VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Paarth Sanyal(1BM22CS188)

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 **Paarth Sanyal(1BM22CS188) ,**who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Sneha S Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor& HOD Department of CSE, BMSCE

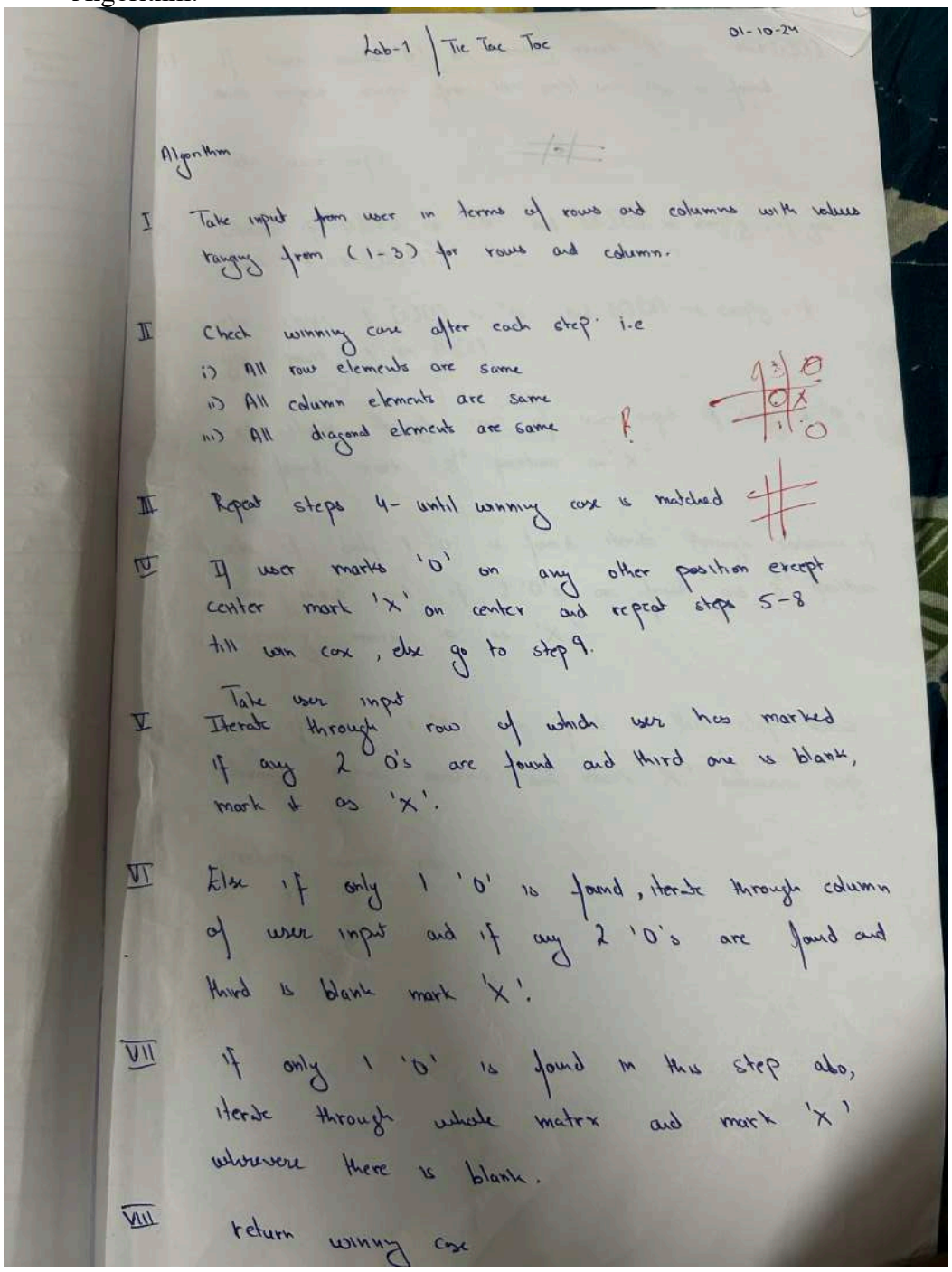
Index

Sl. No.	Date	Experiment Title	Page No.
1	1-10-2024	Implement Tic –Tac –Toe Game	
2	1-10-2024	Implement vacuum cleaner agent	
3	08-10-2024	Implement 8 puzzle problems using Depth First Search (DFS)	
4	15-10-2024	Implement Iterative deepening search algorithm Implement A* search algorithm to solve N-Queens problem	
5	22-10-2024	Simulated Annealing	
6	29-10-2024	Solve 8-Queens problem	
7	12-11-2024	Create a knowledge base using propositional logic	
8	19-11-2024	Create a knowledge base consisting of first order logic statements.	
9	03-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	
10	03-12-2024	Implement Alpha-Beta Pruning.	

GITHUB LINK: https://github.com/ski69per/AI-LAB

Implement Tic –Tac –Toe Game

Algorithm:



Lab-1 | Tic Tac Toe 01-10-24

Algorithm

I Take input from user in terms of rows and columns with values ranging from (1-3) for rows and column.

II Check winning case after each step i.e
i) All row elements are same
ii) All column elements are same
iii) All diagonal elements are same

III Repeat steps 4- until winning case is matched

IV If user marks '0' on any other position except center mark 'x' on center and repeat steps 5-8 till win case, else go to step 9.

V Take user input
Iterate through row of which user has marked if any 2 0's are found and third one is blank, mark it as 'x'.

VI Else if only 1 '0' is found, iterate through column of user input and if any 2 '0's are found and third is blank mark 'x'.

VII if only 1 '0' is found in this step also, iterate through whole matrix and mark 'x' whereever there is blank.

VIII return winning case

9) If user marks '0' on center, mark 'x' on mat[1][1] and repeat steps from 10- until win case is found

10) Take user input

11) Check if [1][3] is '0' and [3][1] is empty, if yes mark 'x' on [3][1].

12) else check if [3][1] is '0' and [1][3] is empty, if yes mark 'x' on [1][3]

13) else iterate through row of user input if only 2 '0's are found, mark 3rd position as 'x'.

14) else if only 1 '0' is found iterate through column of user input and if 2 '0's are found and 3rd position is empty mark is as 'x'

15) else if only 1 '0' is found in this step, iterate through entire matrix and mark 'x' wherever empty

16) return winning case

Stop

1) Create the game board

2) Get user's move and capture the input

3) Verify that selected cell is within bounds and not occupied

4) Place cross on neighbours ; for chosen cell loop through its neighbors.

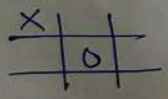
Output:

User move: O at (2,2)
Response: X at (1,1)

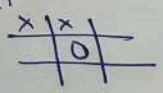
User move : O at (1,2)
Response : X at (1,3)

User move : O at (3,2

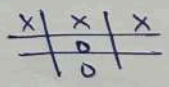
After move 1:



After move 2:



After move 3:



CODE:

```
import random

def initialize_board():
    return [[' ' for _ in range(3)] for _ in range(3)]

def display_board(board):
    for row in board:
        print('|'.join(row))
        print('-' * 5)

def check_winner(board):
    for row in board:
        if row[0] == row[1] == row[2] != ' ':
            return row[0]

    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] != ' ':
            return board[0][col]

    if board[0][0] == board[1][1] == board[2][2] != ' ':
        return board[0][0]

    if board[0][2] == board[1][1] == board[2][0] != ' ':
        return board[0][2]

    return None

def available_moves(board):
    return [(i, j) for i in range(3) for j in range(3) if board[i][j] == ' ']

def check_two_in_a_row(board, player):
    for row in range(3):
        if board[row].count(player) == 2 and board[row].count(' ') == 1:
            return row, board[row].index(' ')

    for col in range(3):
        if [board[row][col] for row in range(3)].count(player) == 2:
            empty_index = [row for row in range(3) if board[row][col] == ' ']
            if empty_index:
                return empty_index[0], col


    if [board[i][i] for i in range(3)].count(player) == 2:
        empty_index = [i for i in range(3) if board[i][i] == ' ']
        if empty_index:
            return empty_index[0], empty_index[0]

    if [board[i][2 - i] for i in range(3)].count(player) == 2:
        empty_index = [i for i in range(3) if board[i][2 - i] == ' ']
        if empty_index:
            return empty_index[0], 2 - empty_index[0]

    return None
```

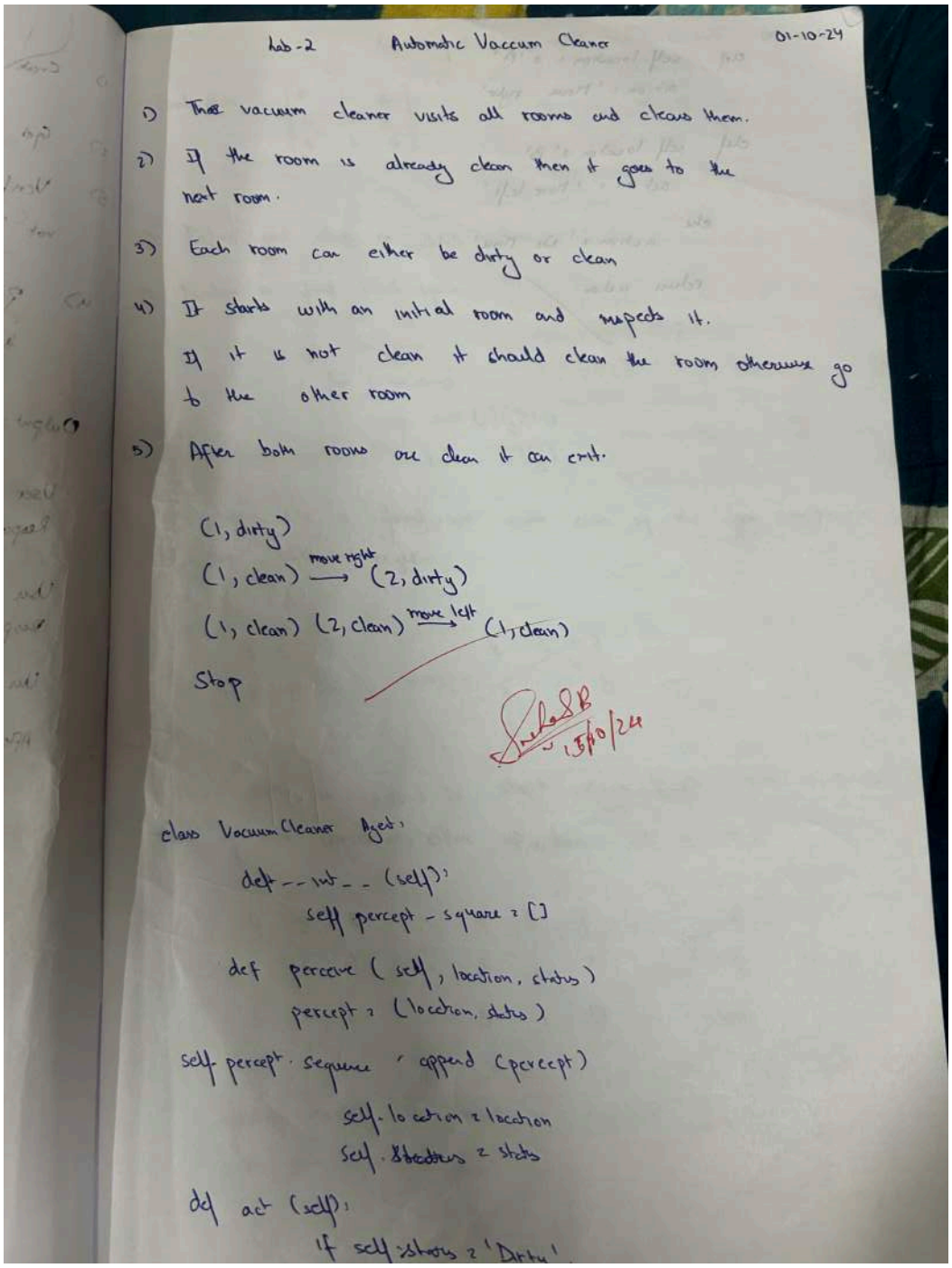
```
def make_move(board, player, move):
    board[move[0]][move[1]] = player

def computer_move(board):

    move = check_two_in_a_row(board, 'O')
    if move:
        make_move(board, 'O', move)
        return


    move = check_two_in_a_row(board, 'X')
    if move:
        make_move(board, 'O', move)
        return


    moves = available_moves(board)
    if moves:
        move = random.choice(moves)
        make_move(board, 'O', move)

def user_move(board):
    while True:
        try:
            row = int(input("Enter row (0-2): "))
            col = int(input("Enter column (0-2): "))
            if board[row][col] == ' ':
                make_move(board, 'X', (row, col))
                return
            else:
                print("That spot is already taken. Try again.")
        except (ValueError, IndexError):
            print("Invalid input. Please enter numbers between 0 and 2.")

def play_game():
    board = initialize_board()
    players = ['X', 'O']
    current_player = 0

    for _ in range(9):
        display_board(board)
        if current_player == 0:
            user_move(board)
        else:
            computer_move(board)

        winner = check_winner(board)
        if winner:
            display_board(board)
            print(f"Player {winner} wins!")
            return

        current_player = 1 - current_player
```

```
    display_board(board)
    print("It's a draw!")


play_game()
```

OUTPUT:

```
O goes first!
Computer is thinking...
- - -
- O -
- - -
Enter row and column of X input: 1 0
- - -
X O -
- - -
Computer is thinking...
O - -
X O -
- - -
Enter row and column of X input: 2 2
O - -
X O -
- - X
Computer is thinking...
O - O
X O -
- - X
Enter row and column of X input: 0 1
O X O
X O -
- - X
Computer is thinking...
O Wins!
O X O
X O -
O - X
```

Implement Vaccum Cleaner Agent



lab-2 Automatic Vaccum Cleaner 01-10-24

1) The vacuum cleaner visits all rooms and cleans them.
2) If the room is already clean then it goes to the next room.
3) Each room can either be dirty or clean
4) It starts with an initial room and inspects it. If it is not clean it should clean the room otherwise go to the other room
5) After both rooms are clean it can exit.

(1, dirty)
(1, clean) —move right→ (2, dirty)
(1, clean) (2, clean) —move left→ (1, clean)
Stop

15/10/24

```
class VacuumCleaner Agent:
    def __init__(self):
        self.percept_square = []

    def perceive(self, location, status):
        percept = (location, status)
        self.percept_sequence.append(percept)
        self.location = location
        self.status = status

    def act(self):
        if self.status == 'Dirty'
```

```
elif    self.location == 'A'
        action = 'Move right'
elif    self.location == 'B'
        action = 'Move left'
    else
        action = 'Do Nothing'
    return action
```

Code:

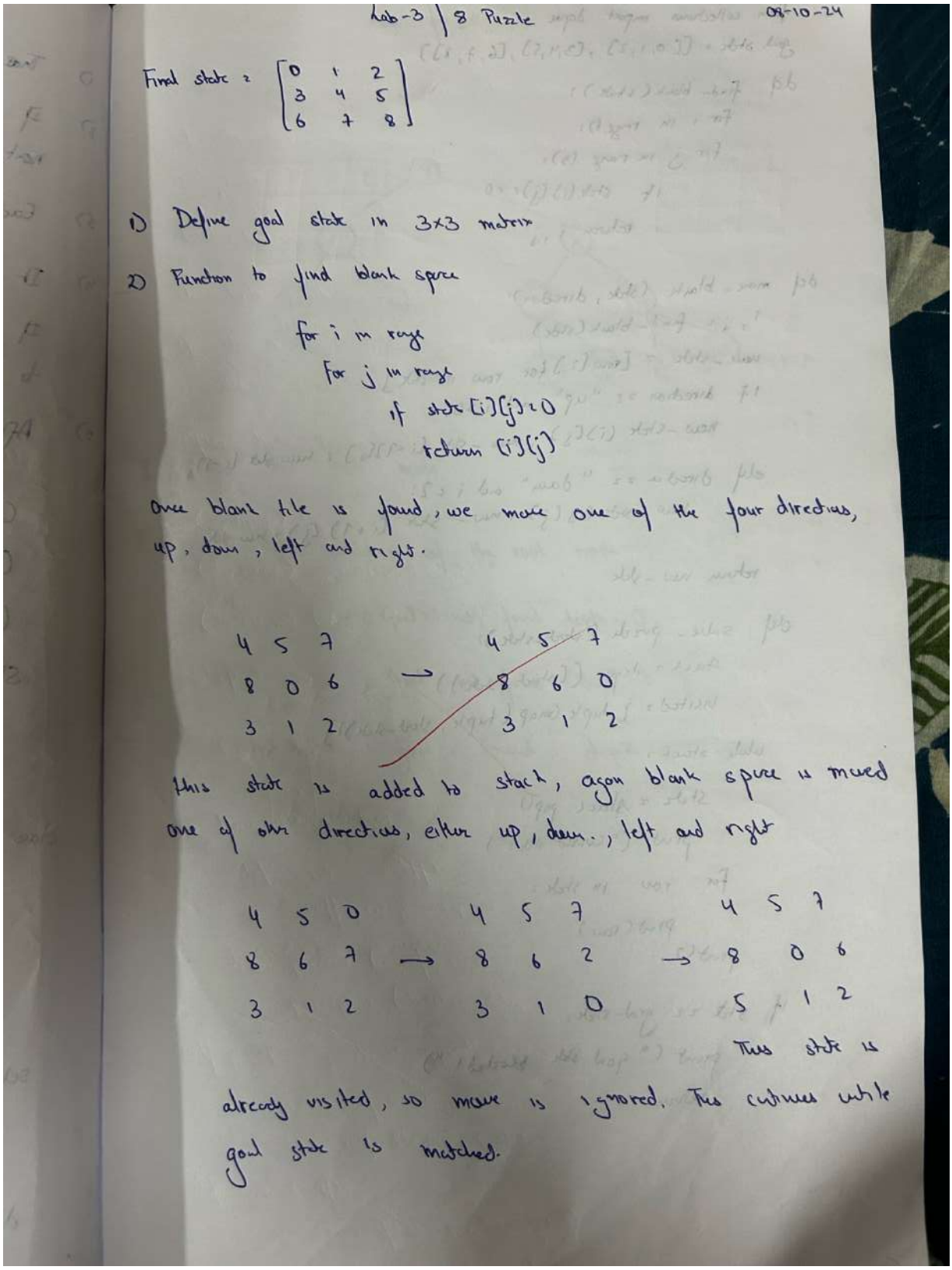
```
if state['A'] == 0 and state['B'] == 0:
print("Turning vacuum off") return


        if state[loc] == 1:
            state[loc] = 0
            count += 1
            print(f"Cleaned {loc}.")
            next_loc = 'B' if loc == 'A' else 'A'
            state[loc] = int(input(f"Is {loc} clean now? (0 if clean, 1 if dirty): "))
            if(state[next_loc]!=1):
              state[next_loc]=int(input(f"Is {next_loc} dirty? (0 if clean, 1 if dirty): "))
        if(state[loc]==1):
          rec(state,loc)
        else:
          next_loc = 'B' if loc == 'A' else 'A'
          dire="left" if loc=="B" else "right"
          print(loc,"is clean")
          print(f"Moving vacuum {dire}")
          if state[next_loc] == 1:
              rec(state, next_loc)


    state = {}
```

```
state['A'] = int(input("Enter state of A (0 for clean, 1 for dirty): "))
state['B'] = int(input("Enter state of B (0 for clean, 1 for dirty): "))
loc = input("Enter location (A or B): ")
```

OUTPUT:

```
Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A
Turning vacuum off
Cost: 0
{'A': 0, 'B': 0}
```

Implement 8 puzzle problems



Lab-3 | 8 Puzzle 08-10-24

Final state = $\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$

1) Define goal state in 3×3 matrix

2) Function to find blank space

```
for i in range
  for j in range
    if state[i][j]=0
      return [i][j]
```

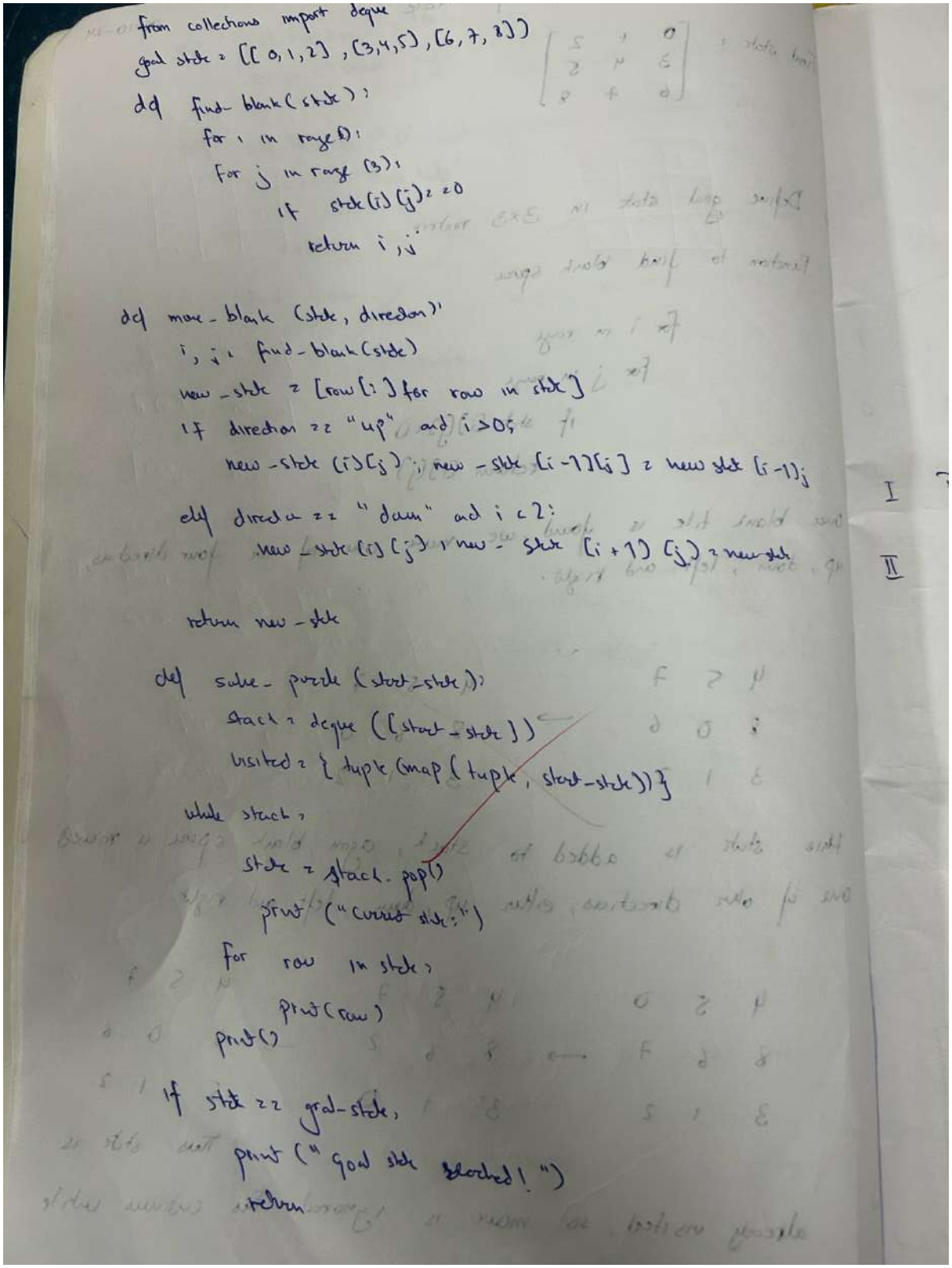
Once blank tile is found, we move one of the four directions, up, down, left and right.

4 5 7 → 4 5 7
8 0 6 → 8 6 0
3 1 2 → 3 1 2

this state is added to stack, again blank space is moved one of other directions, either up, down, left and right

4 5 0 → 4 5 7 → 4 5 7
8 6 7 → 8 6 2 → 8 0 6
3 1 2 → 3 1 0 → 5 1 2

This state is already visited, so move is ignored. This continues until goal state is matched.



```
from collections import deque
goal state = [[0,1,2],[3,4,5],[6,7,8]]
def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0
                return i,j

def move_blank (state, direction):
    i, j = find_blank(state)
    new_state = [row[:] for row in state]
    if direction == "up" and i > 0:
        new_state[i][j], new_state[i-1][j] = new_state[i-1][j],

    elif direction == "down" and i < 2:
        new_state[i][j], new_state[i+1][j] = new_state

    return new_state

def solve_puzzle(start_state):
    stack = deque([start_state])
    visited = {tuple(map(tuple, start_state))}

    while stack:
        state = stack.pop()
        print("current state:")
        for row in state:
            print(row)
        print()

    if state == goal_state:
        print("Goal state reached!")
        return
```

Code:

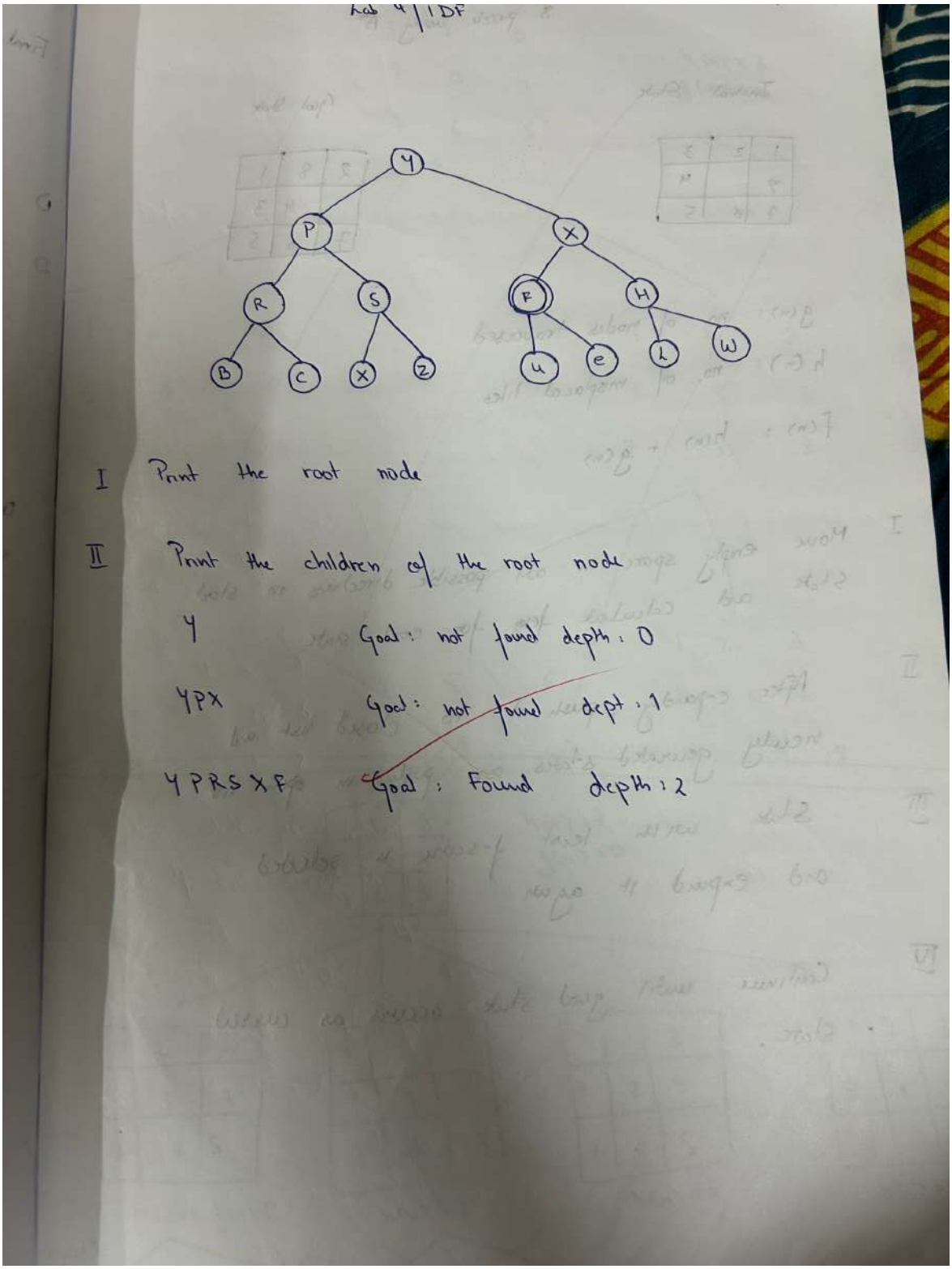
```
class SlidingPuzzle:
    def __init__(self, board, empty_pos, path=[]):
        self.board = board
        self.empty_pos = empty_pos
        self.path = path

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

    def get_moves(self):
        x, y = self.empty_pos
        possible_moves = []
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                new_board = self.board[:]
                new_board[x * 3 + y], new_board[nx * 3 + ny] = new_board[nx * 3 + ny], new_board[x * 3 + y]
                possible_moves.append((new_board, (nx, ny)))
        return possible_moves

def depth_first_search(initial_puzzle):
    stack, visited = [initial_puzzle], set()
    while stack:
        current_puzzle = stack.pop()
        if current_puzzle.is_solved():
            return current_puzzle.path
        visited.add(tuple(current_puzzle.board))
        for new_board, new_empty_pos in current_puzzle.get_moves():
            new_state = SlidingPuzzle(new_board, new_empty_pos, current_puzzle.path + [new_board])
            if tuple(new_board) not in visited:
                stack.append(new_state)
    return None

def display_board(board):
    for i in range(0, 9, 3):
        print(board[i:i + 3])
    print()

def main():
    initial_board = [1, 2, 3, 4, 0, 5, 7, 8, 6]
    empty_pos = initial_board.index(0)
    initial_puzzle = SlidingPuzzle(initial_board, (empty_pos // 3, empty_pos % 3))

    print("Initial state:")
    display_board(initial_board)

    solution = depth_first_search(initial_puzzle)

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

if __name__ == "__main__":
```

main()

OUTPUT:

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

Implement A* Search Algorithm
And IDF



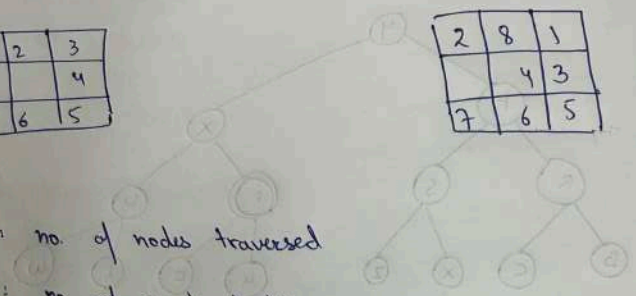
8 puzzle using A*

Initial State

1	2	3
9		4
7	6	5

Goal State

2	8	1
	4	3
7	6	5



$g(n)$: no. of nodes traversed

$h(n)$: no. of misplaced tiles

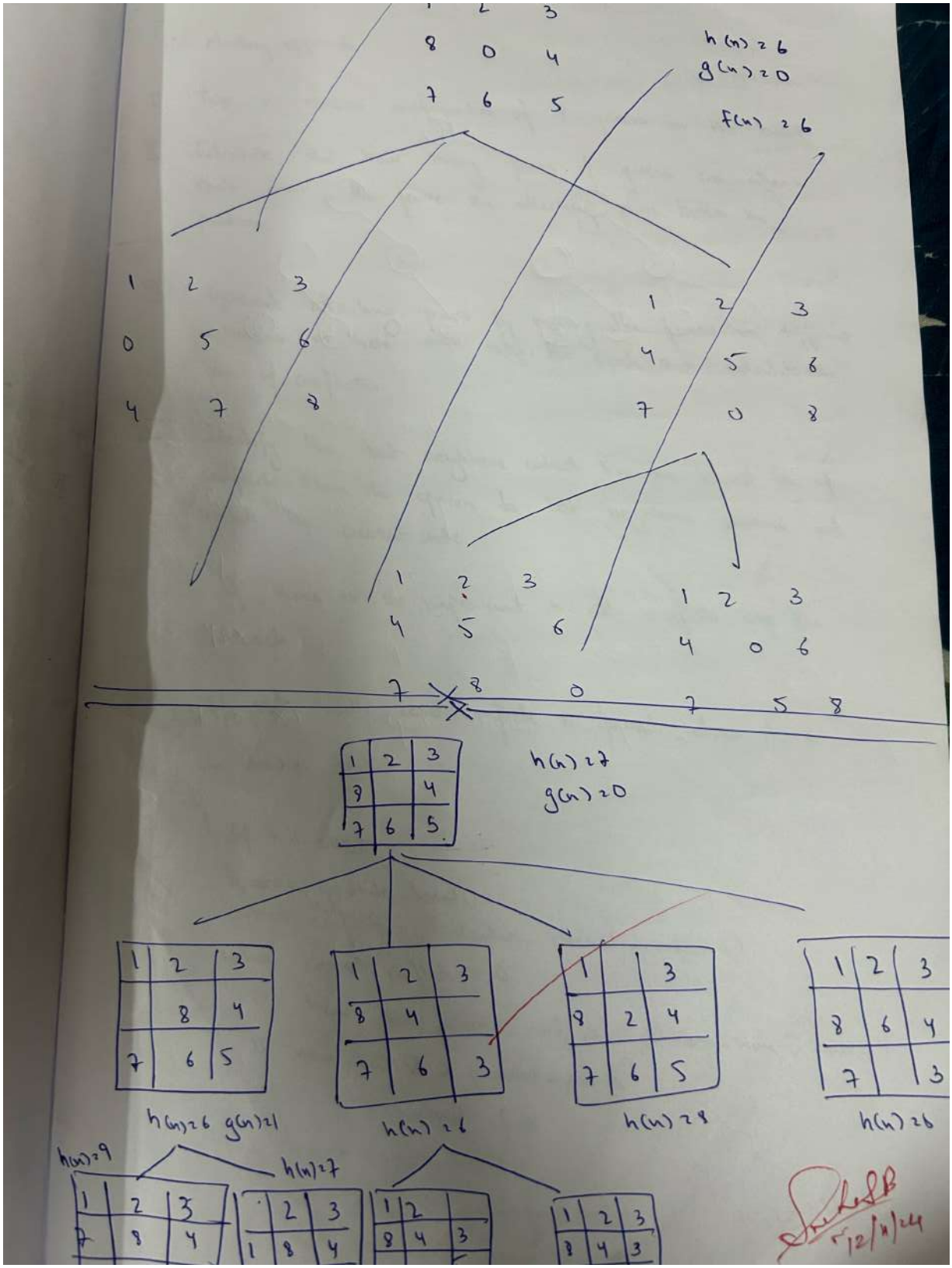
$f(n)$: $h(n) + g(n)$

I. Move empty space in all possible directions in start state and calculate $f(n)$ for each state

II. After expanding, push it to closed list and newly generated states are pushed in open list

III. State with least f-score is selected and expand it again

IV. Continue until goal state occurs as current state.



CODE:

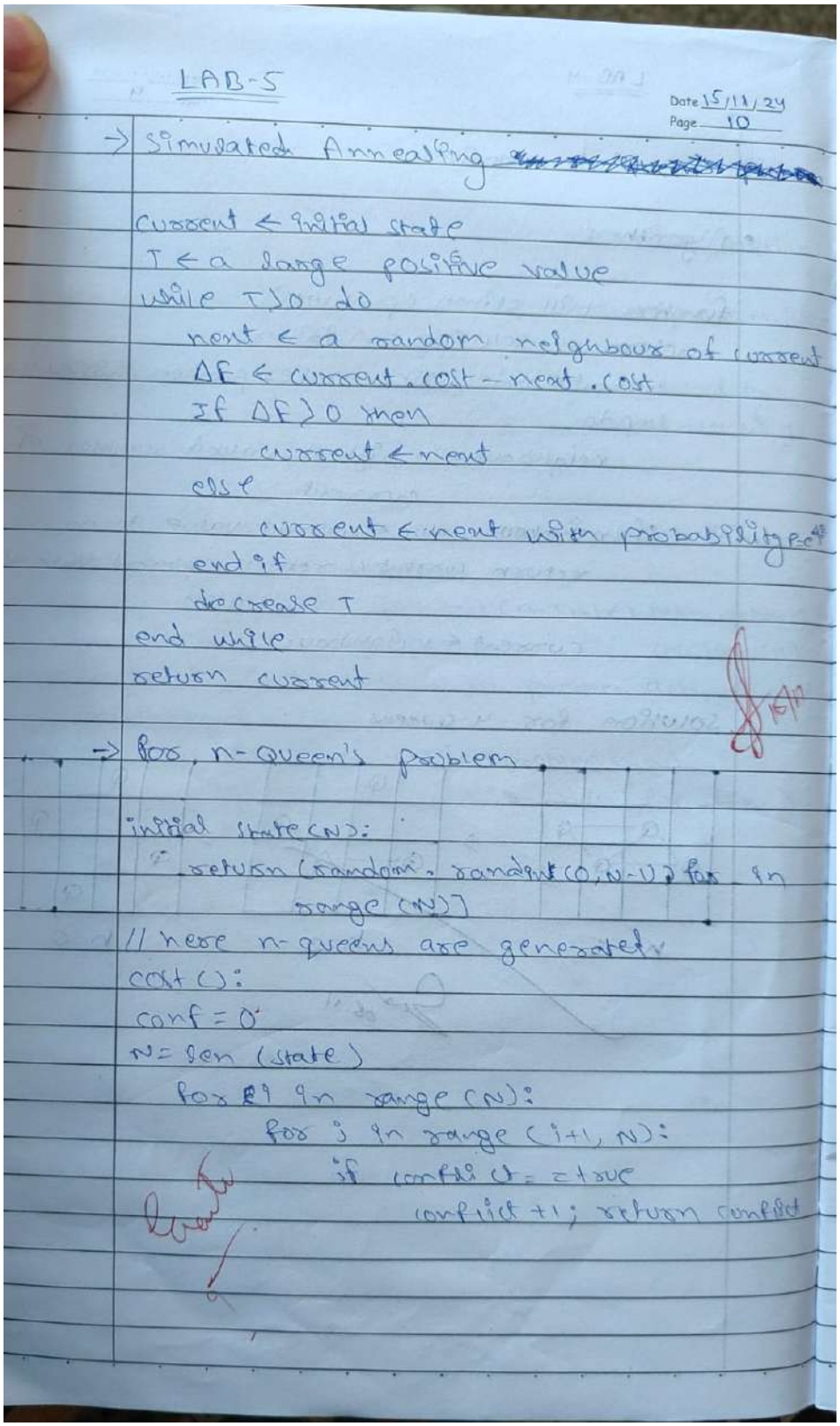
```
import heapq
# Goal state where blank (0) is the first tile
goal_state = [
[0, 1, 2],
[3, 4, 5],
[6, 7, 8]
]
# Helper functions
def flatten(puzzle):
return [item for row in puzzle for item in row]
def find_blank(puzzle):
for i in range(3):
for j in range(3):
if puzzle[i][j] == 0:
return i, j
def misplaced_tiles(puzzle):
flat_puzzle = flatten(puzzle)
flat_goal = flatten(goal_state)
return sum([1 for i in range(9) if flat_puzzle[i] != flat_goal[i] and flat_puzzle[i] != 0])
def generate_neighbors(puzzle):
x, y = find_blank(puzzle)
neighbors = []
moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
for dx, dy in moves:
nx, ny = x + dx, y + dy
if 0 <= nx < 3 and 0 <= ny < 3:
new_puzzle = [row[:] for row in puzzle]
new_puzzle[x][y], new_puzzle[nx][ny] = new_puzzle[nx][ny], new_puzzle[x][y]
neighbors.append(new_puzzle)
return neighbors
def is_goal(puzzle):
return puzzle == goal_state
def print_puzzle(puzzle):
for row in puzzle:
print(row)
print()
def a_star_misplaced_tiles(initial_state):
# Priority queue (min-heap) and visited states
frontier = []
heapq.heappush(frontier, (misplaced_tiles(initial_state), 0, initial_state, []))
visited = set()
while frontier:
f, g, current_state, path = heapq.heappop(frontier)
# Print the current state
print("Current State:")
print_puzzle(current_state)
h = misplaced_tiles(current_state)
print(f"g(n) = {g}, h(n) = {h}, f(n) = {g + h}")
print("-" * 20)
if is_goal(current_state):
print("Goal reached!")
return path
visited.add(tuple(flatten(current_state)))
for neighbor in generate_neighbors(current_state):
if tuple(flatten(neighbor)) not in visited:
h = misplaced_tiles(neighbor)
```

```
heapq.heappush(frontier, (g + 1 + h, g + 1, neighbor, path + [neighbor]))
return None # No solution found
# Initial puzzle state
initial_state = [
[1, 2, 0],
[3, 4, 5],
[6, 7, 8]
]
solution = a_star_misplaced_tiles(initial_state)
if solution:
print("Solution found!")
else:
print("No solution found.")
```

OUTPUT:

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

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

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

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

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

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

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

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

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

Step: 9
[2, 8, 1]
[0, 4, 3]
[7, 6, 5]

Goal Reached
```

Simulated Annealing to Solve 8-Queens problem.



LAB-5

Date 15/11/24
Page 10

→ Simulated Annealing

```
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 e^(ΔE/T)
    end if
    decrease T
end while
return current
```

→ for n-Queen's problem

```
initial state(N):
    return [random.randint(0, N-1) for in
            range(N)]
// here n-queens are generated
cost():
conf = 0
N = len(state)
    for i in range(N):
        for j in range(i+1, N):
            if conflict() == true
                conflict +1; return conflict
```

CODE:
code:-

```
import numpy as np
import math
import random

def objective_function(x):
    """Objective function to minimize: f(x) = x^2"""
    return x ** 2

def simulated_annealing(initial_state, initial_temp, cooling_rate, max_iterations):
    """Simulated Annealing algorithm to find the minimum of the objective function."""

    current_state = initial_state
    current_energy = objective_function(current_state)
    best_state = current_state
    best_energy = current_energy

    temp = initial_temp

    for iteration in range(max_iterations):
        # Generate a new candidate state by perturbing the current state
        candidate_state = current_state + random.uniform(-1, 1)
        candidate_energy = objective_function(candidate_state)

        # Calculate energy difference
        energy_diff = candidate_energy - current_energy

        # If the candidate state is better, or accepted with a certain probability
        if energy_diff < 0 or random.uniform(0, 1) < math.exp(-energy_diff / temp):
            current_state = candidate_state
            current_energy = candidate_energy

            # Update best state found
            if current_energy < best_energy:
                best_state = current_state
                best_energy = current_energy

        # Cool down the temperature
        temp *= cooling_rate

        # Print the current state and temperature for debugging
        print(f"Iteration {iteration + 1}: Current State = {current_state:.4f}, Current Energy =
{current_energy:.4f}, Temperature = {temp:.4f}")

    return best_state, best_energy

# Get user input for parameters
try:
    initial_state = float(input("Enter the initial state (starting point): "))
    initial_temp = float(input("Enter the initial temperature: "))
    cooling_rate = float(input("Enter the cooling rate (between 0 and 1): "))
    max_iterations = int(input("Enter the number of iterations: "))

    # Validate cooling rate
```

```
    if cooling_rate <= 0 or cooling_rate >= 1:
        raise ValueError("Cooling rate must be between 0 and 1.")

    # Execute the simulated annealing algorithm
    best_state, best_energy = simulated_annealing(initial_state, initial_temp, cooling_rate,
max_iterations)

    # Output the best state and energy found
    print(f"Best State: {best_state:.4f}, Best Energy: {best_energy:.4f}")

except ValueError as e:
    print(f"Invalid input: {e}")
```

OUTPUT:

```
Best solution: x = -1.0362423205966222
Best energy: f(x) = 0.0013135058022284 43
Total iterations: 110
```

Solve 8-Queens problem

Lab 6 | 8 Queens

Hill climbing approach

I Take a random configuration of 8 queens on the board

II Calculate that how many pairs of queens can attack each other, the fewer the attacking pairs better the positions.

III Generate attacking pairs by moving the queens in different columns in their own row. For each state calculate the no. of conflicts.

IV Identify the best neighbour which has the lowest no. of conflicts. Move the queen to best neighbour column and update the current state

V If there is no improvement in conflicts, stop the search.

VI If state with zero conflicts is found, return it as the solution state.

```
def hill_climbing
board = generate_board()
current conflicts = Calculate_current_conflicts()
while current_conflicts > 0
        new_board, new_conflicts = get_best_move(board)
    if new_conflicts > current_conflicts
            return false
    board = new_board
    current_conflicts = new_conflicts
    if new conflicts < current_conflicts
        board = new board
```

Proceed

Output

Solution : [2, 0, 6, 4, 1, 7, 5, 3]

This means

Row 0: Queen in 2

Row 1: Queen in 0

Row 2: _ _ Q _ _ _ _ _ 6

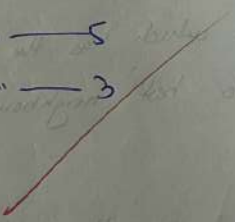
Row 3: _ _ _ _ Q _ _ _ 4

Row 4: _ _ _ _ _ _ Q _ 1

Row 5: _ Q _ _ _ _ _ _ 7

Row 6: _ _ _ _ _ Q _ _ 5

Row 7: _ _ _ Q _ _ _ _ 3



I Take an empty baard

II Create a function that counts how many pairs of queens can attack each other.

III Fewer conflict pairs means a better state

IV Create a goal state function to check if 8 queens placed without any threats to each other

V Use open and closed list to keep track of states to be explored and already explored

VI Sort the open list based on

$f(n) = g(n) + h(n)$

where

$g(n)$ = no. of queens placed so far

$h(n)$ = no. of attacking pairs

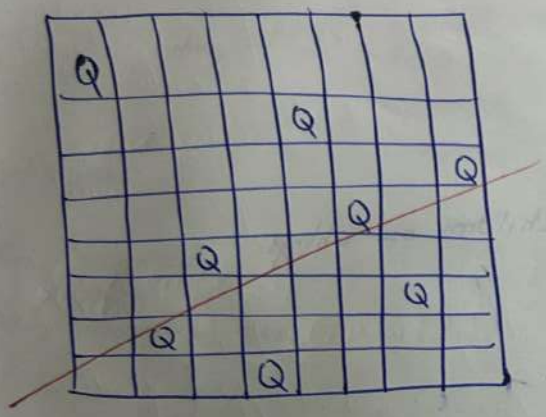
VII Take the state with lowest $f(n)$ from open list, if this state is goal state return it as solution.

VIII Generate new states by placing queen in empty column of row. If this state is not in closed list send it open list

```
function heuristic(state):
    count_attacks = 0
    for each pair of queens in state:
        if queens attack each other:
            count_attack += 1
    return count_attacks

function is_goal(state):
    return len(state) == 8 and heuristic(state) == 0

function a_star_8_queens():
    open_list = priority_queue()
    closed_lst = set()
add (0, []) to open_list

when open lst is not empty:
    current_state = pop_smallest f(n)
    if is_goal(current_state):
        return current_state

add current state to closed_lst.

for each colum in range(8)
    if colum not in current state
    new state = current state + [colum]
    if new state not in closed_lst
    f_n = len(new_state) + heuristic(new_state)
add (f_n) to open list.
```

Output:

Solution: [0, 4, 7, 5, 2, 6, 1, 3]

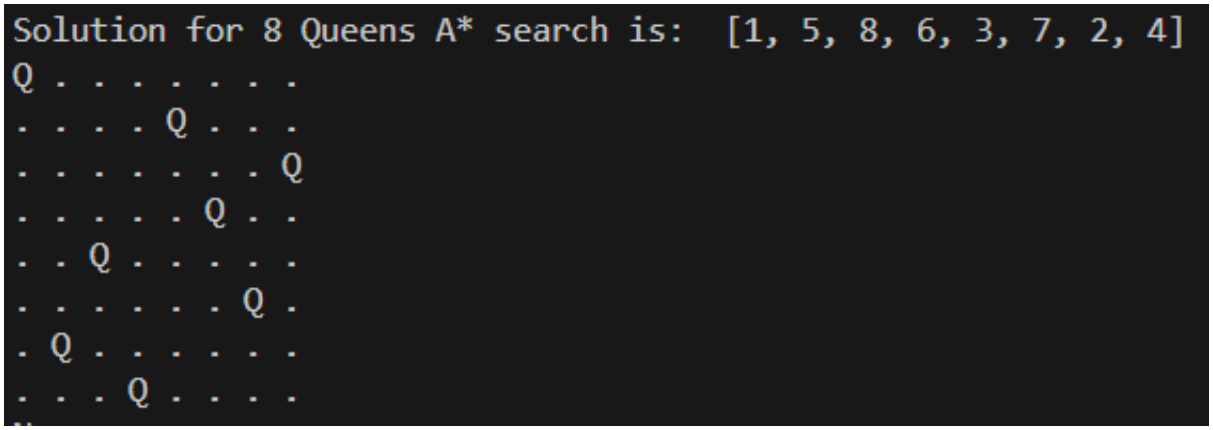


29/10/24

CODE:

```
import numpy as np
import heapq

class Node:
    def __init__(self, state, g, h):
        self.state = state  # current state of the board
        self.g = g          # cost to reach this state
        self.h = h          # heuristic cost to reach goal
        self.f = g + h      # total cost
    def __lt__(self, other):
        return self.f < other.f

def heuristic(state):
    # Count pairs of queens that can attack each other
    attacks = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] == state[j] or abs(state[i] - state[j]) == j - i:
                attacks += 1
    return attacks

def a_star_8_queens():
    initial_state = [-1] * 8  # -1 means no queen placed
    open_list = []
    closed_set = set()

    initial_h = heuristic(initial_state)
    heapq.heappush(open_list, Node(initial_state, 0, initial_h))

    while open_list:
        current_node = heapq.heappop(open_list)
        current_state = current_node.state
        closed_set.add(tuple(current_state))

        # Check if we reached the goal
        if current_node.h == 0:
            return current_state

        for col in range(8):
            for row in range(8):
                if current_state[col] == -1:  # Only place a queen if none is present in this column
                    new_state = current_state.copy()
                    new_state[col] = row
                    if tuple(new_state) not in closed_set:
                        g_cost = current_node.g + 1
                        h_cost = heuristic(new_state)
                        heapq.heappush(open_list, Node(new_state, g_cost, h_cost))

    return None

solution = a_star_8_queens()
print("A* solution:", solution)
```

OUTPUT:

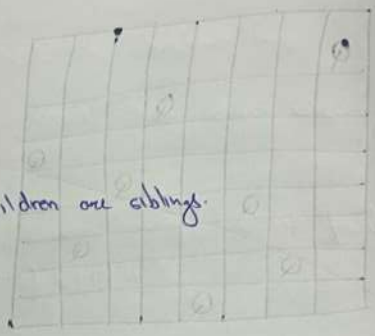


Create a knowledge base using propositional logic

12-11-2024

Propositional Lab

1. Alice is mother of Bob.
2. Bob is father of Charlie.
3. A father is a parent.
4. A mother is a parent
5. All parents have children
6. If someone is a parent, their children are siblings.
7. Alice is married to David.



Hypothesis:

- "Charlie is a sibling of Bob".

Entailment reasoning

1 From statement 1 and 3:
- Alice is mother of Bob, and a mother is parent.
- Therefore, Alice is a parent.

2 From statement 2 and 3:
- Bob is father of Charlie, and a father is parent
- Therefore, Bob is a parent

3 From statement 6:
- If someone is parent, their children are siblings
- Since Bob is parent of Charlie, Alice is parent of Bob, it is determined Charlie is sibling of Bob.

4 Analysis using statement 6:

- Statement 6 implies that children of a parent are siblings.
- However, since Bob is direct parent of Charlie, they cannot be siblings.

Conclusion:

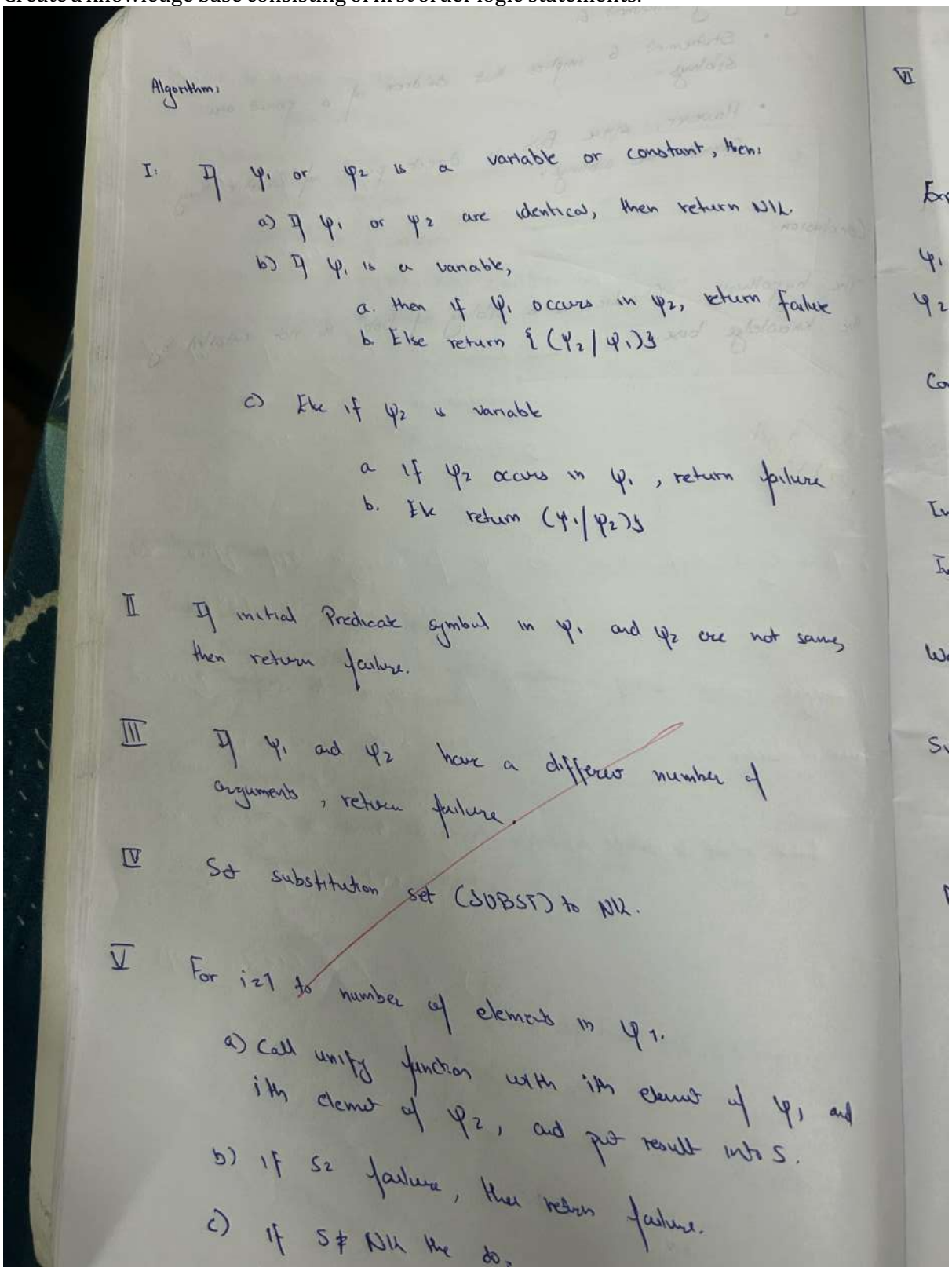
The hypothesis "Charlie is a sibling of Bob" is not entailed by the knowledge base.

19/11/24

CODE:

```
# Function to check entailment based on user input
def check_entailment():
    print("Welcome to the Entailment Checker!")

    # Step 1: Gather user input for facts (Premises)
    alice_is_mother_of_bob = input("Enter the fact: Alice is the mother of Bob. (e.g., 'Alice is the mother of Bob')\n")
    bob_is_father_of_charlie = input("Enter the fact: Bob is the father of Charlie. (e.g., 'Bob is the father of Charlie')\n")
    father_is_parent = input("Enter the fact: A father is a parent. (e.g., 'A father is a parent')\n")
    mother_is_parent = input("Enter the fact: A mother is a parent. (e.g., 'A mother is a parent')\n")
    all_parents_have_children = input("Enter the fact: All parents have children. (e.g., 'All parents have children')\n")
    parents_children_are_siblings = input("Enter the fact: Parents' children are siblings. (e.g., 'Parents' children are siblings')\n")
    alice_is_married_to_david = input("Enter the fact: Alice is married to David. (e.g., 'Alice is married to David')\n")

    # Step 2: Entailment reasoning process
    if ('Alice is the mother of Bob' in alice_is_mother_of_bob and
        'Bob is the father of Charlie' in bob_is_father_of_charlie and
        'A father is a parent' in father_is_parent and
        'A mother is a parent' in mother_is_parent and
        'All parents have children' in all_parents_have_children and
        "Parents' children are siblings" in parents_children_are_siblings and
        'Alice is married to David' in alice_is_married_to_david):

        # Conclusion: Check if Charlie is a sibling of Bob
        print("\nSince Alice is Bob's mother and Bob is Charlie's father, Charlie and Bob are siblings.")
        print("Conclusion: Charlie is a sibling of Bob. The hypothesis is entailed by the knowledge base.")
    else:
        print("\nThe information provided does not fully support the conclusion.")

# Run the function
check_entailment()
```

Create a knowledge base consisting of first order logic statements.



Algorithm:

I: 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) If ψ_1 is a variable,

a. then if ψ_1 occurs in ψ_2, return failure

b. Else return $\{(\psi_2 / \psi_1)\}$

c) Else if ψ_2 is variable

a. if ψ_2 occurs in ψ_1, return failure

b. Else return $(\psi_1 / \psi_2)\}$

II If initial Predicate symbol in ψ_1 and ψ_2 are not same, then return failure.

III If ψ_1 and ψ_2 have a different number of arguments, return failure.

IV Set substitution set (SUBST) to NIL.

V For i=1 to number of elements in ψ_1.

a) Call unify function with ith element of ψ_1 and ith element of ψ_2, and put result into S.

b) If S_2 failure, then return failure.

c) If $S \neq$ NIL then do.

b. SUBST = append(s, SUBST).

VI Return SUBST.

Expressions:

ψ_1 : "loves (x, Dog)"

ψ_2 : "loves (John, y)"

Comparison

ψ_1 is compared to ψ_2

In ψ_1, the first parameter is x and argument is constant Dog
In ψ_2, the first parameter is John and argument is variable y.

We unify variable x in ψ_1 with constant John in ψ_2.

$x = John$

Similarly y in ψ_2 is unified with constant Dog

$\therefore y = Dog$

Applying Substitutions

ψ_1 becomes "loves (John, Dog)"

ψ_2 becomes "loves (John, Dog)"

Unified Expression

Both expressions are now identical. Unification is

ψ_1 : Likes (x, y, Pizza)

ψ_2 : Likes (John, z, y)

In ψ_1, first argument is variable x

Second variable y

Third constn Pizza

In ψ_2, first argument is constn John

Seand is variable z

Third is variable y

We unify first argument of ψ_1 (x) with first argument of ψ_2 (John) ∴ x = John

Similarly second argument of ψ_1 (y) with thrist argument of ψ_2, since both are same variable, no subst is needed

We unify the third argument of ψ_1 (Pizza) with seand argument of ψ_2 (z) ∴ z = pizza

After substitution Likes (John, pizza, y).

```
def Unify(psi1, psi2):

    Args:
        psi1: The first term
        psi2: The second term

    if is_variable(psi1) or is_constant(psi1):
        if is_variable(psi2) or is_constant(psi2):
            if psi1 == psi2:
                return {}
            elif is_variable(psi1):
                if occurs(psi1, psi2):
                    return None
                else:
                    return {psi1: psi2}
            elif is_variable(psi2):
                if occurs(psi2, psi1):
                    return None
                else:
                    return {psi2: psi1}
            else:
                return None

    if predicate_symbol(psi1) != predicate_symbol(psi2):
        return None

    if len(args(psi1)) != len(args(psi2)):
        return None

    subst = {}
```

```
for i in range (len (args (psi 1))):
    s = unify ( args (psi 1) [i], args (psi 2) [i])
    if s is None :
        return None
    else
      subst = append ( s, subst)
return subst
```

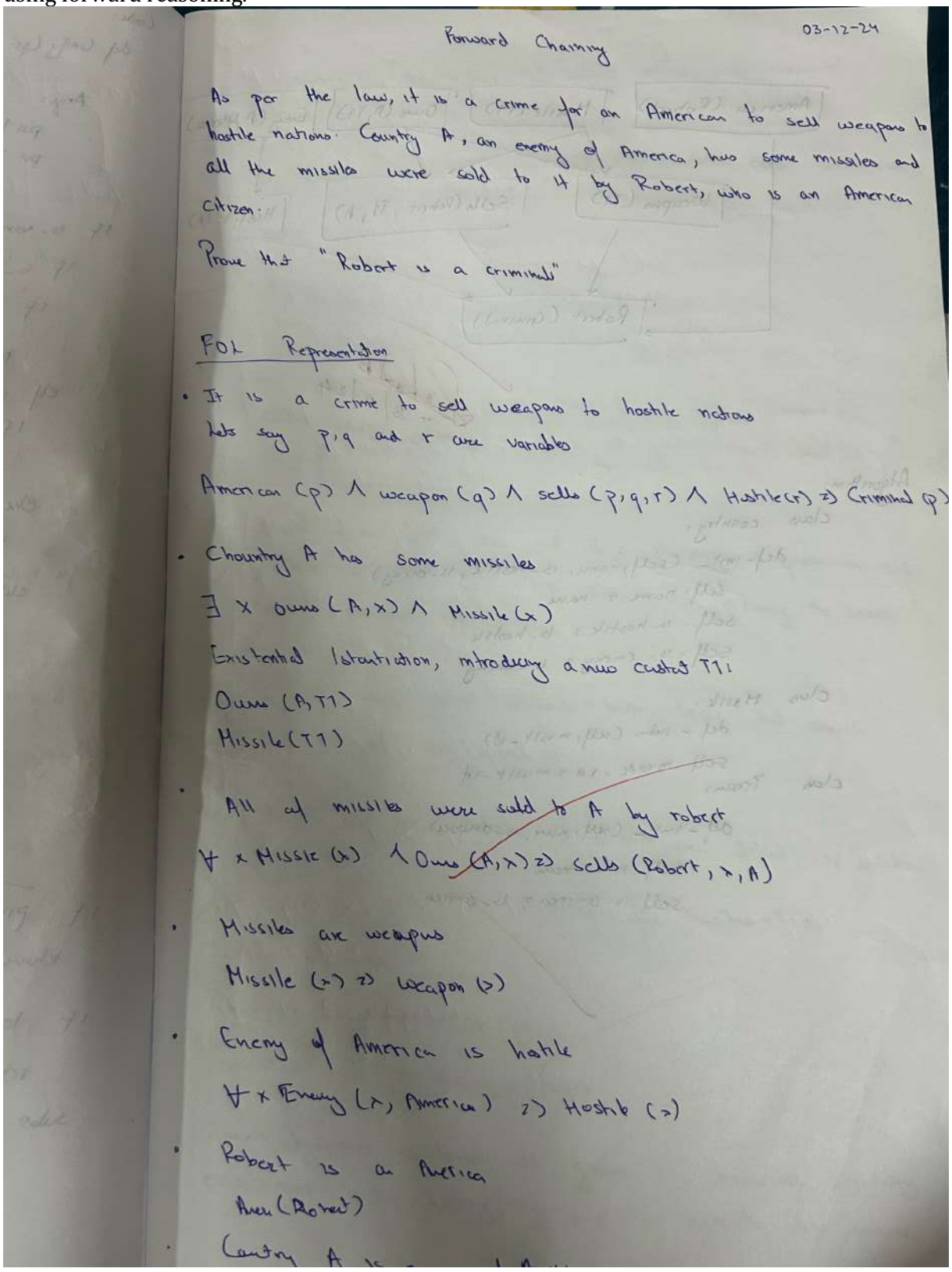
8
19/11/24

CODE:

```
import re
# Define a simple function for extracting predicates from sentences
def extract_predicate(sentence):
# Regular expression to find patterns like Predicate(Argument)
pattern = r"([A-Za-z]+)\((\w+)\)"
match = re.search(pattern, sentence)
if match:
predicate = match.group(1)
subject = match.group(2)
return predicate, subject
return None, None
# Function for unification
def unify(fact, query):
# Check if the fact and query are the same
if fact == query:
return True
# Extract predicate and subject from fact and query
fact_predicate, fact_subject = extract_predicate(fact)
query_predicate, query_subject = extract_predicate(query)
# If predicates match, unify the subjects
if fact_predicate == query_predicate:
if fact_subject == query_subject:
return True
else:
# Here, we could handle variable substitution (unification)
return False
return False
# Function to deduce the goal using given rules
def deduct(rules, goal):
# Try to find unification for the goal from the rules
for rule in rules:
if unify(rule, goal):
print(f"Unification successful: {rule} matches with {goal}.")
return True
return False
# Main function to handle user input
def main():
# Step 1: Get the rules (facts/implications) from the user
print("Enter the rules (facts/implications). Type 'done' to finish entering rules.")
rules = []
while True:
rule_input = input("Enter rule: ")
if rule_input.lower() == 'done':
break
else:
rules.append(rule_input.strip())
# Step 2: Get the goal (query) from the user
goal_input = input("Enter the goal (query) to prove: ").strip()
# Step 3: Try to deduce the goal using the given rules
print("\nAttempting to deduce the goal...")
if deduct(rules, goal_input):
print(f"Conclusion: The goal '{goal_input}' is true based on the rules.")
else:
```

```
print(f"Conclusion: The goal '{goal_input}' cannot be proven with the
provided rules.")
# Run the program
main()
```

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



Forward Chaining

03-12-24

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

Prove that "Robert is a criminal"

FOL Representation

- It is a crime to sell weapons to hostile nations
  Lets say p, q and r are variables

  American(p) ∧ weapon(q) ∧ sells(p, q, r) ∧ Hostile(r) ⇒ Criminal(p)

- Chountry A has some missiles

  ∃ x Owns(A, x) ∧ Missile(x)

  Existential Instantiation, introducing a new constant T1:

  Owns(A, T1)

  Missile(T1)

- All of missiles were sold to A by robert

  ∀ x Missile(x) ∧ Owns(A, x) ⇒ Sells(Robert, x, A)

- Missiles are weapons

  Missile(x) ⇒ Weapon(x)

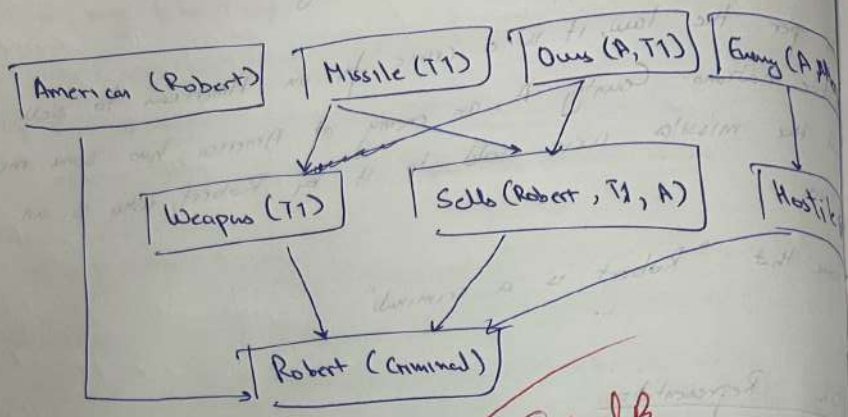
- Enemy of America is hostile

  ∀ x Enemy(x, America) ⇒ Hostile(x)

- Robert is an American

  American(Robert)

- Country A



Robert B
3/12/24

Algorithm:

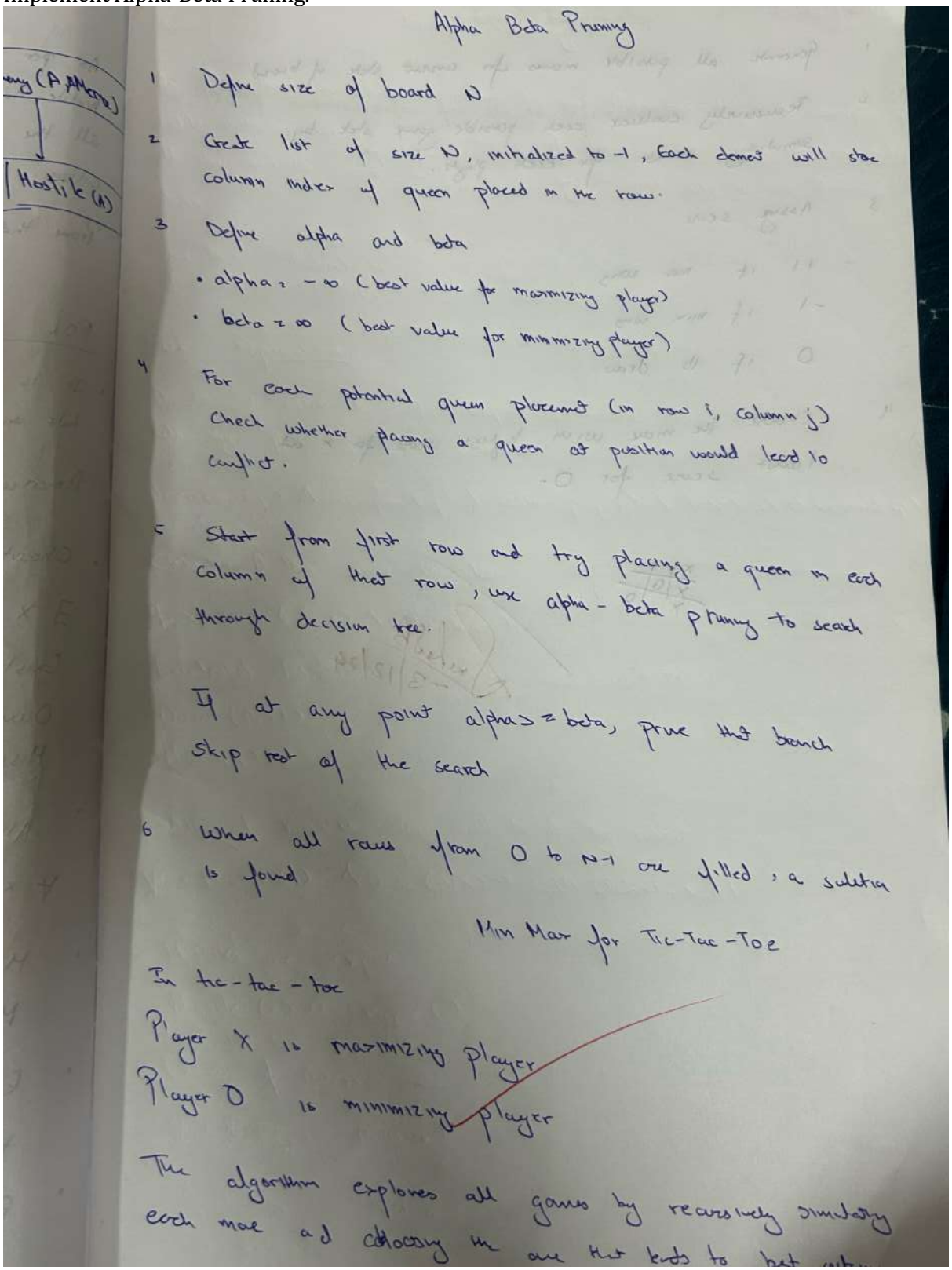
```
class country,
    def-init- (self, name, is_hostile, is_enemy):
        self, name = name
        self. is_hostile = is_hostile
        self - is- enemy = is_enemy
class Missile,
        def - init- (self, missile_id)
        self. missile - id = missile-id
class Person,
        def - init - (self, name, is_american)
        self name = name
        self. is-american = is_america
```

CODE:

```
def is_variable(term):
    """
    Check if a term is a variable.
    Variables are typically single lowercase letters.
    """
    return isinstance(term, str) and term.islower()

def unify(expr1, expr2, subst={}):
    """
    Unify two expressions expr1 and expr2 under the given substitution subst.
    """
    if subst is None:
        return None  # Failure case
    if expr1 == expr2:
        return subst  # Expressions are identical
    if is_variable(expr1):
        return unify_variable(expr1, expr2, subst)
    if is_variable(expr2):
        return unify_variable(expr2, expr1, subst)
    if isinstance(expr1, tuple) and isinstance(expr2, tuple):
        if len(expr1) != len(expr2):
            return None  # Different arity
        # Recursively unify each component
        for arg1, arg2 in zip(expr1, expr2):
            subst = unify(arg1, arg2, subst)
            if subst is None:
                return None  # Failure
        return subst
    return None  # No unification possible

def unify_variable(var, term, subst):
    """
    Unify a variable with a term, updating the substitution.
    """
    if var in subst:
        return unify(subst[var], term, subst)  # Apply substitution to var
    if term in subst:
        return unify(var, subst[term], subst)  # Apply substitution to term
    if occurs_check(var, term, subst):
        return None  # Circular substitution detected
    # Add var -> term to the substitution
    subst = subst.copy()
    subst[var] = term
    return subst

def occurs_check(var, term, subst):
    """
    Check if var occurs in term (directly or indirectly) to prevent circular substitutions.
    """
    if var == term:
        return True
    if isinstance(term, tuple):
        return any(occurs_check(var, t, subst) for t in term)
    if term in subst:
```

```
        return occurs_check(var, subst[term], subst)
    return False

def parse_input(expr):
    """
    Parse user input into a structured format (nested tuples for functions and terms).
    Example: "f(X, g(y))" -> ('f', 'X', ('g', 'y'))
    """
    expr = expr.strip()
    if '(' not in expr:
        return expr  # Simple variable or constant
    func_name = expr[:expr.index('(')].strip()
    args = expr[expr.index('(') + 1:expr.rindex(')')].split(',')
    args = [parse_input(arg.strip()) for arg in args]
    return (func_name, *args)

def format_output(expr):
    """
    Convert the nested tuple representation back into a string for output.
    Example: ('f', 'X', ('g', 'y')) -> "f(X, g(y))"
    """
    if isinstance(expr, str):
        return expr
    return f"{expr[0]}({', '.join(format_output(arg) for arg in expr[1:])})"

# Main Program
if __name__ == "__main__":
    print("Enter the first term:")
    expr1 = parse_input(input().strip())
    print("Enter the second term:")
    expr2 = parse_input(input().strip())

    print("Unifying.....")
    result = unify(expr1, expr2)

    if result is None:
        print("Unification failed")
    else:
        print("Unification succeeded with substitution:")
        for var, term in result.items():
            print(f"{var} -> {format_output(term)}")
```

Implement Alpha-Beta Pruning.



Alpha Beta Pruning

1. Define size of board N
2. Create list of size N, initialized to -1, Each demes will store column index of queen placed in the row.
3. Define alpha and beta
   - alpha = $-\infty$ (best value for maximizing player)
   - beta = ∞ (best value for minimizing player)
4. For each potential queen placement (in row i, column j) Check whether placing a queen at position would lead to conflict.
5. Start from first row and try placing a queen in each column of that row, use alpha - beta pruning to search through decision tree.

If at any point alpha >= beta, prune that branch skip rest of the search

6. When all rows from 0 to N-1 are filled, a solution is found

Min Max for Tic-Tac-Toe

In tic-tac-toe

Player X is maximizing player

Player O is minimizing player

The algorithm explores all games by recursively simulating each move and choosing the one that leads to best

1 Generate all possible moves for current state of board

2 Recursively evaluate each possible game state by simulating moves for each player.

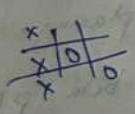
3 Assign score

+1 if max wins

-1 if min wins

0 if its draw.

4 Choose the move with highest score for x and lowest score for O.



3/12/24

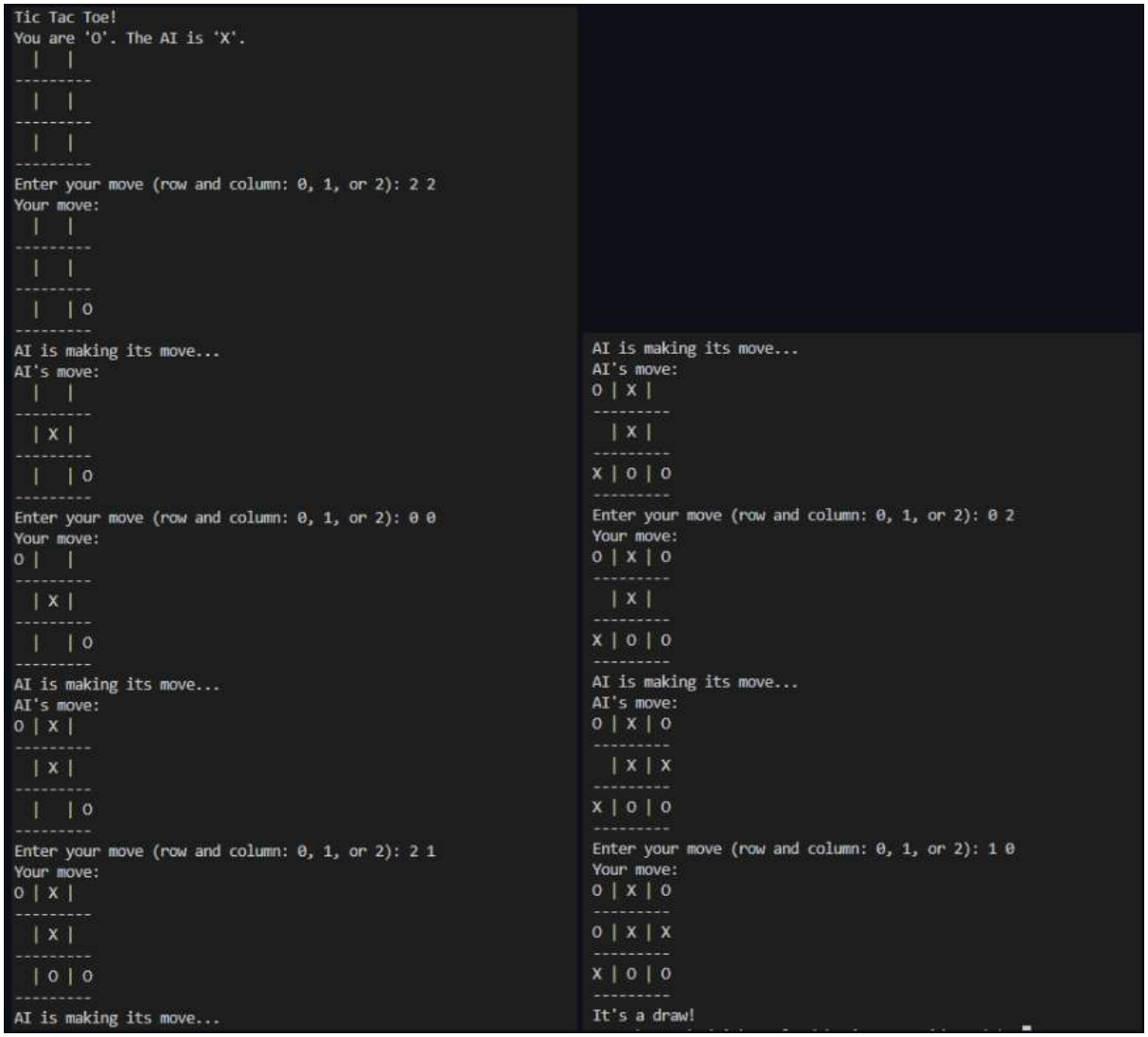
CODE:

```
import math
from copy import deepcopy

# Define the Tic-Tac-Toe board size and players
EMPTY = "-"
PLAYER_X = "X"  # Maximizing player (Computer)
PLAYER_O = "O"  # Minimizing player (User)

# Helper functions
def is_terminal(board):
    """Checks if the game has ended."""
    winner = get_winner(board)
    if winner or not any(EMPTY in row for row in board):
        return True
    return False

def get_winner(board):
    """Checks for a winner on the board."""
    # Check rows and columns
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != EMPTY:
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != EMPTY:
            return board[0][i]
    # Check diagonals
    if board[0][0] == board[1][1] == board[2][2] != EMPTY:
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != EMPTY:
        return board[0][2]
    return None

def utility(board):
    """Returns the utility of a terminal state."""
    winner = get_winner(board)
    if winner == PLAYER_X:
        return 1
    elif winner == PLAYER_O:
        return -1
    return 0

def get_actions(board):
    """Returns a list of possible moves."""
    actions = []
    for i in range(3):
        for j in range(3):
            if board[i][j] == EMPTY:
                actions.append((i, j))
    return actions

def result(board, action, player):
    """Returns the board resulting from applying an action."""
    new_board = deepcopy(board)
    new_board[action[0]][action[1]] = player
    return new_board
```

```
# Alpha-Beta Search
def alpha_beta_search(board):
    """Performs Alpha-Beta Pruning to find the best action."""
    alpha = -math.inf
    beta = math.inf
    best_action = None

    def max_value(state, alpha, beta):
        if is_terminal(state):
            return utility(state)
        v = -math.inf
        for action in get_actions(state):
            v = max(v, min_value(result(state, action, PLAYER_X), alpha, beta))
            if v >= beta:
                return v
            alpha = max(alpha, v)
        return v

    def min_value(state, alpha, beta):
        if is_terminal(state):
            return utility(state)
        v = math.inf
        for action in get_actions(state):
            v = min(v, max_value(result(state, action, PLAYER_O), alpha, beta))
            if v <= alpha:
                return v
            beta = min(beta, v)
        return v

    for action in get_actions(board):
        value = min_value(result(board, action, PLAYER_X), alpha, beta)
        if value > alpha:
            alpha = value
            best_action = action

    return best_action

# Game loop
def print_board(board):
    """Displays the board."""
    for row in board:
        print(" | ".join(row))
    print()

def play_game():
    """Runs the Tic-Tac-Toe game with user input."""
    board = [[EMPTY for _ in range(3)] for _ in range(3)]
    print("Welcome to Tic-Tac-Toe!")
    print("You are 'O', and the computer is 'X'.")
    print_board(board)

    while not is_terminal(board):
        # User's turn
        user_move = None
```

```
        while user_move not in get_actions(board):
            try:
                print("Your turn! Enter your move as 'row col' (e.g., '1 2'):")
                row, col = map(int, input().split())
                user_move = (row - 1, col - 1)  # Convert to 0-based index
                if user_move not in get_actions(board):
                    print("Invalid move! Try again.")
            except ValueError:
                print("Invalid input! Please enter two numbers separated by a space.")

        board = result(board, user_move, PLAYER_O)
        print("You played:")
        print_board(board)

        if is_terminal(board):
            break

        # Computer's turn
        print("Computer's turn...")
        computer_move = alpha_beta_search(board)
        board = result(board, computer_move, PLAYER_X)
        print("Computer played:")
        print_board(board)

    # Game over
    winner = get_winner(board)
    if winner == PLAYER_X:
        print("Computer wins!")
    elif winner == PLAYER_O:
        print("Congratulations! You win!")
    else:
        print("It's a draw!")

# Run the game
if __name__ == "__main__":
    play_game()
```

OUTPUT:


Tic Tac Toe!
You are 'O'. The AI is 'X'.
| |

| |

| |

Enter your move (row and column: 0, 1, or 2): 2 2
Your move:
| |

| |

| | O

AI is making its move...
AI's move:
| |

| X |

| | O

Enter your move (row and column: 0, 1, or 2): 0 0
Your move:
O | |

| X |

| | O

AI is making its move...
AI's move:
O | X |

| X |

| | O

Enter your move (row and column: 0, 1, or 2): 2 1
Your move:
O | X |

| X |

| O | O

AI is making its move...
AI is making its move...
AI's move:
O | X |

| X |

X | O | O

Enter your move (row and column: 0, 1, or 2): 0 2
Your move:
O | X | O

| X |

X | O | O

AI is making its move...
AI's move:
O | X | O

| X | X

X | O | O

Enter your move (row and column: 0, 1, or 2): 1 0
Your move:
O | X | O

O | X | X

X | O | O

It's a draw!