if this neighbor is already in the open list
            if any(neighbor.position == neighbor_node.position and neighbor.f <= neighbor_node.f for
neighbor in open_list):
                continue

            heapq.heappush(open_list, neighbor_node)
```

```
    return []  # Return empty path if no path found

# Example usage
if __name__ == "__main__":
    grid = [
        [0, 0, 0, 0, 0],
        [0, 1, 1, 1, 0],
        [0, 0, 0, 0, 0],
        [0, 1, 1, 0, 0],
        [0, 0, 0, 0, 0]
    ]
    start = (0, 0)
    goal = (4, 4)
    path = astar(start, goal, grid)
    print("Path from start to goal:", path)
    print("Tanush Prajwal S")
    print("1BM22CS304")
```

Output Snapshot



State Space Tree



MISPLACED TILES

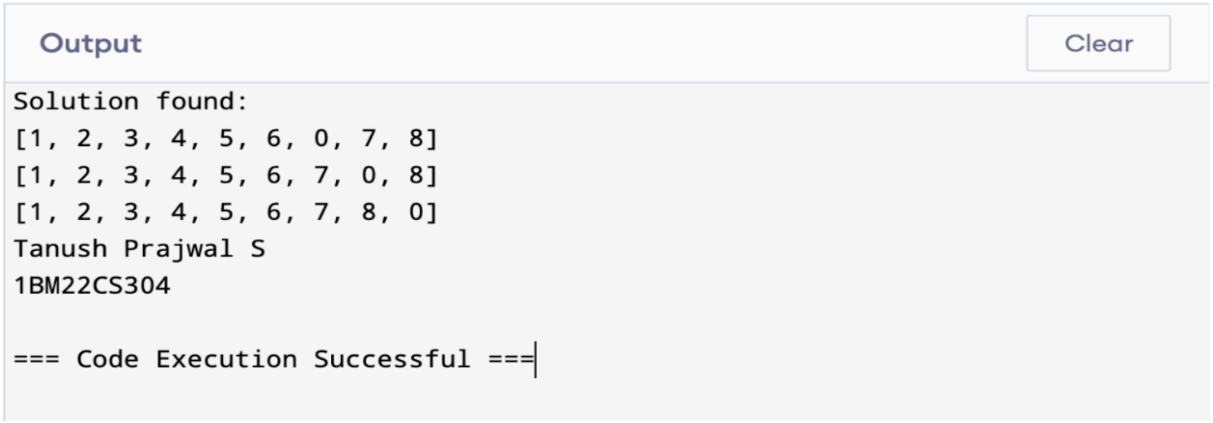
```
#Misplaced Tiles
import heapq


class PuzzleState:
    def __init__(self, board, g=0):
```

```
        self.board = board
        self.g = g  # Cost from start to this state
        self.zero_pos = board.index(0)  # Position of the empty space

    def h(self):
        # Calculate the number of misplaced tiles (Misplaced Tile Heuristic)
        return sum(1 for i in range(9) if self.board[i] != 0 and self.board[i] != i + 1)

    def f(self):
        return self.g + self.h()

    def get_neighbors(self):
        neighbors = []
        x, y = divmod(self.zero_pos, 3)
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_zero_pos = new_x * 3 + new_y
                new_board = self.board[:]
                # Swap zero with the neighboring tile
                new_board[self.zero_pos], new_board[new_zero_pos] = new_board[new_zero_pos], new_board[self.zero_pos]
                neighbors.append(PuzzleState(new_board, self.g + 1))
        return neighbors

def a_star(initial_state, goal_state):
    open_set = []
    heapq.heappush(open_set, (initial_state.f(), 0, initial_state))  # Add a unique identifier (0 in this case)
    came_from = {}
```

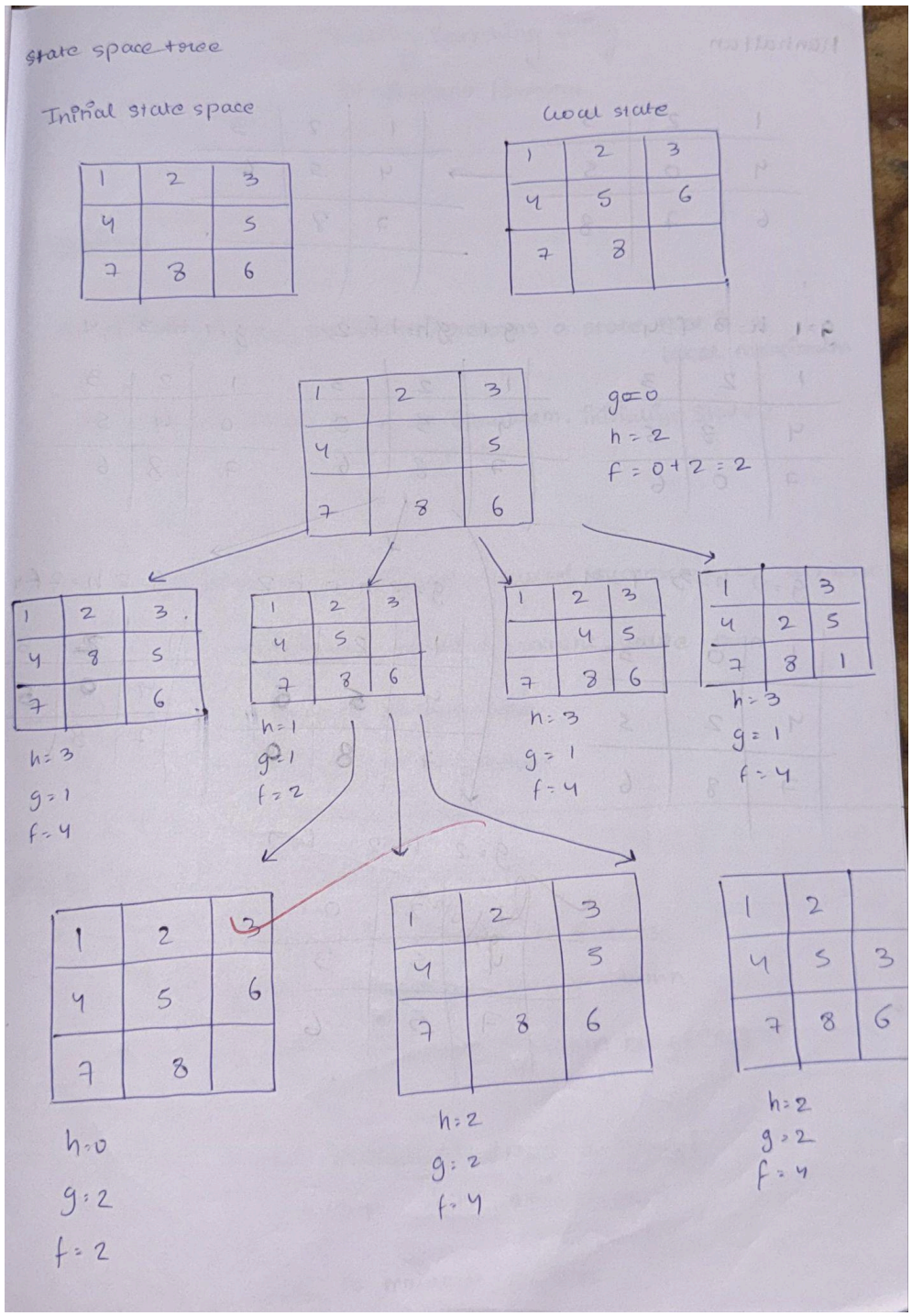
```
    g_score = {tuple(initial_state.board): 0}

    while open_set:
        current_f, _, current = heapq.heappop(open_set)

        if current.board == goal_state:
            return reconstruct_path(came_from, current)

        for neighbor in current.get_neighbors():
            neighbor_tuple = tuple(neighbor.board)
            tentative_g_score = g_score[tuple(current.board)] + 1

            if neighbor_tuple not in g_score or tentative_g_score < g_score[neighbor_tuple]:
                came_from[neighbor_tuple] = current
                g_score[neighbor_tuple] = tentative_g_score
                heapq.heappush(open_set, (neighbor.f(), neighbor.g, neighbor))

    return None  # No solution found

def reconstruct_path(came_from, current):
    path = []
    while current is not None:
        path.append(current.board)
        current = came_from.get(tuple(current.board), None)
    return path[::-1]

# Example usage
if __name__ == "__main__":
    initial_state = PuzzleState([1, 2, 3, 4, 5, 6, 0, 7, 8])
    goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
```

```
solution = a_star(initial_state, goal_state)


if solution:
    for step in solution:
        print(step)
else:
    print("No solution found")


print("Tanush Prajwal S")
print("1BM22CS304")
```

Output Snapshot

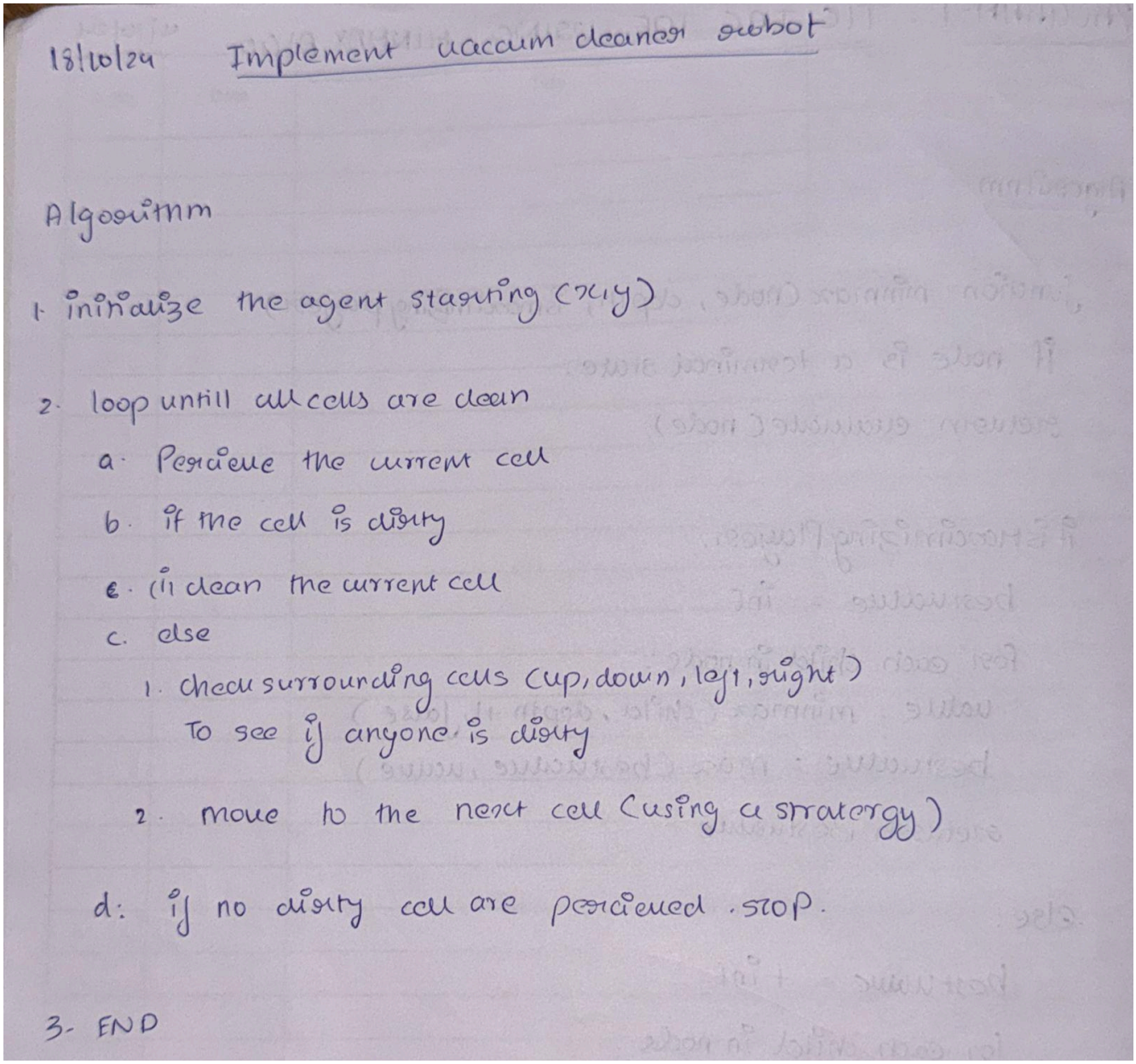


State Space Tree


state space tree
Initial state space
Goal state
g=0
h=2
f=0+2=2
h=3
g=1
f=4
h=1
g=1
f=2
h=3
g=1
f=4
h=3
g=1
f=4
h=0
g=2
f=2
h=2
g=2
f=4
h=2
g=2
f=4

Program 4 - Vacuum Cleaner

Algorithm



18/10/24 Implement vaccum cleaner robot

Algorithm

1. initialize the agent starting (x,y)

2. loop until all cells are clean
   a. Percieve the current cell
   b. if the cell is dirty
   c. (i) clean the current cell
   c. else
      1. check surrounding cells (up, down, left, right) To see if anyone is dirty
      2. move to the next cell (using a strategy)

   d: if no dirty cell are percieved . stop.

3. END

Code

```
def vacuum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0

    location_input = input("Enter Location of Vacuum: ")
    status_input = input("Enter status of " + location_input + " (0 for Clean, 1 for Dirty): ")
    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':
        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")
            goal_state['A'] = '0'
            cost += 1
            print("Cost for CLEANING A: " + str(cost))
            print("Location A has been Cleaned.")

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

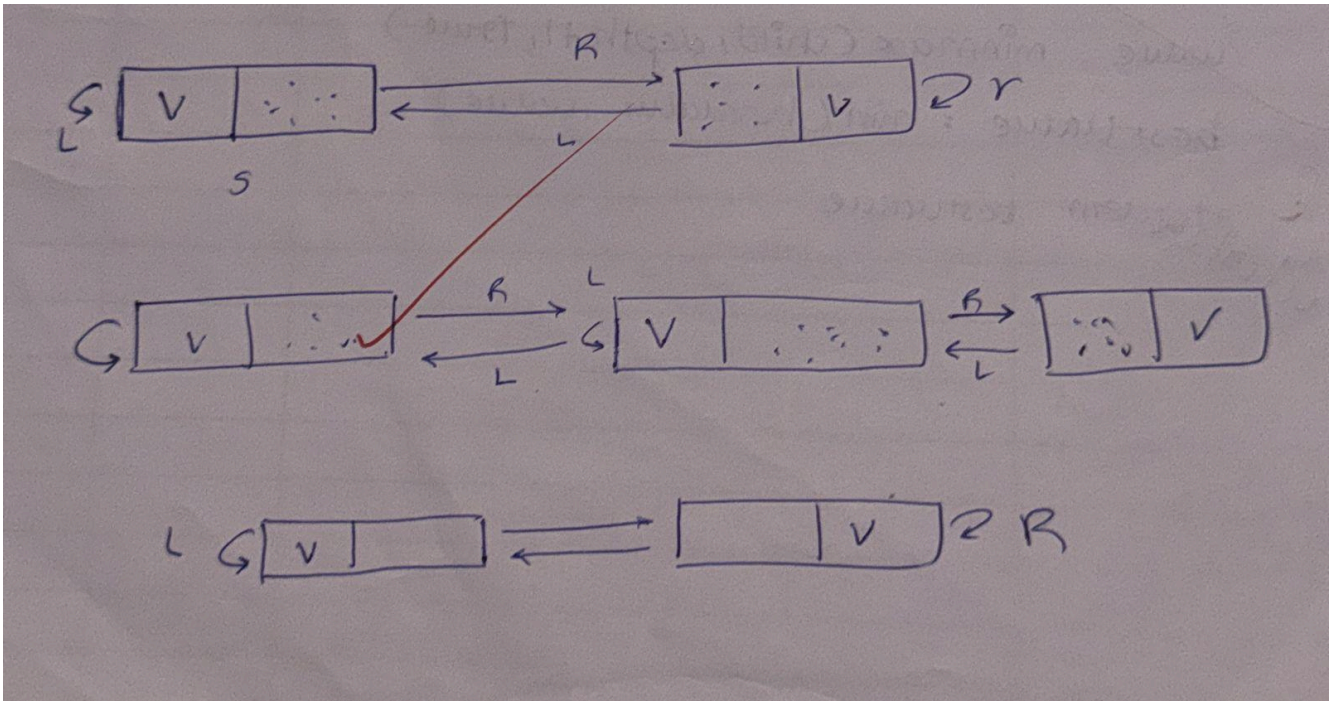
```
        cost += 1
        print("COST for moving RIGHT: " + str(cost))
        goal_state['B'] = '0'
        cost += 1
        print("COST for SUCK: " + str(cost))
        print("Location B has been Cleaned.")
    else:
        print("No action needed; Location B is already clean.")

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

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

```
            cost += 1
            print("COST for SUCK: " + str(cost))
            print("Location A has been Cleaned.")
        else:
            print("No action needed; Location A is already clean.")

    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))
    print("Tanush Prajwal S")
    print("1BM22CS304")

vacuum_world()
```

Output Snapshot

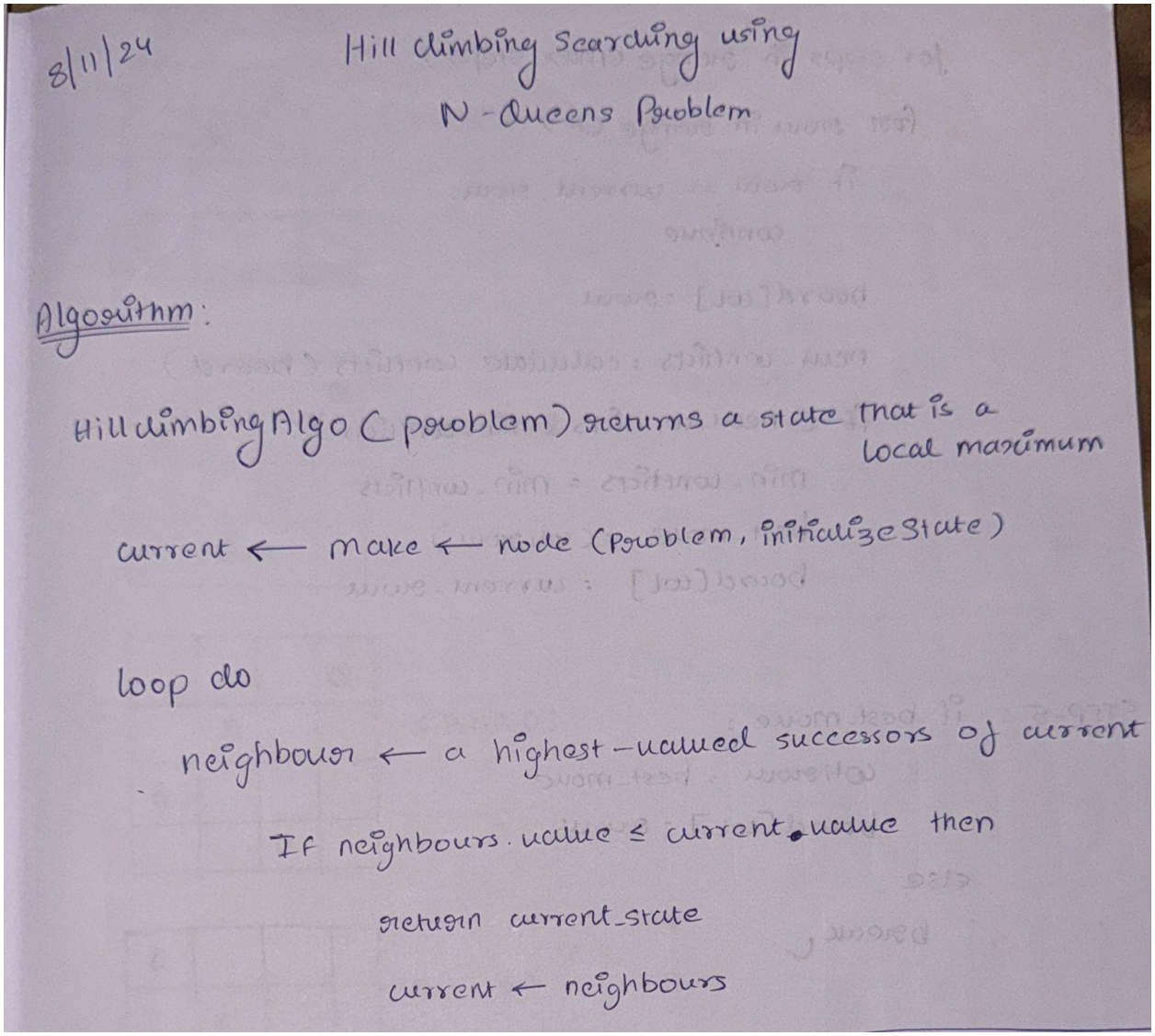
```
Output
Enter Location of Vacuum (A or B): A
Enter status of A (0 for Clean, 1 for Dirty): 1
Enter status of other room (0 for Clean, 1 for Dirty): 1
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
COST for CLEANING A: 1
Location A has been Cleaned.
Location B is Dirty.
Moving RIGHT to Location B.
COST for moving RIGHT: 2
COST for SUCK: 3
Location B has been Cleaned.

GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
Tanush Prajwal S
1BM22CS304

=== Code Execution Successful ===
```

State Space Tree


R
V
V
r
L
S
R
L
V
V
V
L
R
L
L
V
V
R

Program-05 Hill Climbing

Algorithm



8/11/24

Hill climbing Searching using
N-Queens Problem

Algorithm:

HillclimbingAlgo (problem) returns a state that is a local maximum

current ← make ← node (Problem, initialize State)

loop do

neighbour ← a highest-valued successors of current

If neighbours.value ≤ current.value then

return current_state

current ← neighbours

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()
print("Tanush Prajwal S")
print("1BM22CS304")
```

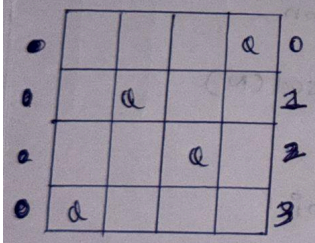
OUTPUT

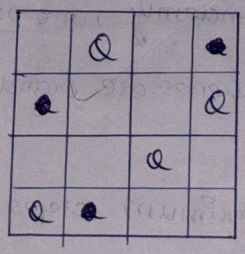
```
Step 5:
 .  .  Q  .
 Q  .  .  .
 .  .  .  .
 .  Q  .  Q

Conflicts: 1

Step 6:
 .  .  Q  .
 Q  .  .  .
 .  .  .  Q
 .  Q  .  .

Conflicts: 0

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

A solution was found with no conflicts!
Tanush Prajwal S
1BM22CS304

=== Code Execution Successful ===
```

State Space Tree

State space diagram

Initial state



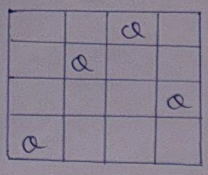


(1,2)(2,3)

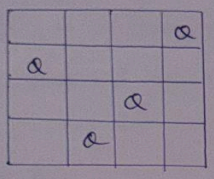
h(x) = 1

[redacted]

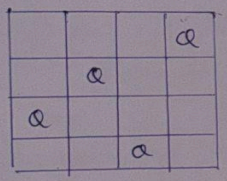
[1,3,2,0]



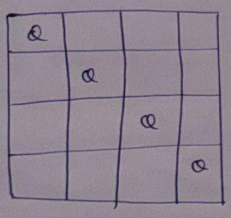
[2,1,3,0]



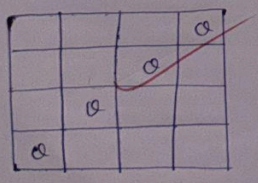
[3,0,2,1]



[3,1,2,0]

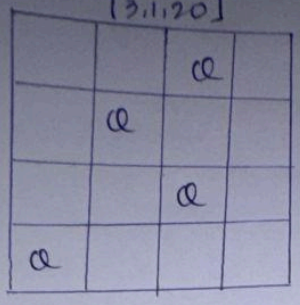


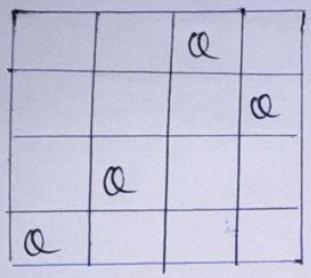
[0,1,2,3]



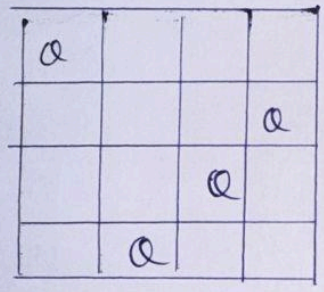
[3,2,1,0]

[2,1,3,0]

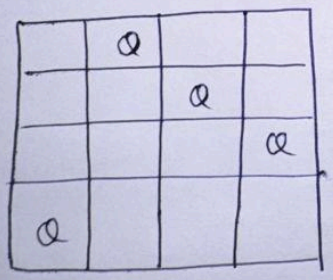




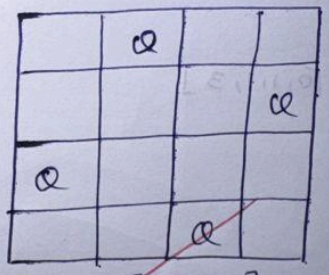
[2,3,1,0]



[0,3,2,1]



[1,2,3,0]



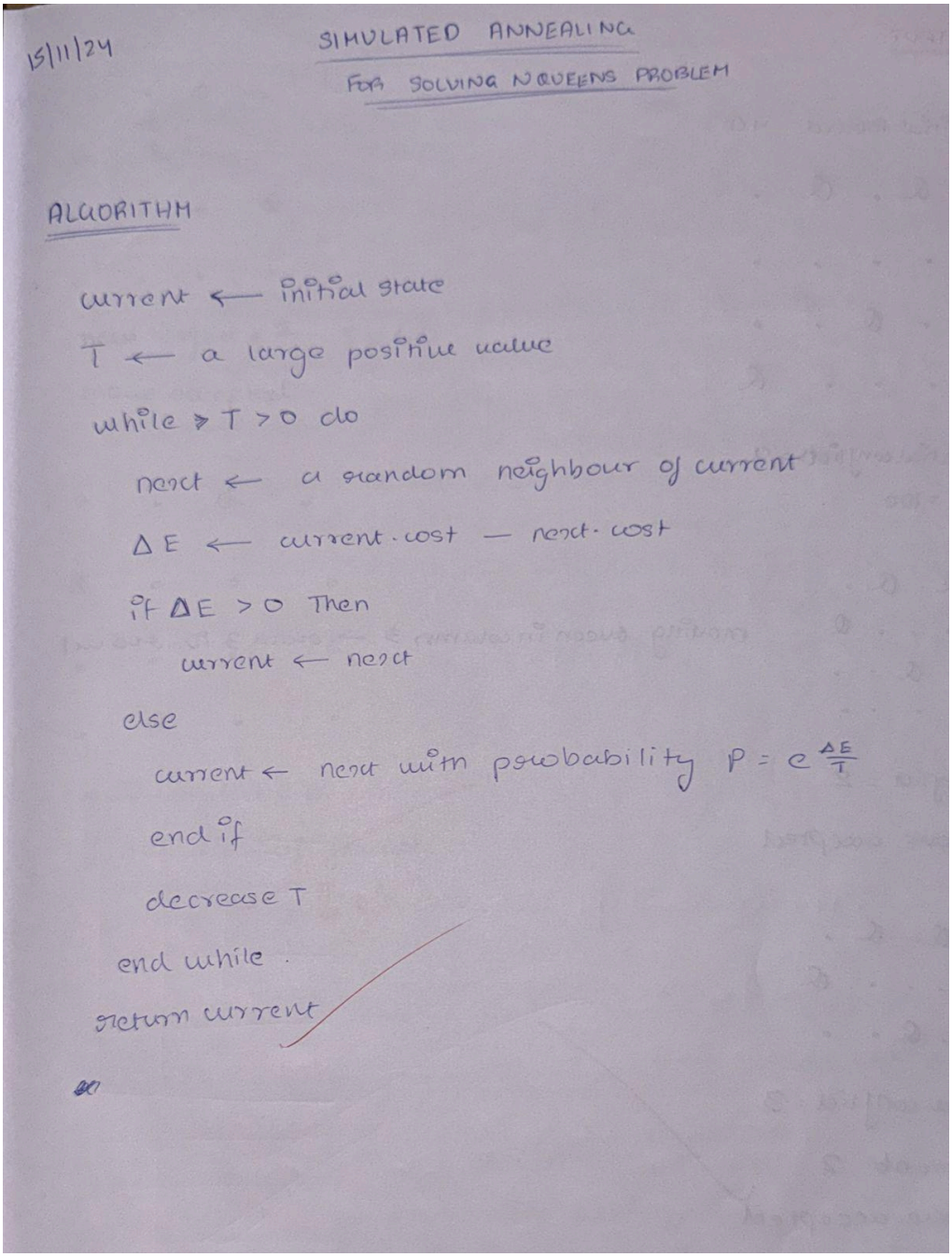
[1,3,0,2]

Solution

15/11/2021

Program-06 Simulated Annealing

Algorithm



15/11/24

SIMULATED ANNEALING
FOR SOLVING N QUEENS PROBLEM

ALGORITHM

current ← initial state
T ← a large positive value
while T > 0 do
  next ← a random neighbour of current
  ΔE ← current.cost − next.cost
  if ΔE > 0 Then
    current ← next
  else
    current ← next with probability $P = e^{\frac{\Delta E}{T}}$
  end if
  decrease T
end while
return current

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 with detailed steps."""
    # 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)

        # Display the current step, board, and conflicts
        print(f"Iteration {iteration + 1}:")
        print(f"Temperature: {temperature:.2f}")
        print(f"Trying to move queen in column {col} from row {original_row} to row {new_row}")
        print_board(board, n)
        print(f"New conflicts: {new_conflicts}, Current conflicts: {current_conflicts}")

        # 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
            print("Move accepted.\n")
        else:
            # If no improvement, revert the move
            board[col] = original_row
            print("Move rejected. Reverting to previous state.\n")

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

        iteration += 1

    return board, current_conflicts
```

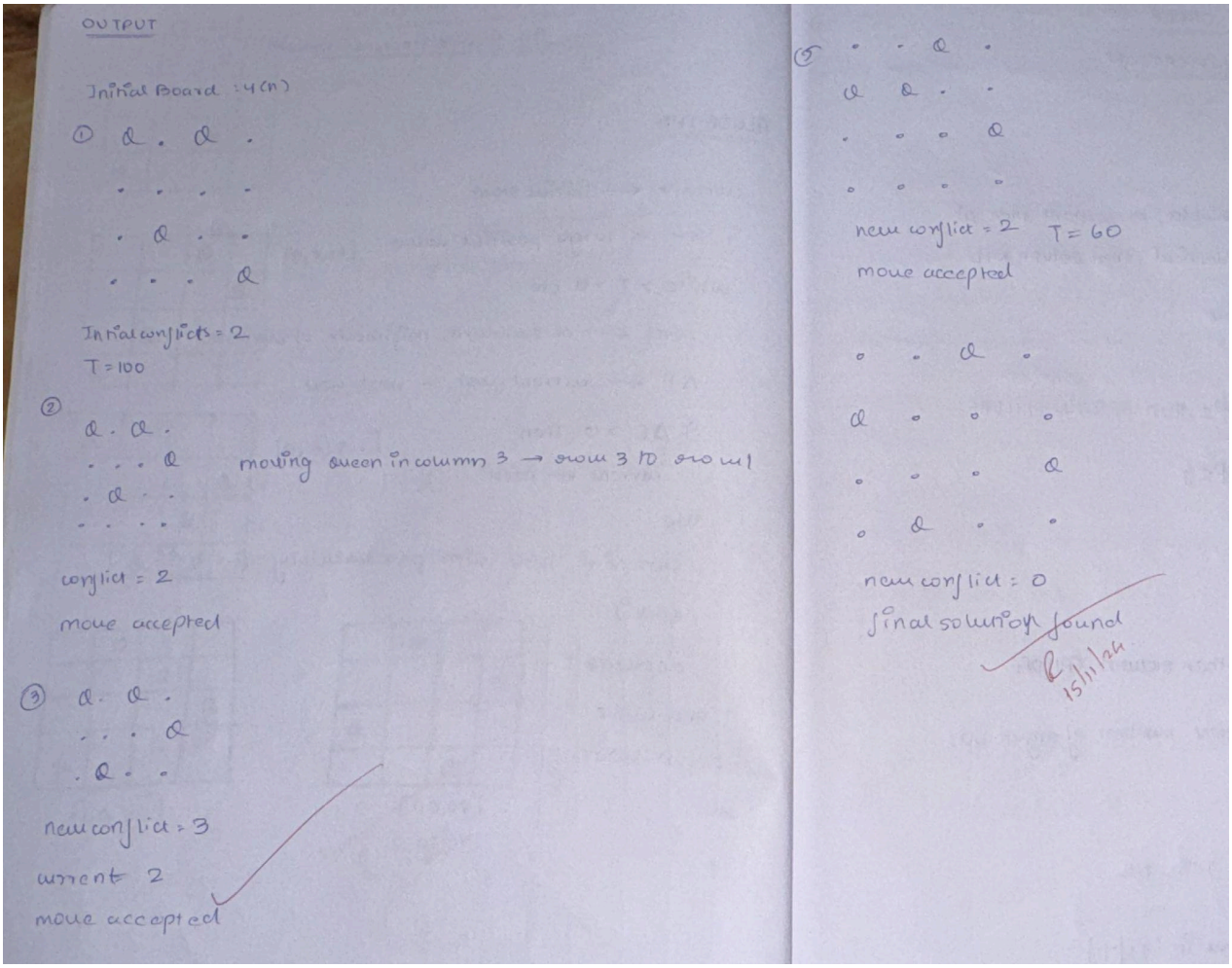
```
def main():
    # Input dynamic parameters
    print("Welcome to the N-Queens Problem Solver using Simulated Annealing!")
    n = int(input("Enter the size of the board (N): "))
    initial_temp = float(input("Enter the initial temperature (e.g., 1000): "))
    cooling_rate = float(input("Enter the cooling rate (e.g., 0.995): "))
    max_iterations = int(input("Enter the maximum number of iterations (e.g., 10000): "))
    print("Tanush Prajwal S ")
    print("1BM22CS304")

    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()
```

Output Snapshot

```
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): 0.99
Enter the maximum number of iterations (e.g., 10000): 10000
Tanush Prajwal S
1BM22CS304
Initial board:
 .  .  .  .
 .  .  .  Q
 Q  Q  .  .
 .  .  Q  .

Initial conflicts: 2

Iteration 1:
Temperature: 1000.00
Trying to move queen in column 2 from row 3 to row 2
 .  .  .  .
 .  .  .  Q
 Q  Q  Q  .
 .  .  .  .

New conflicts: 4, Current conflicts: 2
Move accepted.

Iteration 2:
Temperature: 990.00
Trying to move queen in column 0 from row 2 to row 3
 .  .  .  .
 .  .  .  Q
 .  Q  Q  .
```

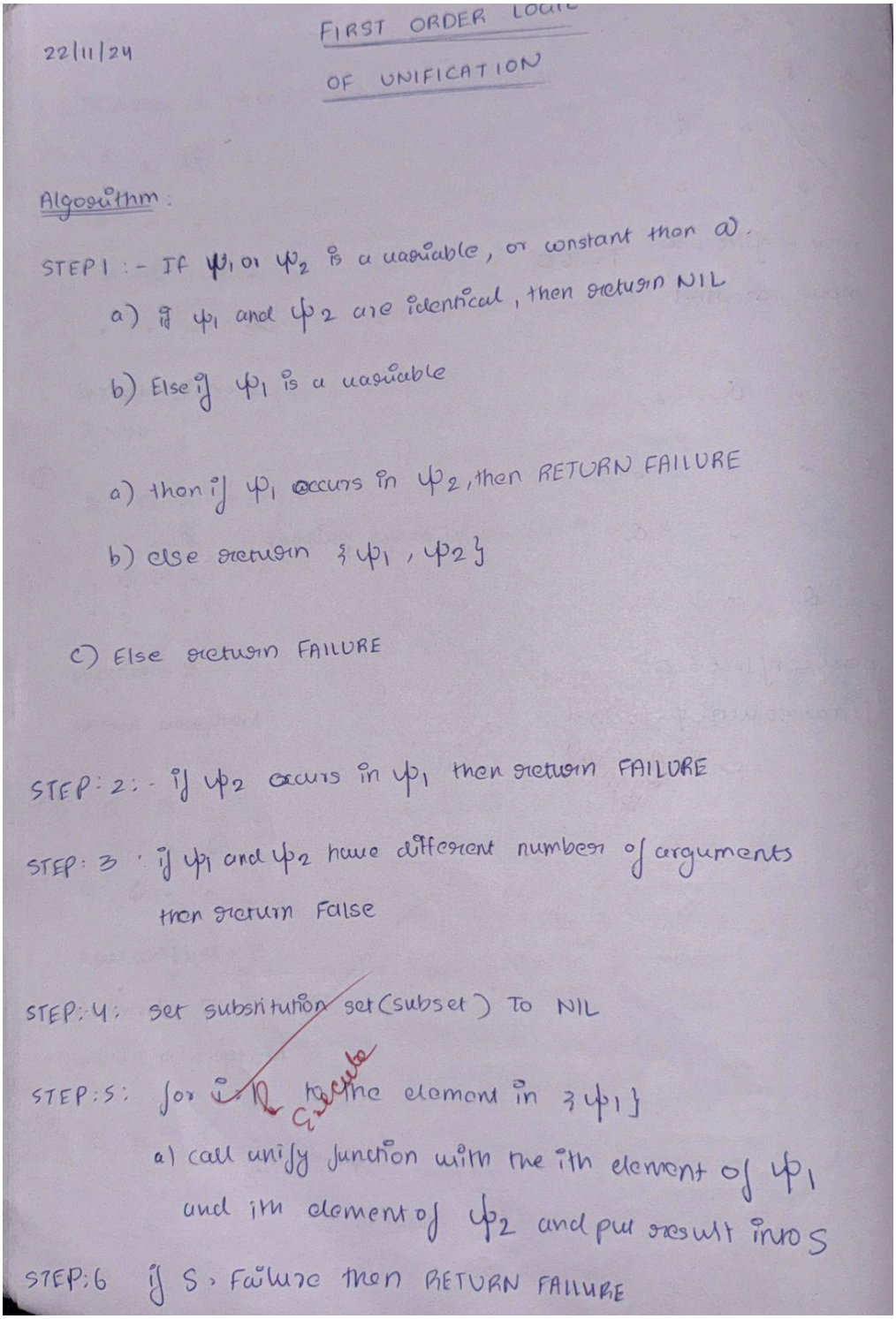
```
Iteration 10:
Temperature: 913.52
Trying to move queen in column 0 from row 3 to row 1
 .  .  Q  .
 Q  .  .  .
 .  .  .  .
 .  Q  .  Q

New conflicts: 1, Current conflicts: 3
Move accepted.

Iteration 11:
Temperature: 904.38
Trying to move queen in column 3 from row 3 to row 2
 .  .  Q  .
 Q  .  .  .
 .  .  .  Q
 .  Q  .  .

New conflicts: 0, Current conflicts: 1
Move accepted.

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

A solution was found with no conflicts!
```

State Space Tree



Program-07 Unification in first order logic

Algorithm



22/11/24

FIRST ORDER LOGIC

OF UNIFICATION

Algorithm:

STEP 1 :- If ψ_1 or ψ_2 is a variable, or constant then a).

a) if ψ_1 and 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 $\{\psi_1, \psi_2\}$

c) Else return FAILURE

STEP: 2:- if ψ_2 occurs in ψ_1 then return FAILURE

STEP: 3 : if ψ_1 and ψ_2 have different number of arguments then return False

STEP: 4: Set substitution set (subset) To NIL

STEP: 5: for i = 1 to the element in $\{\psi_1\}$

a) call unify function with the ith element of ψ_1 and ith element of ψ_2 and put result into S

STEP: 6 if S = Failure then RETURN FAILURE

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))
```

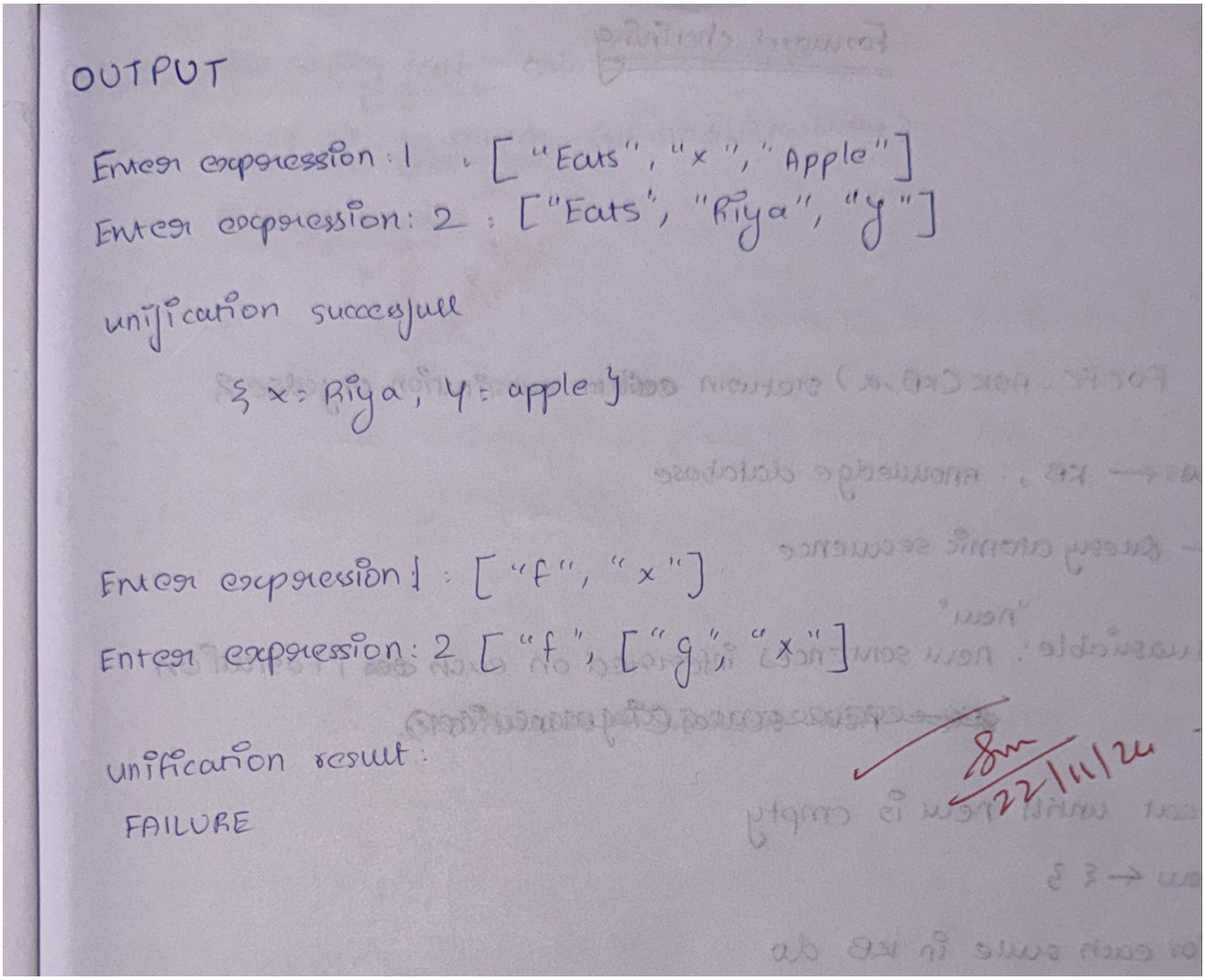
```
print("Tanush Prajwal S")
print("1BM22CS304 \n")

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

Output Snapshot

```
=== Unification Algorithm ===

Tanush Prajwal S
1BM22CS304
Please enter the expressions in list format.
Example: ["Eats", "x", "Apple"]

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

Unification Successful:
{ x = Riya, y = Apple }
```

```
=== Unification Algorithm ===

Tanush Prajwal S
1BM22CS304
Please enter the expressions in list format.
Example: ["Eats", "x", "Apple"]

Enter Expression 1: ["f", "x"]
Enter Expression 2: ["f", ["g", "x"]]

Unification Result:
FAILURE
```

State Space Tree



OUTPUT

Enter expression 1 : ["Eats", "x", "Apple"]
Enter expression: 2 : ["Eats", "Riya", "y"]

unification successfull

{ x: Riya, y: apple }

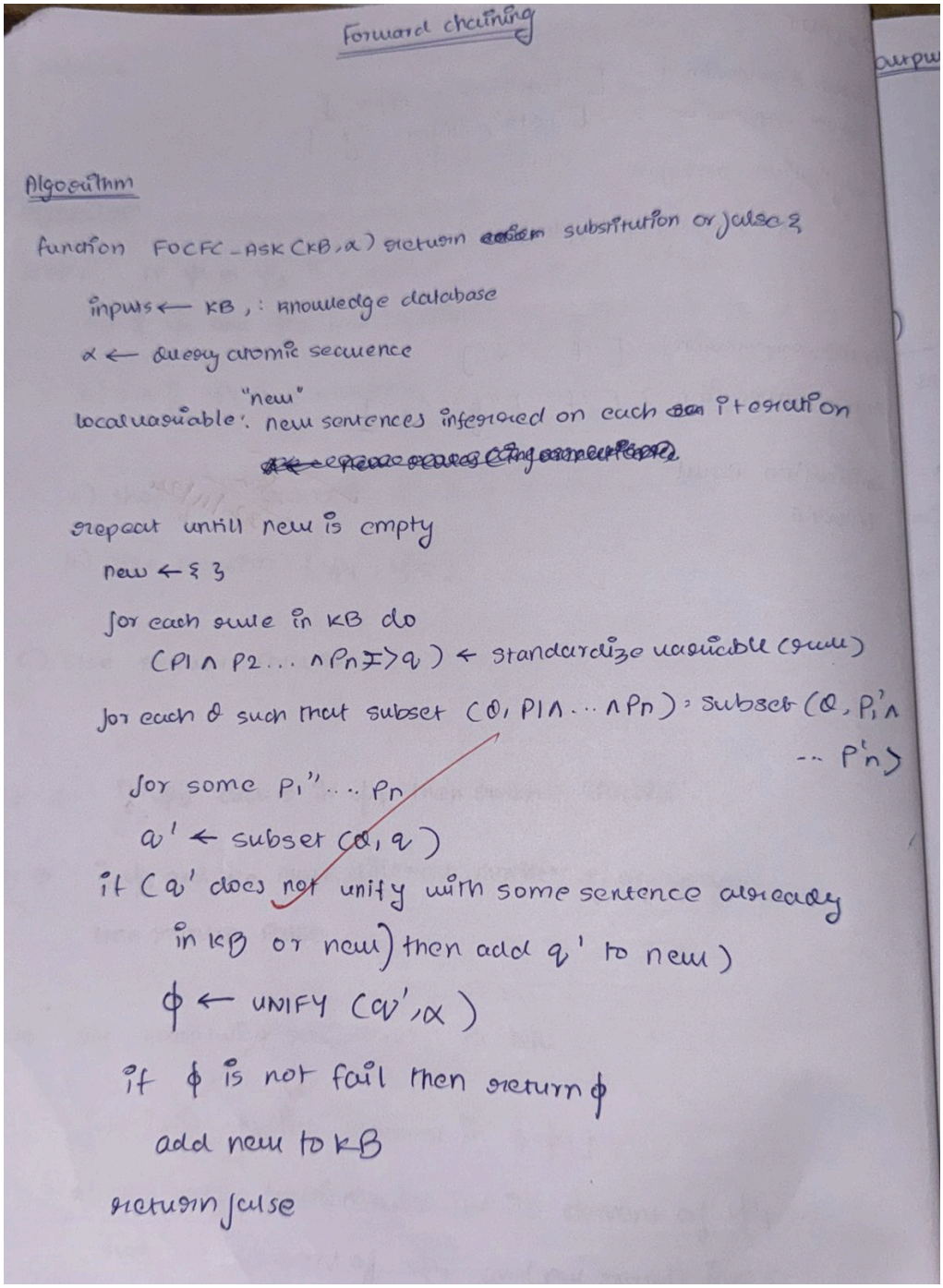
Enter expression 1 : ["f", "x"]
Enter expression: 2 ["f", ["g", "x"]]

unification result:
FAILURE

22/11/24

Program-8 Forward Reasoning

Algorithm



Forward chaining

Algorithm

function FOCFC-ASK(KB, α) return substitution or false

inputs ← KB, : knowledge database

α ← query atomic sequence

local variable: "new" new sentences inferred on each iteration

repeat until new is empty

new ← { }

for each rule in KB do

(P1 ∧ P2 ... ∧ Pn ⇒ q) ← standardize variable (rule)

for each θ such that subset (θ, P1 ∧ ... ∧ Pn) = subset (θ, P1' ∧ ... P'n)

for some P1'' ... Pn

q' ← subset (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

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
```

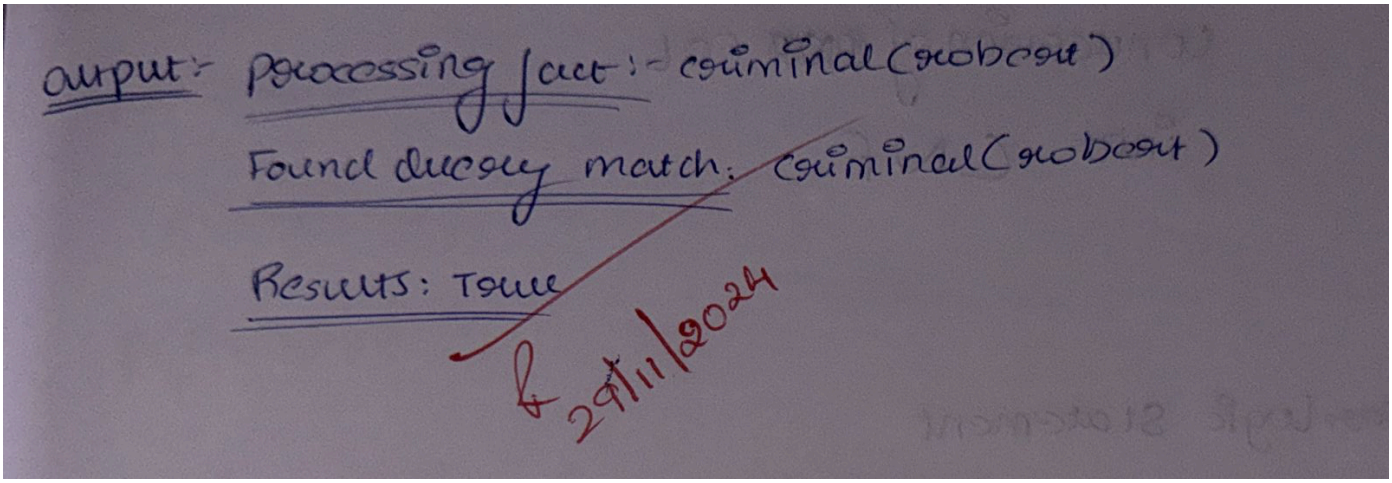
```
                new_fact = {'conclusion': rule['conclusion']}
                agenda.append(new_fact)  # Add new fact to agenda
                print(f"New fact inferred: {new_fact['conclusion']}")

    # 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("Final Result:", result)
print("Tanush Prajwal S")
print("1BM22CS304")
```

OutputSnapshot

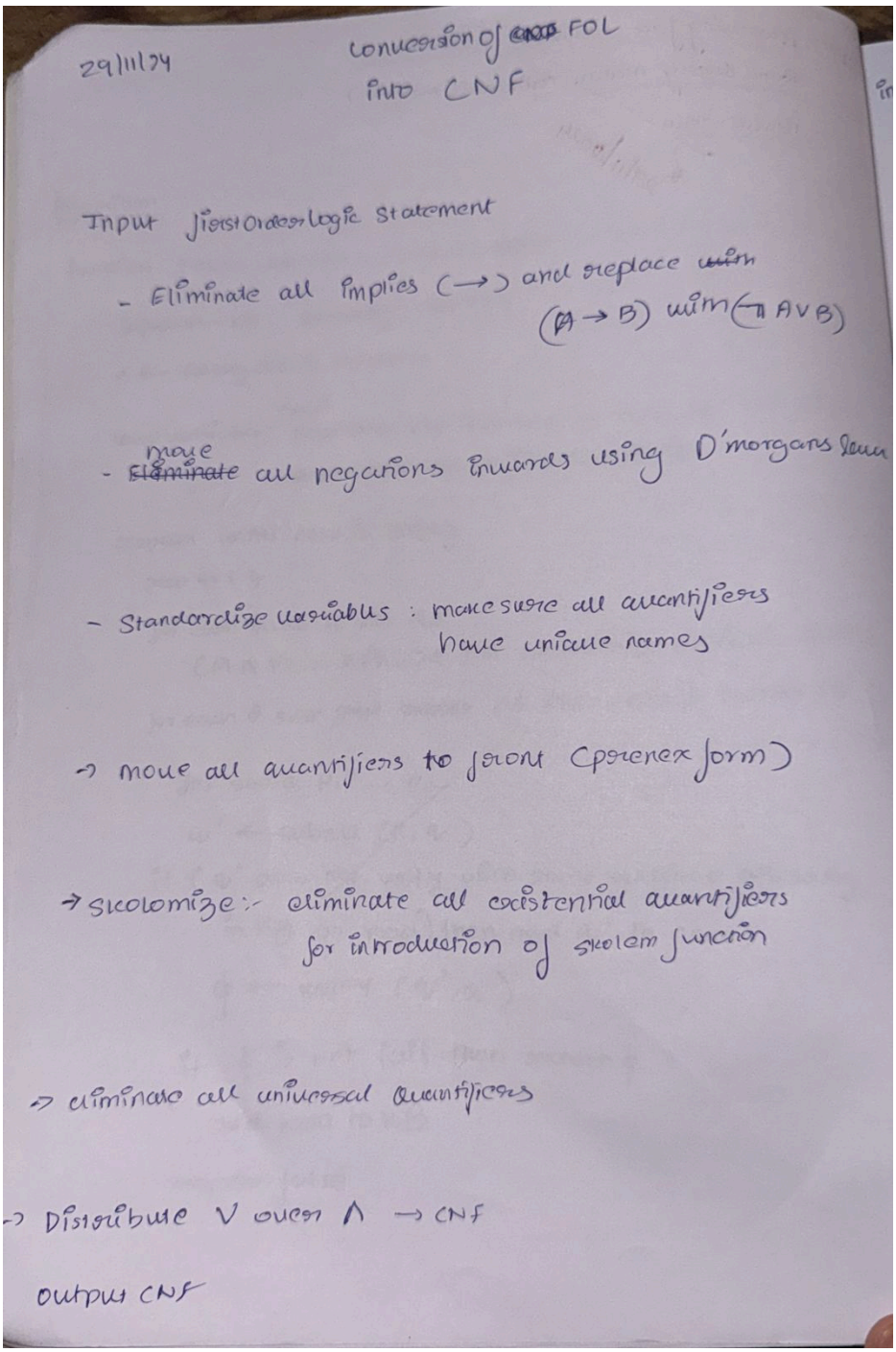
```
Initial agenda: ['American(Robert)', 'Missile(T1)', 'Owns(A, T1)', 'Enemy(A, America)', 'Sells(Robert, T1, A)']
Initial inferred: set()

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: {'American(Robert)', 'Owns(A, T1)', '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: {'American(Robert)', 'Owns(A, T1)', 'Missile(T1)'}

Processing fact: Enemy(A, America)
Inferred facts: {'American(Robert)', 'Owns(A, T1)', 'Enemy(A, America)', 'Missile(T1)'}
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: {'American(Robert)', 'Owns(A, T1)', 'Enemy(A, America)', 'Missile(T1)'}

Processing fact: Sells(Robert, T1, A)
Inferred facts: {'American(Robert)', 'Missile(T1)', 'Enemy(A, America)', 'Sells(Robert, T1, A)', 'Owns(A, T1)'}
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)', 'Enemy(A, America)', 'Sells(Robert, T1, A)', 'Owns(A, T1)'}

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

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

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

Final Result: True
Tanush Prajwal S
1BM22CS304
```

State Space Tree

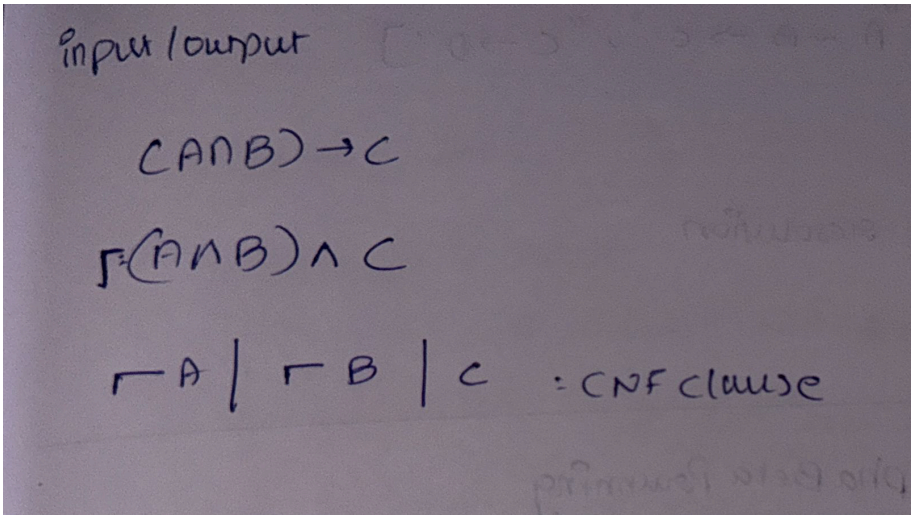


Program - 9: Conversion of FOL into CNF

Algorithm



Output:



Code

```
import copy
import re
from typing import Union, List

# ----------------------------------------
# Data structures to represent FOL
# ----------------------------------------

class Var:
    """Represents a variable."""
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name

    def __repr__(self):
        return f"Var({self.name})"


class Func:
    """Represents a function or constant symbol with arguments."""
    def __init__(self, name, args=None):
```

```
        self.name = name
        self.args = args if args else []

    def __str__(self):
        if self.args:
            return f"{self.name}({', '.join(str(a) for a in self.args)})"
        return self.name

    def __repr__(self):
        return f"Func({self.name}, {self.args})"


class Pred:
    """Represents a predicate with arguments."""
    def __init__(self, name, args=None):
        self.name = name
        self.args = args if args else []

    def __str__(self):
        return f"{self.name}({', '.join(str(a) for a in self.args)})"

    def __repr__(self):
        return f"Pred({self.name}, {self.args})"


class Not:
    """Represents negation of a formula."""
    def __init__(self, formula):
        self.formula = formula

    def __str__(self):
        return f"¬({self.formula})"

    def __repr__(self):
        return f"Not({self.formula})"


class And:
    """Represents conjunction of two formulas."""
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def __str__(self):
        return f"({self.left} ∧ {self.right})"

    def __repr__(self):
        return f"And({self.left}, {self.right})"
```

```
class Or:
    """Represents disjunction of two formulas."""
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def __str__(self):
        return f"({self.left} ∨ {self.right})"

    def __repr__(self):
        return f"Or({self.left}, {self.right})"


class Impl:
    """Represents an implication (left -> right)."""
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def __str__(self):
        return f"({self.left} → {self.right})"

    def __repr__(self):
        return f"Impl({self.left}, {self.right})"


class Equiv:
    """Represents an equivalence (left <-> right)."""
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def __str__(self):
        return f"({self.left} ↔ {self.right})"

    def __repr__(self):
        return f"Equiv({self.left}, {self.right})"


class ForAll:
    """Represents universal quantification ∀x. formula."""
    def __init__(self, var, formula):
        self.var = var
        self.formula = formula

    def __str__(self):
        return f"∀{self.var}.({self.formula})"

    def __repr__(self):
        return f"ForAll({self.var}, {self.formula})"
```

```
class Exists:
    """Represents existential quantification ∃x. formula."""
    def __init__(self, var, formula):
        self.var = var
        self.formula = formula

    def __str__(self):
        return f"∃{self.var}.({self.formula})"

    def __repr__(self):
        return f"Exists({self.var}, {self.formula})"


# ----------------------------------------
# 1. Example: Predefined Input
# ----------------------------------------
#
# We'll use an example formula:
#   ∀x ( P(x) → ∃y ( Q(x,y) ∧ R(x,y) ) )
#
# This formula says:
#   "For every x, if P(x) holds, then there exists some y such that Q(x,y) and R(x,y)."

def predefined_formula():
    """
    Builds and returns an internal representation of:
      ∀x [ P(x) → ∃y ( Q(x,y) ∧ R(x,y) ) ]
    """
    x = Var("x")
    y = Var("y")
    p_x = Pred("P", [x])
    q_xy = Pred("Q", [x, y])
    r_xy = Pred("R", [x, y])
    conj = And(q_xy, r_xy)           # Q(x,y) ∧ R(x,y)
    exists_y = Exists(y, conj)       # ∃y (Q(x,y) ∧ R(x,y))
    implication = Impl(p_x, exists_y) # P(x) → ∃y (...)
    return ForAll(x, implication)    # ∀x [ P(x) → ... ]


# ----------------------------------------
# 2. Eliminate implications and equivalences
#    - (A → B)  becomes  (¬A ∨ B)
#    - (A ↔ B)  becomes  (A → B) ∧ (B → A) -> then we expand as above
# ----------------------------------------

def eliminate_imp_equiv(formula):
    if isinstance(formula, Impl):
        # (A → B) => (¬A ∨ B)
```

```
        return Or(Not(eliminate_imp_equiv(formula.left)),
                  eliminate_imp_equiv(formula.right))
    elif isinstance(formula, Equiv):
        # (A ↔ B) => (A → B) ∧ (B → A)
        left_imp = Impl(formula.left, formula.right)
        right_imp = Impl(formula.right, formula.left)
        return And(eliminate_imp_equiv(left_imp), eliminate_imp_equiv(right_imp))
    elif isinstance(formula, And):
        return And(eliminate_imp_equiv(formula.left), eliminate_imp_equiv(formula.right))
    elif isinstance(formula, Or):
        return Or(eliminate_imp_equiv(formula.left), eliminate_imp_equiv(formula.right))
    elif isinstance(formula, Not):
        return Not(eliminate_imp_equiv(formula.formula))
    elif isinstance(formula, ForAll):
        return ForAll(formula.var, eliminate_imp_equiv(formula.formula))
    elif isinstance(formula, Exists):
        return Exists(formula.var, eliminate_imp_equiv(formula.formula))
    else:
        # Pred, Func, Var are terminal
        return formula


# ----------------------------------------
# 3. Move negations inwards (using De Morgan's, etc.)
#    - ¬(A ∧ B) = (¬A ∨ ¬B)
#    - ¬(A ∨ B) = (¬A ∧ ¬B)
#    - ¬∀x. φ = ∃x. ¬φ
#    - ¬∃x. φ = ∀x. ¬φ
#    - Double negation elimination
# ----------------------------------------

def move_negation_inwards(formula):
    if isinstance(formula, Not):
        inner = formula.formula
        # Handle double negation
        if isinstance(inner, Not):
            return move_negation_inwards(inner.formula)

        # De Morgan's: ¬(A ∧ B) => (¬A ∨ ¬B)
        if isinstance(inner, And):
            return Or(move_negation_inwards(Not(inner.left)),
                      move_negation_inwards(Not(inner.right)))
        # De Morgan's: ¬(A ∨ B) => (¬A ∧ ¬B)
        if isinstance(inner, Or):
            return And(move_negation_inwards(Not(inner.left)),
                       move_negation_inwards(Not(inner.right)))
        # ¬∀x. φ => ∃x. ¬φ
        if isinstance(inner, ForAll):
            return Exists(inner.var, move_negation_inwards(Not(inner.formula)))
        # ¬∃x. φ => ∀x. ¬φ
```

```
        if isinstance(inner, Exists):
            return ForAll(inner.var, move_negation_inwards(Not(inner.formula)))

        # If it's a predicate, just keep Not(...) as is
        return Not(move_negation_inwards(inner))

    elif isinstance(formula, And):
        return And(move_negation_inwards(formula.left),
                   move_negation_inwards(formula.right))
    elif isinstance(formula, Or):
        return Or(move_negation_inwards(formula.left),
                  move_negation_inwards(formula.right))
    elif isinstance(formula, ForAll):
        return ForAll(formula.var, move_negation_inwards(formula.formula))
    elif isinstance(formula, Exists):
        return Exists(formula.var, move_negation_inwards(formula.formula))
    else:
        # Predicate, variable, function - no negation to move
        return formula


# ----------------------------------------
# 4. Standardize variables
#    - Ensures each quantified variable is unique
# ----------------------------------------

var_id_counter = 0

def new_var_name():
    global var_id_counter
    var_id_counter += 1
    return f"x_{var_id_counter}"

def standardize_vars(formula, bound_vars=None):
    """
    Returns a version of formula where each quantifier uses a fresh variable name.
    bound_vars is a mapping from old variable -> new variable
    """
    if bound_vars is None:
        bound_vars = {}

    if isinstance(formula, ForAll):
        new_name = new_var_name()
        new_variable = Var(new_name)
        # Extend the bound_vars mapping
        extended_bound_vars = copy.copy(bound_vars)
        extended_bound_vars[formula.var.name] = new_variable
        new_subformula = standardize_vars(formula.formula, extended_bound_vars)
        return ForAll(new_variable, new_subformula)
```

```
    elif isinstance(formula, Exists):
        new_name = new_var_name()
        new_variable = Var(new_name)
        extended_bound_vars = copy.copy(bound_vars)
        extended_bound_vars[formula.var.name] = new_variable
        new_subformula = standardize_vars(formula.formula, extended_bound_vars)
        return Exists(new_variable, new_subformula)

    elif isinstance(formula, And):
        return And(standardize_vars(formula.left, bound_vars),
                   standardize_vars(formula.right, bound_vars))

    elif isinstance(formula, Or):
        return Or(standardize_vars(formula.left, bound_vars),
                  standardize_vars(formula.right, bound_vars))

    elif isinstance(formula, Not):
        return Not(standardize_vars(formula.formula, bound_vars))

    elif isinstance(formula, Pred):
        # Replace arguments if they are variables bound by something above
        new_args = []
        for arg in formula.args:
            if isinstance(arg, Var) and arg.name in bound_vars:
                new_args.append(bound_vars[arg.name])
            else:
                new_args.append(arg)
        return Pred(formula.name, new_args)

    elif isinstance(formula, Func):
        new_args = []
        for arg in formula.args:
            if isinstance(arg, Var) and arg.name in bound_vars:
                new_args.append(bound_vars[arg.name])
            else:
                new_args.append(arg)
        return Func(formula.name, new_args)

    else:
        return formula


# ---------------------------------------
# 5. Skolemization
#    - Eliminate ∃ by introducing Skolem functions/constants
#    - For each ∃x, we create a Skolem function f(...) whose arguments are
#      all universally quantified variables in scope.
# ---------------------------------------

skolem_counter = 0
```

```
def new_skolem_name():
    global skolem_counter
    skolem_counter += 1
    return f"Sk{skolem_counter}"

def skolemize(formula, universally_bound=None):
    """
    Skolemize the formula by eliminating existential quantifiers.
    'universally_bound' is the list of universal variables in scope.
    """
    if universally_bound is None:
        universally_bound = []

    if isinstance(formula, ForAll):
        # push the universal variable into scope
        new_universally_bound = universally_bound + [formula.var]
        return ForAll(formula.var, skolemize(formula.formula, new_universally_bound))

    elif isinstance(formula, Exists):
        # Introduce a Skolem function/constant
        sk_name = new_skolem_name()
        if len(universally_bound) == 0:
            # no universal variables in scope => Skolem constant
            skolem_func = Func(sk_name, [])
        else:
            # Skolem function with arguments = all universal vars
            skolem_func = Func(sk_name, universally_bound)

        # Replace all occurrences of formula.var with skolem_func
        replaced = replace_var(formula.formula, formula.var, skolem_func)
        # Drop the existential quantifier
        return skolemize(replaced, universally_bound)

    elif isinstance(formula, And):
        return And(skolemize(formula.left, universally_bound),
                   skolemize(formula.right, universally_bound))
    elif isinstance(formula, Or):
        return Or(skolemize(formula.left, universally_bound),
                  skolemize(formula.right, universally_bound))
    elif isinstance(formula, Not):
        return Not(skolemize(formula.formula, universally_bound))
    else:
        # Pred / Func / Var
        return formula

def replace_var(formula, var_to_replace, new_term):
    """ Replace all free occurrences of var_to_replace in formula with new_term. """
    if isinstance(formula, Pred):
        new_args = []
```

```
        for arg in formula.args:
            if isinstance(arg, Var) and arg.name == var_to_replace.name:
                new_args.append(new_term)
            else:
                new_args.append(replace_var(arg, var_to_replace, new_term))
        return Pred(formula.name, new_args)

    elif isinstance(formula, Func):
        new_args = []
        for arg in formula.args:
            if isinstance(arg, Var) and arg.name == var_to_replace.name:
                new_args.append(new_term)
            else:
                new_args.append(replace_var(arg, var_to_replace, new_term))
        return Func(formula.name, new_args)

    elif isinstance(formula, And):
        return And(replace_var(formula.left, var_to_replace, new_term),
                   replace_var(formula.right, var_to_replace, new_term))
    elif isinstance(formula, Or):
        return Or(replace_var(formula.left, var_to_replace, new_term),
                  replace_var(formula.right, var_to_replace, new_term))
    elif isinstance(formula, Not):
        return Not(replace_var(formula.formula, var_to_replace, new_term))
    elif isinstance(formula, ForAll) or isinstance(formula, Exists):
        # If the bound variable is the same as var_to_replace, skip
        if formula.var.name == var_to_replace.name:
            # var_to_replace is shadowed, do not replace inside
            return formula
        else:
            new_subf = replace_var(formula.formula, var_to_replace, new_term)
            if isinstance(formula, ForAll):
                return ForAll(formula.var, new_subf)
            else:
                return Exists(formula.var, new_subf)
    else:
        # Var or anything else
        return formula


# ---------------------------------------
# 6. Drop universal quantifiers
#    After Skolemization, all variables are universally-quantified.
#    Typically we just drop them, assuming they are universally bound.
# ---------------------------------------

def drop_universal_quantifiers(formula):
    if isinstance(formula, ForAll):
        return drop_universal_quantifiers(formula.formula)
    elif isinstance(formula, And):
```

```
        return And(drop_universal_quantifiers(formula.left),
                   drop_universal_quantifiers(formula.right))
    elif isinstance(formula, Or):
        return Or(drop_universal_quantifiers(formula.left),
                  drop_universal_quantifiers(formula.right))
    elif isinstance(formula, Not):
        return Not(drop_universal_quantifiers(formula.formula))
    else:
        # Pred, Func, Var are terminal
        return formula


# ---------------------------------------
# 7. Distribute ∨ over ∧ to get final CNF
#    - We want a conjunction of disjunctions
#    - (A ∨ (B ∧ C)) => (A ∨ B) ∧ (A ∨ C)
# ---------------------------------------

def distribute_or_over_and(formula):
    """ Distribute disjunctions over conjunctions to produce CNF. """
    if isinstance(formula, And):
        left_cnf = distribute_or_over_and(formula.left)
        right_cnf = distribute_or_over_and(formula.right)
        return And(left_cnf, right_cnf)

    if isinstance(formula, Or):
        left = distribute_or_over_and(formula.left)
        right = distribute_or_over_and(formula.right)
        # If left or right is an And, apply distribution
        if isinstance(left, And):
            return And(distribute_or_over_and(Or(left.left, right)),
                       distribute_or_over_and(Or(left.right, right)))
        elif isinstance(right, And):
            return And(distribute_or_over_and(Or(left, right.left)),
                       distribute_or_over_and(Or(left, right.right)))
        else:
            return Or(left, right)

    if isinstance(formula, Not):
        return Not(distribute_or_over_and(formula.formula))

    # Terminal cases (Pred, Func, Var) or quantifiers (should be removed by now)
    return formula


# ---------------------------------------
# Main: Putting it all together
# ---------------------------------------

def fol_to_cnf(formula):
```

```
    """
    Convert a formula in FOL to CNF following the standard steps.
    """
    # 1. Eliminate -> and <->
    step1 = eliminate_imp_equiv(formula)

    # 2. Move negations inward
    step2 = move_negation_inwards(step1)

    # 3. Standardize variables
    step3 = standardize_vars(step2)

    # 4. Skolemize
    step4 = skolemize(step3)

    # 5. Drop universal quantifiers
    step5 = drop_universal_quantifiers(step4)

    # 6. Distribute OR over AND to get final CNF
    cnf = distribute_or_over_and(step5)
    return cnf


if __name__ == "__main__":
    # Build the predefined formula
    formula = predefined_formula()
    print("Original FOL formula:")
    print(formula)

    # Convert to CNF
    cnf_formula = fol_to_cnf(formula)

    print("\nCNF form:")
    print(cnf_formula)
    print(“Tanush Prajwal S”)
    print(“1BM22CS304”)
```

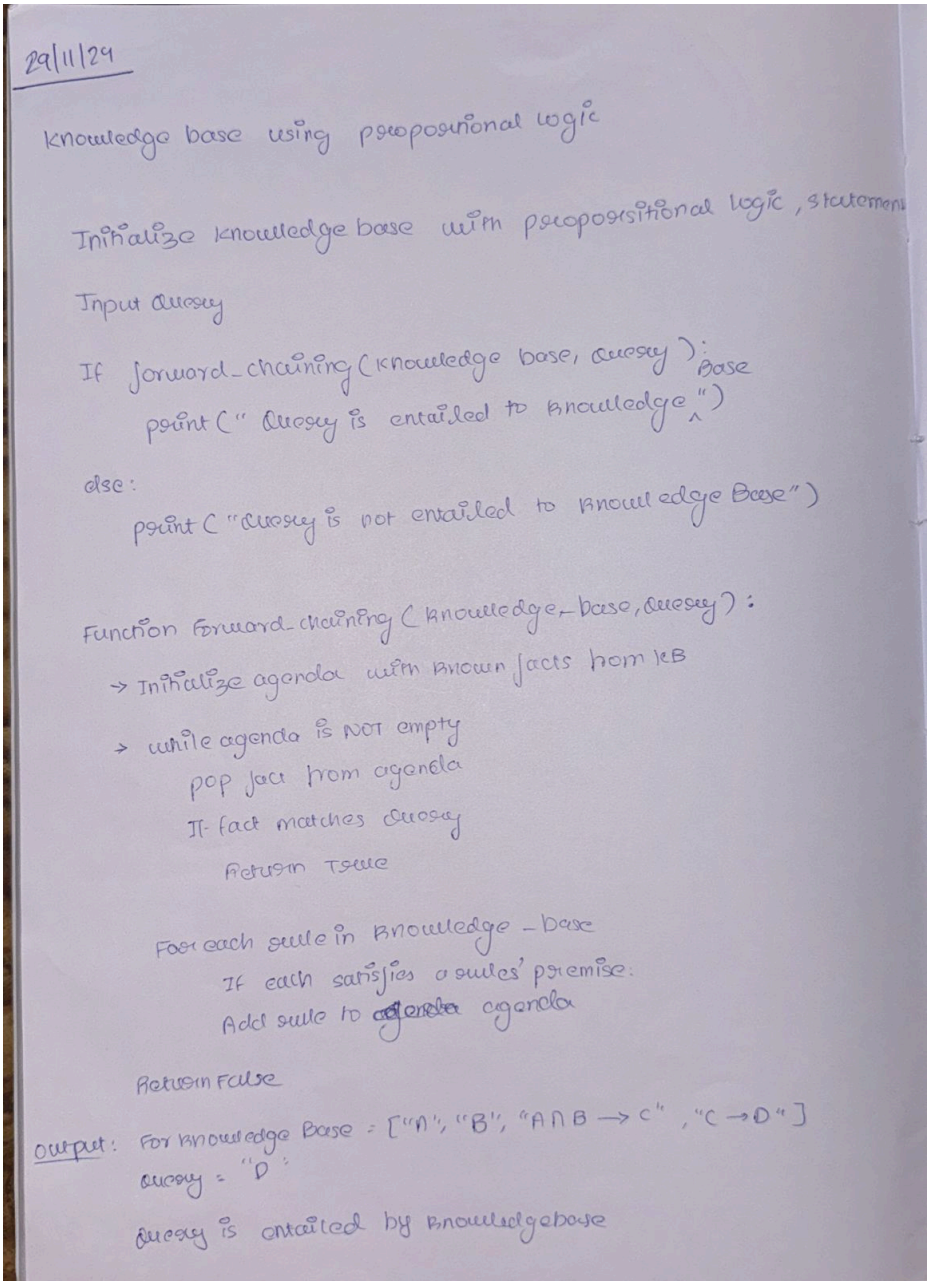
Output

Output Clear

```
Original FOL formula:
∀x.((P(x) → ∃y.((Q(x, y) ∧ R(x, y)))))

CNF form:
((¬(P(x_1)) ∨ Q(x_1, Sk1(x_1))) ∧ (¬(P(x_1)) ∨ R(x_1, Sk1(x_1))))
Tanush Prajwal S
USN : 1BM22CS304

=== Code Execution Successful ===
```

Program - 10: Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

Algorithm



29/11/24

knowledge base using propositional logic

Initialize knowledge base with propositional logic, statement

Input query

If forward_chaining(knowledge base, query): Base
    print(" query is entailed to knowledge ")

else:
    print("query is not entailed to knowledge Base")

Function forward_chaining(knowledge_base, query):
→ Initialize agenda with known facts from KB
→ while agenda is NOT empty
    pop fact from agenda
    If fact matches query
        Return True

    For each rule in knowledge_base
        If each satisfies a rules' premise:
        Add rule to ~~agenda~~ agenda

    Return False

Output: For knowledge Base = ["A"; "B"; "A ∩ B → C" , "C → D"]
query = "D"

query is entailed by knowledgebase

Code

```
combinations = [
  (True, True, True), (True, True, False), (True, False, True), (True, False, False),
  (False, True, True), (False, True, False), (False, False, True), (False, False, False)
]

variable = {'p': 0, 'q': 1, 'r': 2}
kb = ''
q = ''
priority = {'~': 3, 'v': 1, '^': 2}

def input_rules():
  global kb, q
  kb = input("Enter rule: ")  # Knowledge Base (KB)
  q = input("Enter the Query: ")  # Query

def entailment():
  global kb, q
  print('*' * 10 + "Truth Table Reference" + '*' * 10)
  print('kb', 'alpha')
  print('*' * 10)

  for comb in combinations:
    s = evaluatePostfix(toPostfix(kb), comb)  # Evaluate KB
    f = evaluatePostfix(toPostfix(q), comb)  # Evaluate Query
    print(s, f)
    print('-' * 10)

    if s and not f:  # If KB is True and query is False, KB does not entail query
      return False
  return True

def isOperand(c):
  return c.isalpha() and c != 'v'

def isLeftParanthesis(c):
```

```
  return c == '('

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

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

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

def hasLessOrEqualPriority(c1, c2):
  try:
    return priority[c1] <= priority[c2]
  except KeyError:
    return False

def toPostfix(infix):
  stack = []
  postfix = ''

  for c in infix:
    if isOperand(c):
      postfix += c
    else:
      if isLeftParanthesis(c):
        stack.append(c)
      elif isRightParanthesis(c):
        operator = stack.pop()
        while not isLeftParanthesis(operator):
          postfix += operator
          operator = stack.pop()
      else:
        while (not isEmpty(stack)) and hasLessOrEqualPriority(c, peek(stack)):
          postfix += stack.pop()
        stack.append(c)

  while not isEmpty(stack):
    postfix += stack.pop()
```

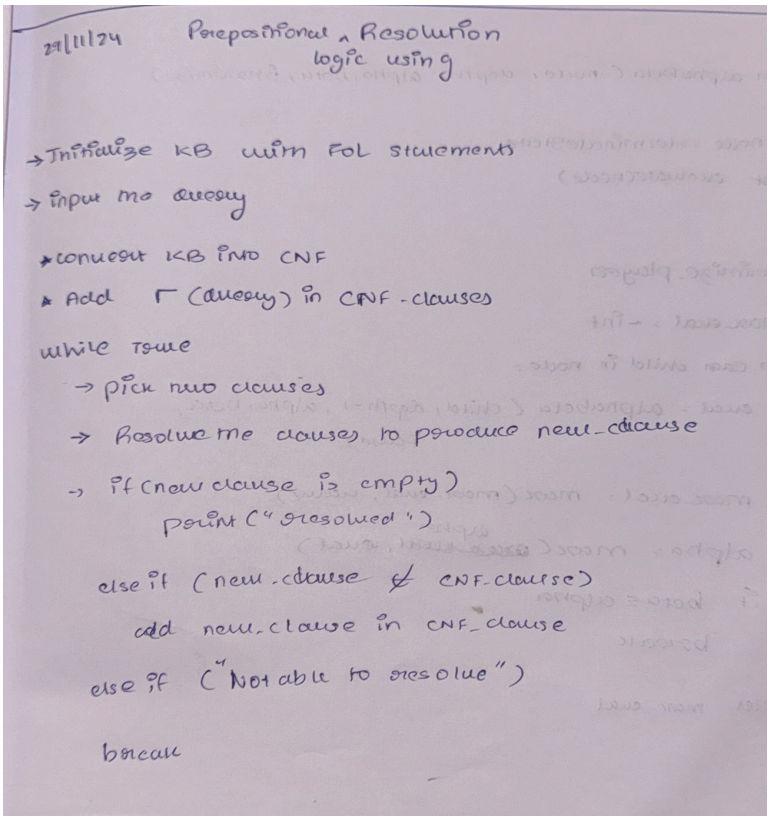
```
    return postfix

def evaluatePostfix(exp, comb):
    stack = []
    for i in exp:
        if isOperand(i):
            stack.append(comb[variable[i]])  # Push the truth value from the combination
        elif i == '~':
            val1 = stack.pop()
            stack.append(not val1)  # Negation
        else:
            val1 = stack.pop()
            val2 = stack.pop()
            stack.append(_eval(i, val2, val1))  # Evaluate AND or OR
    return stack.pop()

def _eval(i, val1, val2):
    if i == '^':
        return val2 and val1  # AND
    return val2 or val1  # OR

# Main Program
input_rules()
ans = entailment()

print("Tanush Prajwal S")
print("1BM22CS304")

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

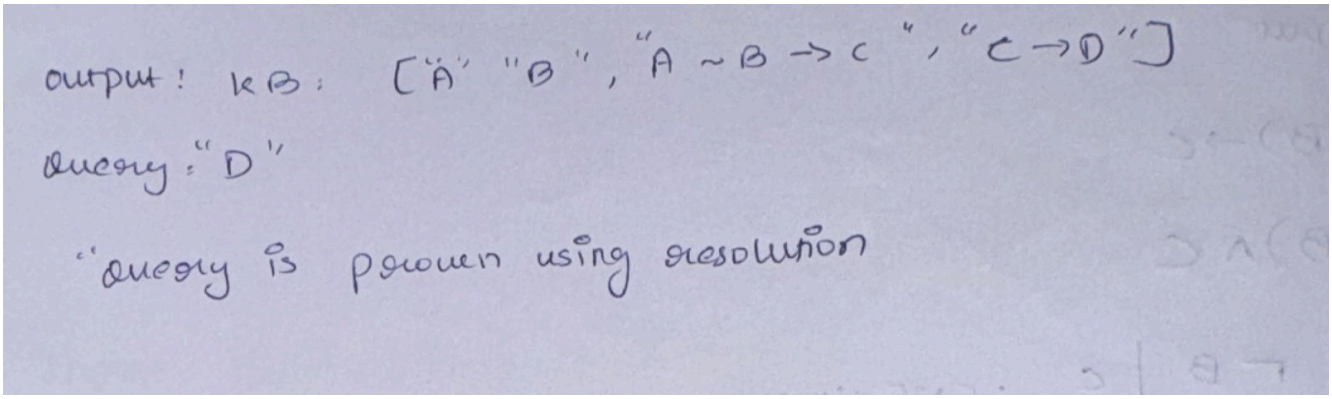
Output Snapshot

```
Enter rule: (p^q)vr
Enter the Query: p
**********Truth Table Reference**********
kb alpha
**********
True True
----------
True True
----------
True True
----------
False True
----------
True False
----------
Tanush Prajwal S
1BM22CS304
The Knowledge Base does not entail the query

=== Code Execution Successful ===
```

Program - 11 Write a proof tree generated using Resolution

Algorithm





Code

```
# Helper function to negate a literal
def negate(literal):
    """Return the negation of a literal."""
    if isinstance(literal, tuple) and literal[0] == "not":
        return literal[1]
    else:
        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.")
```

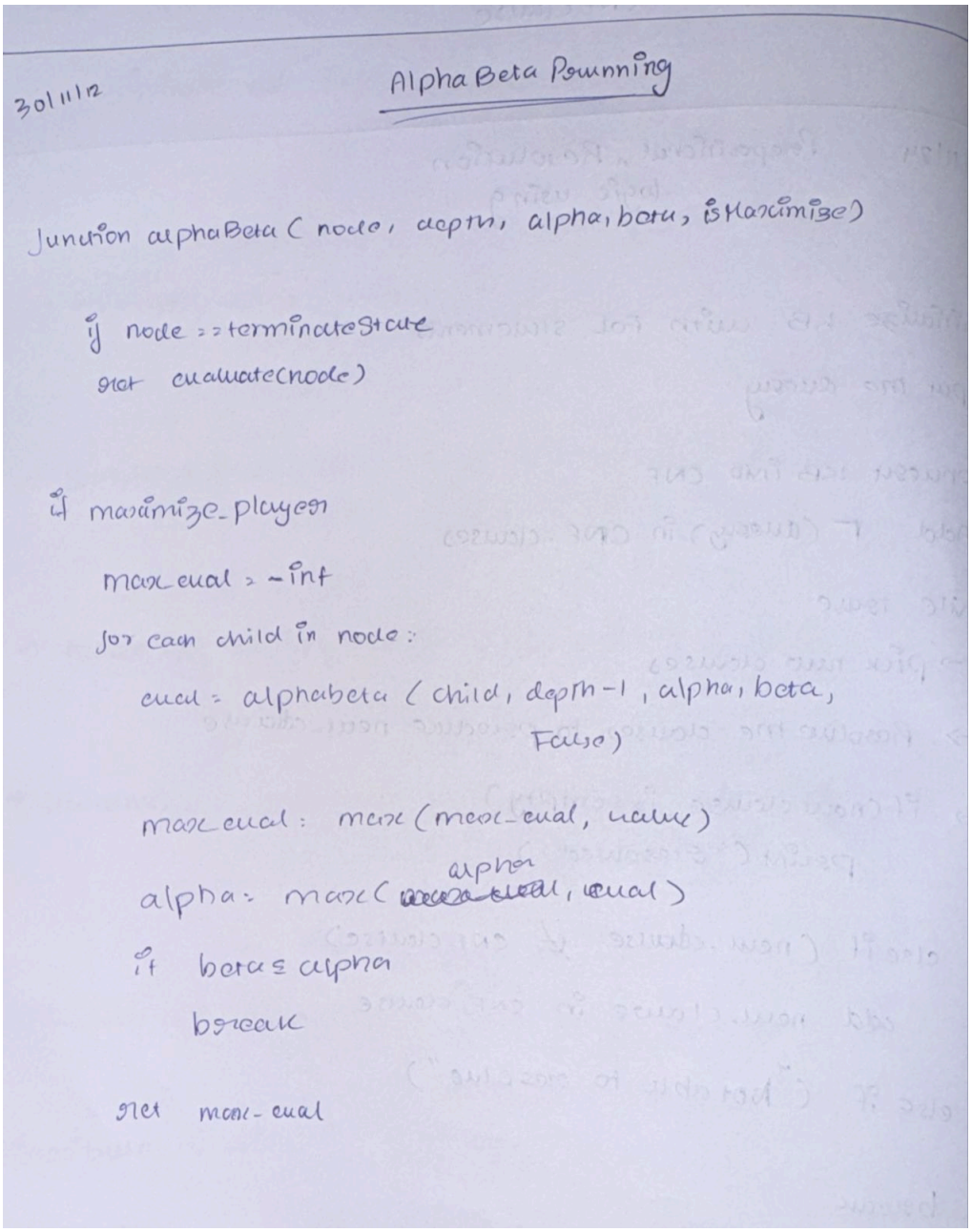
```
else:
  print("\nQuery is not provable.")

print("\nTanush Prajwal S")
print("1BM22CS304")
```

Output Snapshot

```
--- Step-by-Step Resolution Process ---
Negated Query Added to KB: ('not', ('likes', 'John', 'Peanuts'))

Step 1: Resolving Clauses
    Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('food', 'vegetables')})
    Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('not', ('likes', 'John',
        'Peanuts'))})
    Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('food', 'Apple')})
    Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('eats', 'Anil', 'Peanuts'
        )})
    Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('killed', 'y'), ('food',
        'z'), ('not', 'eats(y, z)')})
    Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('likes', 'John', 'Peanuts'
        )})
    Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('eats', 'Harry', 'w'),
        ('not', 'eats(Anil, w)')})
    Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('likes', 'John', 'x'),
        ('not', 'food(x)')})
    Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('killed', 'g'), ('alive',
        'g')})
    Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('alive', 'Anil')})
    Resolving clauses: frozenset({('food', 'vegetables')}) and frozenset({('not', 'killed(k)'), ('not', 'alive(k)')})
    Resolving clauses: frozenset({('food', 'vegetables')}) and frozenset({('not', ('likes', 'John', 'Peanuts'))})
    Resolving clauses: frozenset({('food', 'vegetables')}) and frozenset({('food', 'Apple')})
    Resolving clauses: frozenset({('food', 'vegetables')}) and frozenset({('eats', 'Anil', 'Peanuts')})
    Resolving clauses: frozenset({('food', 'vegetables')}) and frozenset({('killed', 'y'), ('food', 'z'), ('not', 'eats
        (y, z)')})
    Resolving clauses: frozenset({('food', 'vegetables')}) and frozenset({('likes', 'John', 'Peanuts')})
    Resolving clauses: frozenset({('food', 'vegetables')}) and frozenset({('eats', 'Harry', 'w'), ('not', 'eats(Anil, w
        )')})
    Resolving clauses: frozenset({('food', 'vegetables')}) and frozenset({('likes', 'John', 'x'), ('not', 'food(x)')})
    Resolving clauses: frozenset({('food', 'vegetables')}) and frozenset({('killed', 'g'), ('alive', 'g')})
    Resolving clauses: frozenset({('food', 'vegetables')}) and frozenset({('alive', 'Anil')})
    Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('not', 'killed(k)'), ('not',
        'alive(k)')})
    Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('food', 'vegetables')})
    Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('food', 'Apple')})
    Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('eats', 'Anil', 'Peanuts')})
    Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('killed', 'y'), ('food', 'z'),
        ('not', 'eats(y, z)')})
    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.

Tanush Prajwal S
1BM22CS304
```

Program - 12 Alpha-Beta values generated to identify the final value of MAX node and subtrees pruned for a given Game tree using the Alpha-Beta pruning algorithm

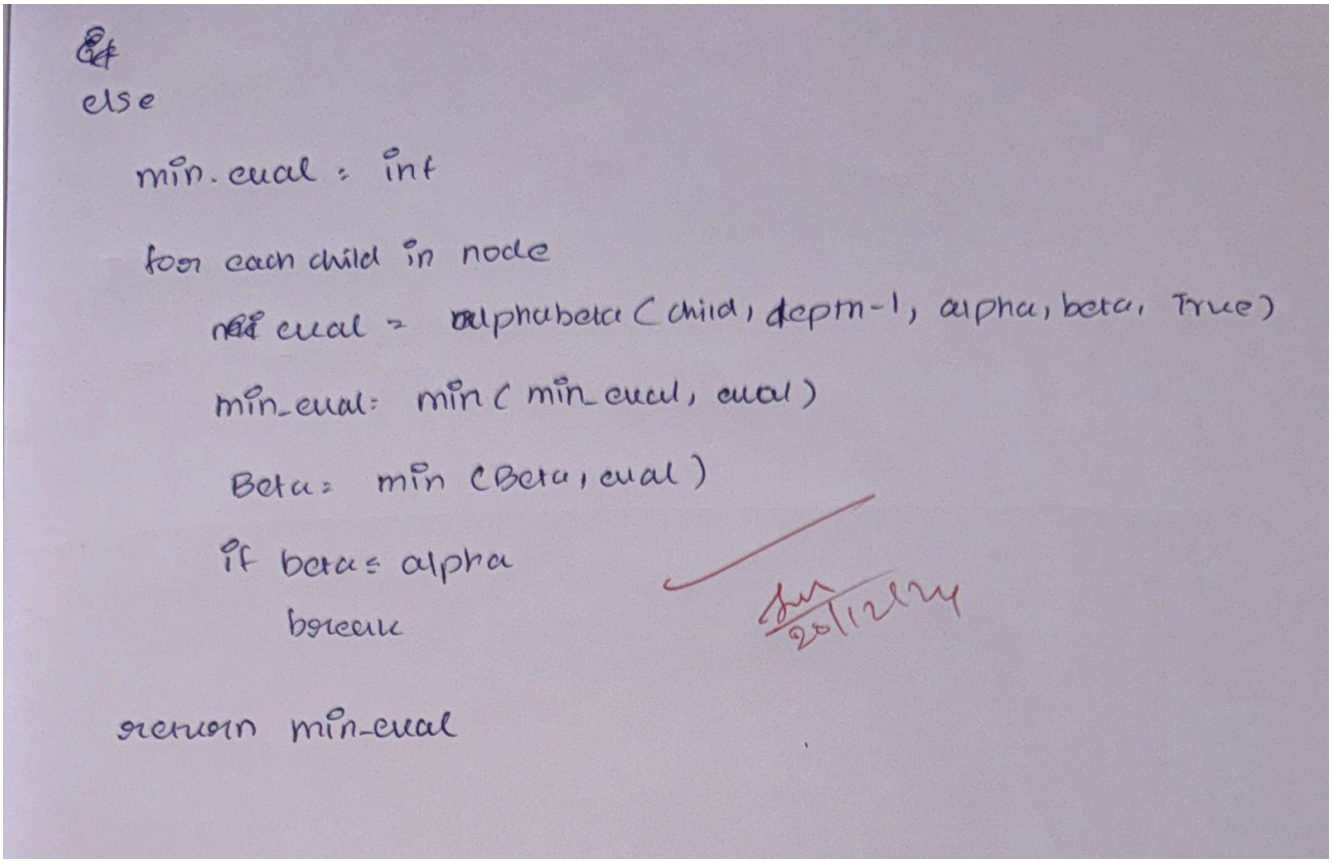
Algorithm



30/11/12

Alpha Beta Pruning

```
function alphaBeta( node, depth, alpha, beta, isMaximize)

    if node == terminateState
    ret evaluate(node)

if maximize_player
    max_eval = -inf
    for each child in node:
        eval = alphabeta ( child, depth-1, alpha, beta,
                           False)

        max_eval: max (max_eval, value)
        alpha: max( alpha, eval)
        if beta ≤ alpha
            break

    ret max_eval
```



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}")

print("Tanush Prajwal S")
print("1BM22CS304\n")

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

Output Snapshot

```
Tanush Prajwal S
1BM22CS304

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
```