VISVESVARAYATECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Sneha Prasanna (1BM22CS284)

in partial fulfillment for the award of the degree of
BACHELOROFENGINEERING
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 **Sneha Prasanna (1BM22CS284),** 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.

Lab faculty Incharge Name Assistant Professor Department of CSE, BMSCE	Dr. Jyothi S Nayak Professor & HOD Department of CSE, BMSCE

Index

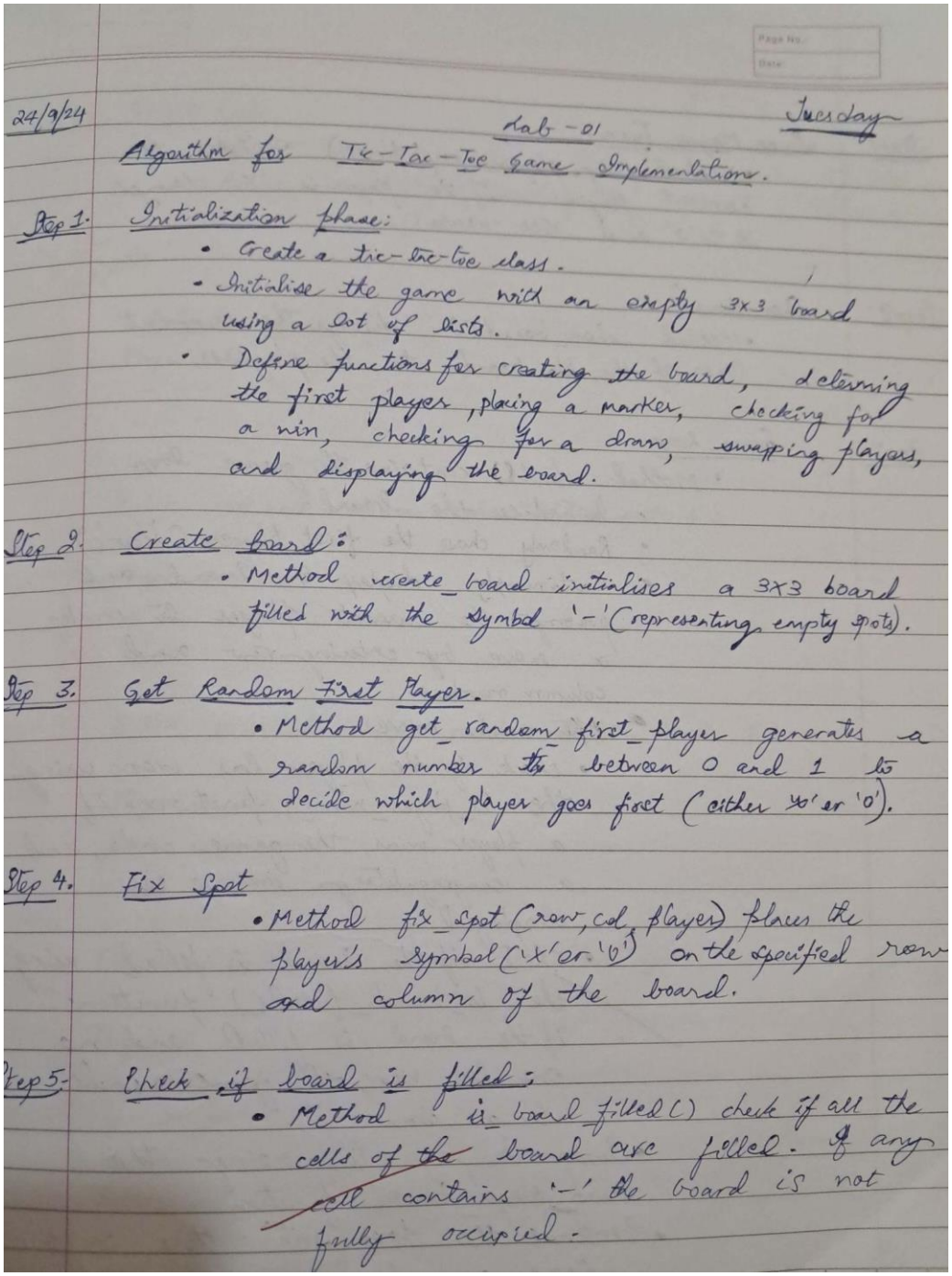
9	16-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	66-71
10	16-12-2024	Implement Alpha-Beta Pruning.	72-74

Github Link: https://github.com/SnehaPrasanna1/SnehaPrasannaAI.git

Program1

Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm



24/9/24 Tuesday

Lab - 01

Algorithm for Tic-Tac-Toe Game Implementation.

Step 1. Initialization phase:
- Create a tic-tac-toe class.
- Initialise the game with an empty 3x3 board using a list of lists.
- Define functions for creating the board, declaring the first player, placing a marker, checking for a win, checking for a draw, swapping players, and displaying the board.

Step 2. Create board:
- Method create_board initialises a 3x3 board filled with the symbol '-' (representing empty spots).

Step 3. Get Random First Player.
- Method get_random_first_player generates a random number that is between 0 and 1 to decide which player goes first (either 'X' or 'O').

Step 4. Fix Spot
- Method fix_spot (row, col, player) places the player's symbol ('X' or 'O') on the specified row and column of the board.

Step 5. Check if board is filled:
- Method is_board_filled() check if all the cells of the board are filled. If any cell contains '-' the board is not fully occupied.

Step7: Swap Player Turn:

- Method swap_players_turn(player) switches the current player. If the player is 'x', change it to 'o' and vice-versa.

Step8: Show Board:

- Method show_board() displays the current state of the ~~pla~~ board to the players.

Step9: Game loop:

- Method start() contains the game loop.
  - Initialize the board.
  - Randomly chose the first player ('x' or 'o')
  - Continously display the board and prompt the current player to make a move by entering row and column numbers.
  - After each move
    - check if the player has won using the is_player_win() function. If a player wins, the game ends, and a congragulatory message is displayed.
    - Check if the board is filled using the is_board_filled() function. If the board is filled and no one has won, the game ends in a draw.
  - Swap the players turn using the swap_player_turn() function.
  - Continue the loop until there is a winner or a draw.

Page No.:
Date:

Step 10: Game End:

- Once the game ends, the final state of the board is displayed, showing the result (either a win or a draw).

24/9/2024

Output of Code

Player X Turn

- - -

- - -

- - -

Enter row and column numbers to fix spot: 1 1

Player O Turn

X - -

- - -

- - -

Enter row and column numbers to fix spot: 1 2

Player X Turn

X O -

- - -

- - -

Enter row and column number to fix spot: 2 1

Player O Turn

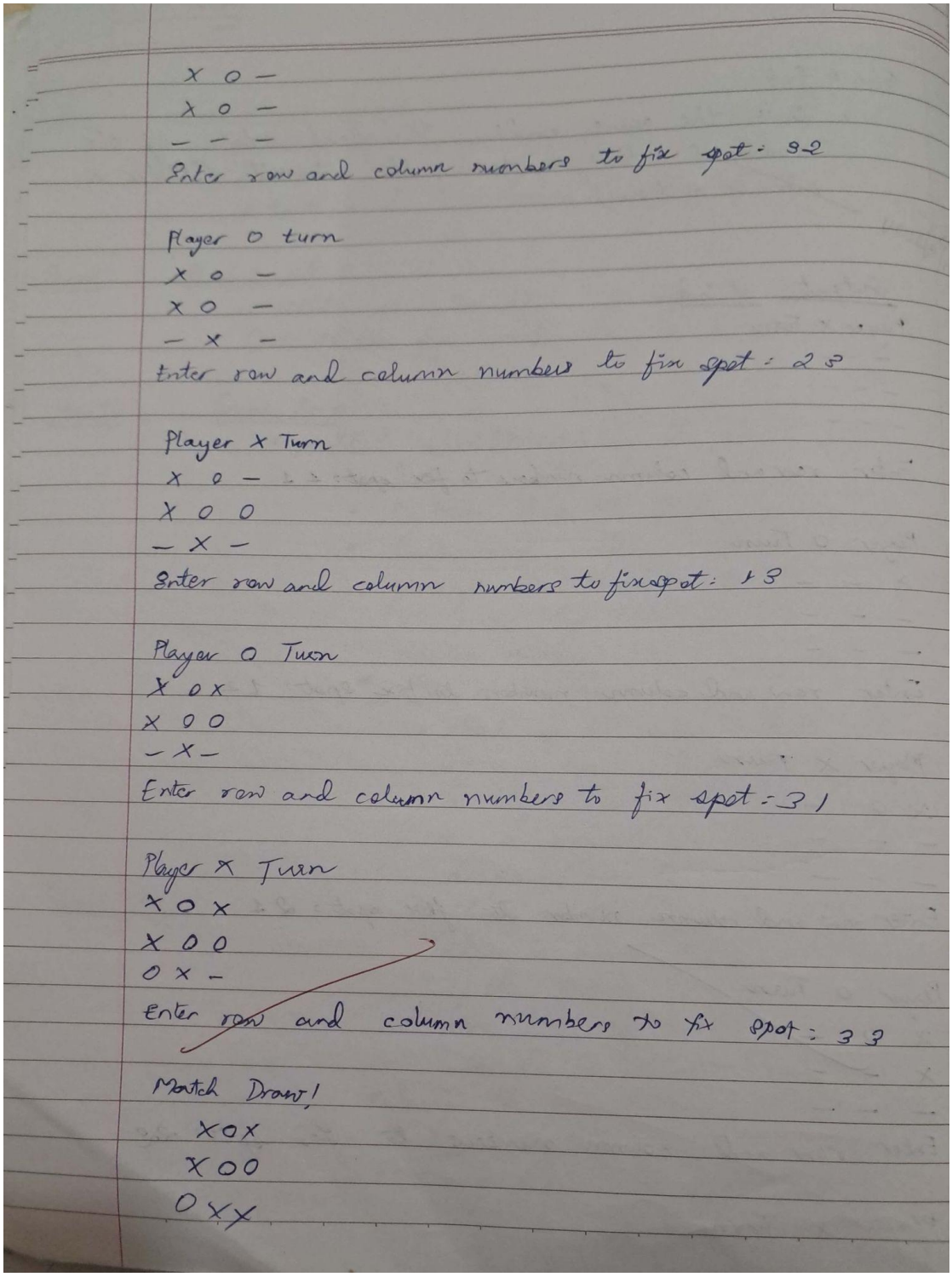
X O -

X - -

- - -

Enter row and column numbers to fix spot: 2 2

Player X Turn



X O –
X O –
– – –
Enter row and column numbers to fix spot: 0 2

Player O turn
X O –
X O –
– X –
Enter row and column numbers to fix spot: 2 0

Player X Turn
X O –
X O O
– X –
Enter row and column numbers to fix spot: 1 0

Player O Turn
X O X
X O O
– X –
Enter row and column numbers to fix spot: 3 1

Player X Turn
X O X
X O O
O X –
Enter row and column numbers to fix spot: 3 3

Match Draw!
X O X
X O O
O X X

Code: import
random

```
class TicTacToe:

    def __init__(self):
        self.board = []

    def create_board(self):
        for i in range(3):
            row = [] for j in
            range(3):
row.append('-')
self.board.append(row)

    def get_random_first_player(self):
        return random.randint(0, 1)

    def fix_spot(self, row, col, player):
        self.board[row][col] = player

    def is_player_win(self, player):
        win = None

        n = len(self.board)

        # checking rows
        for i in range(n):
        win = True for j in
        range(n):
                if self.board[i][j] !=
                    player: win = False
                    break
            if win:
                return win

        # checking
        columns for i in
        range(n): win =
        True for j in
        range(n):
```

```
            if self.board[j][i] !=
                player: win = False
                break
        if win:
            return
            win
    # checking
    diagonals win =
    True for i in
    range(n):
        if self.board[i][i] !=
            player: win = False
            break
    if win:
        return win

    win = True for i
    in range(n):
        if self.board[i][n - 1 - i] != player:
            win = False
            break
    if win:
        return win
    return False

    for row in self.board:
        for item in row:
            if item == '-':
                return False
    return True

def is_board_filled(self):
    for row in self.board:
        for item in row:
            if item == '-':
                return False
    return True

def swap_player_turn(self, player):
    return 'X' if player == 'O' else 'O'

def show_board(self):
    for row in self.board:
```

```
            for item in row:
                print(item, end=" ")
            print()

    def start(self):
        self.create_board()

        player = 'X' if self.get_random_first_player() == 1 else 'O'
        while True:
            print(f"Player {player} turn")
            self.show_board()

            # taking user input
            row, col = list(
                map(int, input("Enter row and column numbers to fix spot: ").split()))
            print()

            # fixing the spot
            self.fix_spot(row - 1, col - 1, player)

            # checking whether current player is won or not
            if self.is_player_win(player):
                print(f"Player {player} wins the game!")
                break

            # checking whether the game is draw or
            not if self.is_board_filled(): print("Match
            Draw!") break

            # swapping the turn
            player = self.swap_player_turn(player)

        # showing the final view of board
        print()
        self.show_board()


# starting the game
tic_tac_toe = TicTacToe()
tic_tac_toe.start()
```

Output

Case 1:

```
Player X turn
- - -
```

- - -
- - -
Enter row and column numbers to fix spot: 1 1

Player O turn
X - -
- - -
- - -
Enter row and column numbers to fix spot: 1 2

Player X turn
X O -
- - -
- - -
Enter row and column numbers to fix spot: 2 2

Player O turn
X O -
- X -
- - -
Enter row and column numbers to fix spot: 1 3

Player X turn
X O O
- X -
- - -
Enter row and column numbers to fix spot: 3 3

Player X wins the game!

X O O
- X -
- - X
Case 2:
Player O turn
- - -
- - -
- - -
Enter row and column numbers to fix spot: 1 1

Player X turn
O - -
- - -

- - -
Enter row and column numbers to fix spot: 2 1

Player O turn
O - -
X - -
- - -
Enter row and column numbers to fix spot: 1 2

Player X turn
O O -
X - -
- - -
Enter row and column numbers to fix spot: 3 3

Player O turn
O O -
X - -
- - X
Enter row and column numbers to fix spot: 2 3

Player X turn
O O -
X - O
- - X
Enter row and column numbers to fix spot: 1 3

Player O turn
O O X
X - O
- - X
Enter row and column numbers to fix spot: 2 2

Player X turn
O O X
X O O
- - X
Enter row and column numbers to fix spot: 3 2

Player O turn
O O X
X O O
- X X

Enter row and column numbers to fix spot: 3 1

Match Draw!

O O X
X O O
O X X
Case 3:
Player O turn
- - -
- - -
- - -
Enter row and column numbers to fix spot: 1 1

Player X turn
O - -
- - -
- - -
Enter row and column numbers to fix spot: 1 2 Player O turn
O X -
- - -
- - -
Enter row and column numbers to fix spot: 2 2

Player X turn
O X -
- O -
- - -
Enter row and column numbers to fix spot: 1 3

Player O turn
O X X
- O -
- - -
Enter row and column numbers to fix spot: 3 3

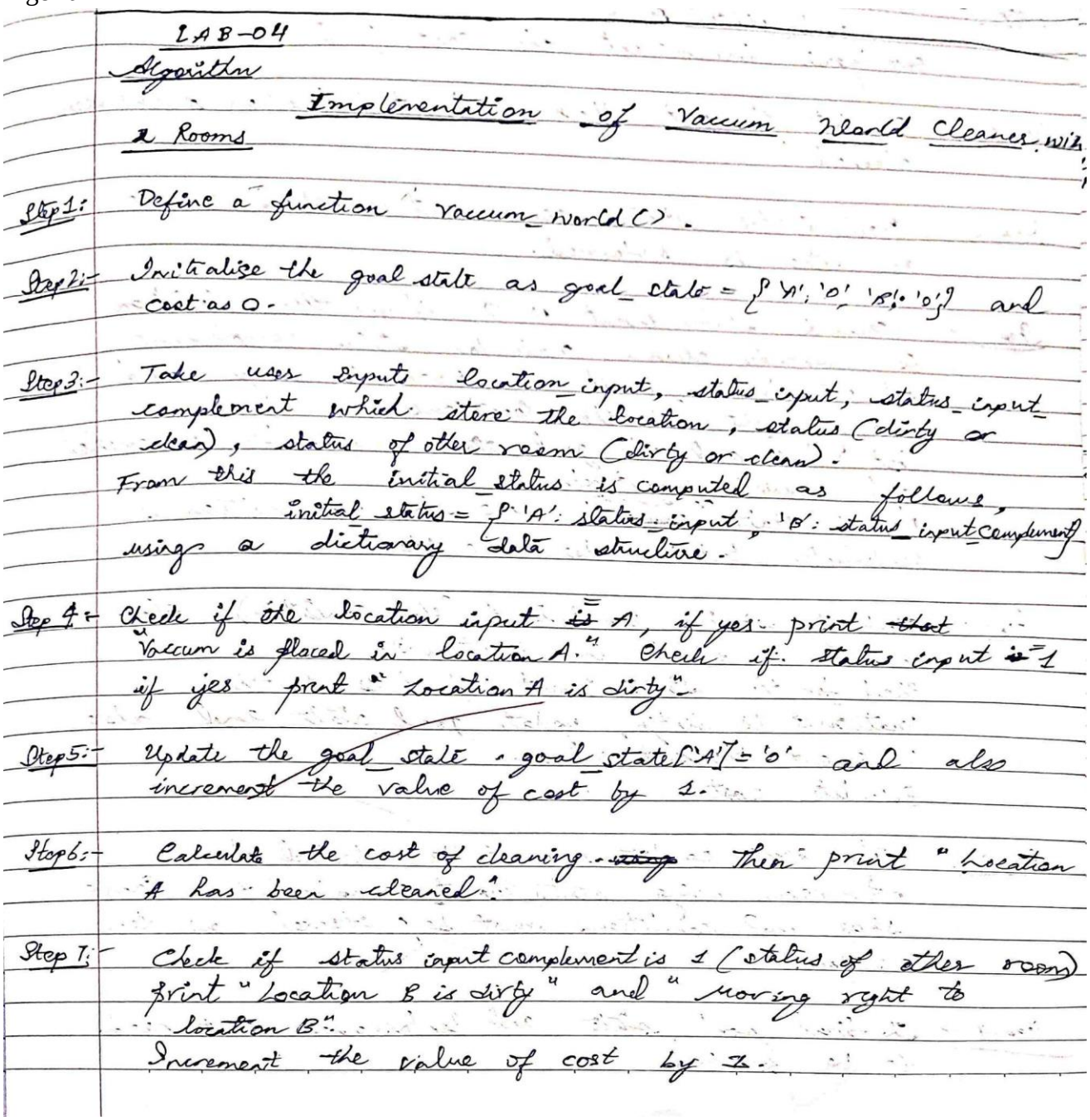
Player O wins the game!

O X X
- O -
- - O

Program1

Implement vacuum cleaner agent

Algorithm



LAB-04

Algorithm

Implementation of Vacuum World Cleaner with 2 Rooms

Step 1: Define a function vacuum_world().

Step 2: Initialise the goal state as goal_state = {'A': '0', 'B': '0'} and cost as 0.

Step 3: Take user inputs location_input, status_input, status_input_complement which store the location, status (dirty or clean), status of other room (dirty or clean).
From this the initial_status is computed as follows,
initial_status = {'A': status_input, 'B': status_input_complement}
using a dictionary data structure.

Step 4: Check if the location input == A, if yes print "Vacuum is placed in location A." Check if status input == 1 if yes print "Location A is dirty".

Step 5: Update the goal state, goal_state['A'] = '0' and also increment the value of cost by 1.

Step 6: Calculate the cost of cleaning then print "Location A has been cleaned".

Step 7: Check if status input complement is 1 (status of other room) print "Location B is dirty" and "moving right to location B".
Increment the value of cost by 1.

Step 8 :- Update the goal state
goal_state['B'] = '0'
Once again increment the cost by 1

Step 9 :- Calculate the cost then, print "location B has been cleaned".

Step 10 :- Check if the status complement is not 1, if not no action is performed.

Step 11 :- Check if status input is 0, if yes then location is already clean. Check for status complement of other room, if it is 1 room is dirty and moving right to the locations.
Cost value and the goal state must be incremented and updated respectively.
Then if status complement is not 0 then no action is performed. Cost is also calculated.

Step 12 :- Check if status input is 1 (else condition), then vacuum is placed in location B.
Check for the status input, if it is 1 then locationB is dirty, update goal state and also increment the cost value.
Calculate cost.

Step 13 :- The same conditions must be checked for other room using status input complement. Same steps and conditions must be checked as seen before.

Step 14 :- The final goal state and performance cost needs to be printed.

Code

```
import random
```

```
def display(room):
    for row in room:
        print(" ".join(str(cell) for cell in row))

# Function to create the room based on user input
def create_room():
    room = []
    print("Please enter the dimensions of the room (rows and columns):")
    rows = int(input("Number of rows: "))
    cols = int(input("Number of columns: "))

    for r in range(rows):
        row = []
        print(f"Enter the dirtiness for row {r + 1} (0 for clean, 1 for dirty):")
        for c in range(cols):
            while True:
                dirt = int(input(f"Cell ({r + 1}, {c + 1}): "))
                if dirt in [0, 1]:
                    row.append(dirt)
                    break
                else:
                    print("Invalid input. Please enter 0 or 1.")
        room.append(row)
    return room

# Create the room based on user input
room = create_room()
print("All the rooms are dirty")
display(room)
```

```
# Variables to track cleaning
x = 0

y = 0
z = 0

print("Before cleaning the room I detect all of these random dirts")
display(room)

while x < len(room):
    while y < len(room[x]):
        if room[x][y] == 1:
            print("Vacuum in this location now,", x, y)
            room[x][y] = 0 # Clean the room
            print("Cleaned", x, y)
            z += 1
        y += 1
    x += 1
    y = 0

pro = (100 - ((z / (len(room) * len(room[0]))) * 100))
print("Room is clean now, Thanks for using: 3710933")
display(room)
print('Performance =', pro, '%')
```

Output:

```
Enter Location of VacuumB

Enter status of B1

Enter status of other room1
```

Initial Location Condition{'A': '1', 'B': '1'}

Vacuum is placed in location B

Location B is Dirty.

COST for CLEANING 1

Location B has been Cleaned.

Location A is Dirty.

Moving LEFT to the Location A.

COST for SUCK 3

Location A has been Cleaned.

GOAL
STATE:

{'A': '0', 'B': '0'}

Performance Measurement: 3

Program2

Implement 8 puzzle problems using Depth First Search (DFS)

Algorithm:

8/10/2024 LAB-02 Tuesday

Algorithm

1) Implementation of 8 puzzle problem using DFS.

Step 1:- Import packages sys and numpy.

Step 2:- Define a class called Node. Use the init method with the parameters self, state, parent and action

Step 3:- Define a class called Stack frontier Use the init method with parameters self. Similarly create another function called add with self and node as parameter. Create functions contain_state, remove, empty to perform the appropriate operations

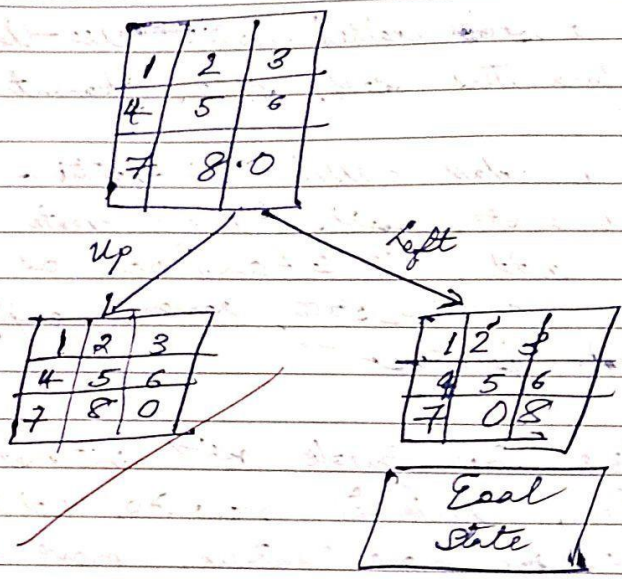
Step 4:- Create a class puzzle with init method, neighbours method, Check for row >0, col >0, row <2 and col <2 for move blank up, move blank down, move blank left and move blank right. Create method called print solution which is used to print solution that contains start state, goal state, states explored.

Step5:- Similarly create another function does not contain state to check if the state is explored

Step6:- Define another function called solve with the ~~parent~~ parameter self. The explored states is set to 0, start state is calculated. All the actions like up, down, left are performed and then solution is obtained. A function is used to get user input. An instance of puzzle is created and solved.

Step7:- A try & block exception is written to handle errors.

State Space Tree



Code:

```
from copy import deepcopy

DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]

class PuzzleState:
    def __init__(self, board, parent=None, move=""):
        self.board = board
        self.parent = parent
        self.move = move

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

    def generate_successors(self):
        successors = []
        x, y = self.get_blank_position()

        for dx, dy in DIRECTIONS:
            new_x, new_y = x + dx, y + dy

            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = deepcopy(self.board)
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
                successors.append(PuzzleState(new_board, parent=self))
```

```
        return successors

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

    def __str__(self):
        return "\n".join([" ".join(map(str, row)) for row in self.board])

def depth_limited_search(current_state, goal_state, depth):
    if depth == 0 and current_state.is_goal(goal_state):
        return current_state

    if depth > 0:
        for successor in current_state.generate_successors():
            found = depth_limited_search(successor, goal_state, depth - 1)
            if found:
                return found
    return None

def iterative_deepening_search(start_state, goal_state, max_depth):
    for depth in range(max_depth + 1):
        print(f"\nSearching at depth level: {depth}")
        result = depth_limited_search(start_state, goal_state, depth)
        if result:
            return result
    return None

def get_user_input():
    print("Enter the start state (use 0 for the blank):")
```

```
    start_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        start_state.append(row)

    print("Enter the goal state (use 0 for the blank):")
    goal_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        goal_state.append(row)
```

```
    max_depth = int(input("Enter the maximum depth for search: "))

    return start_state, goal_state, max_depth

def main():
    start_board, goal_board, max_depth = get_user_input()
    start_state = PuzzleState(start_board)
    goal_state = goal_board

    result = iterative_deepening_search(start_state, goal_state, max_depth)

    if result:
        print("\nGoal reached!")
        path = []
        while result:
            path.append(result)
            result = result.parent
        path.reverse()
```

```
        for state in path:
            print(state, "\n")
    else:
        print("Goal state not found within the specified depth.")


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

Output:

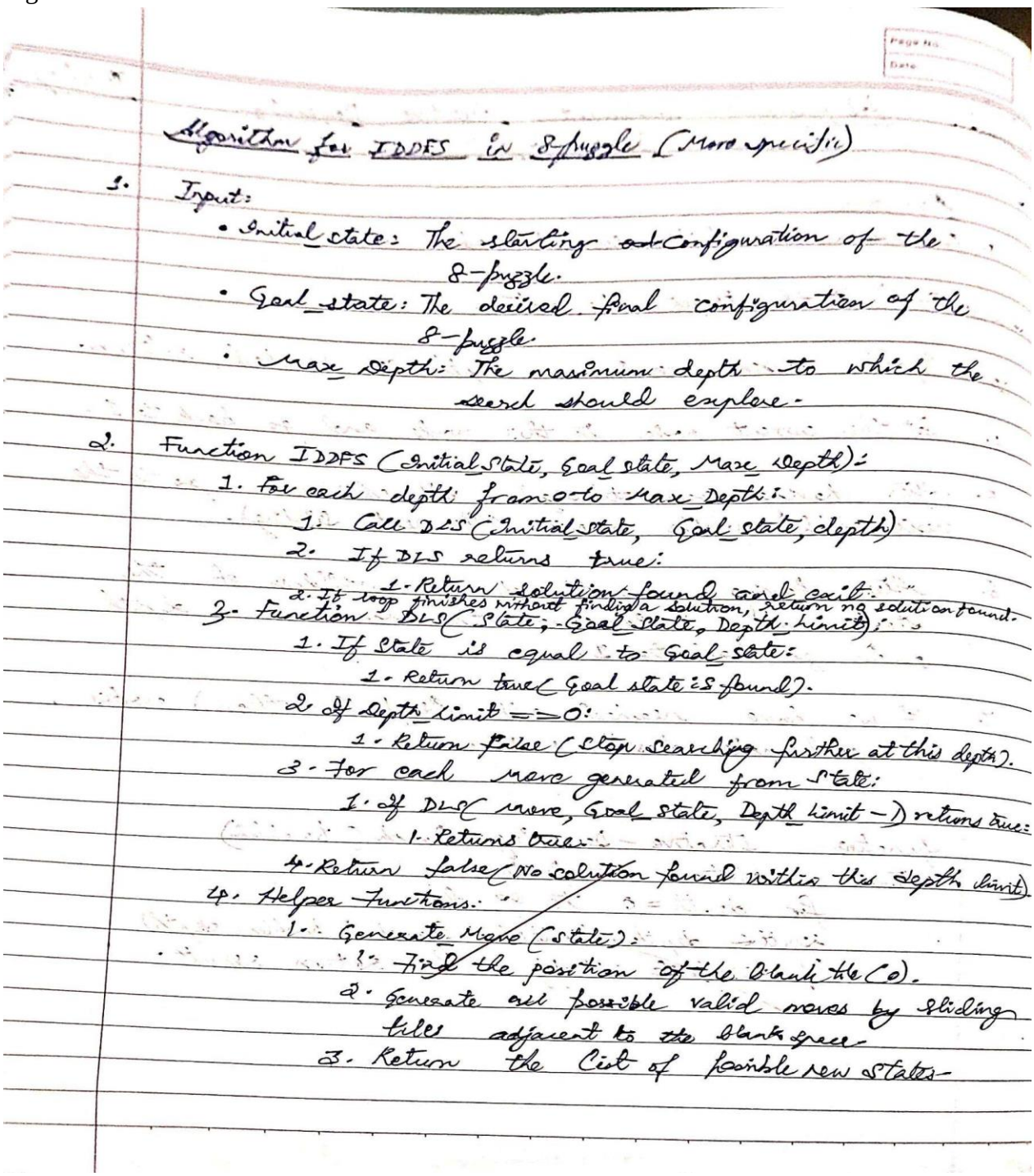
```
Enter the start state (use 0 for the blank):

1 2 3

4 5 6

 7 0 8

Enter the goal state (use 0 for the blank):

1 2 3

4 5 6

7 8 0

Enter the maximum depth for search: 5


Searching at depth level: 0
```

```
Searching at depth level: 1



Goal reached!

1 2 3

4 5 6

7 0 8



1 2 3

4 5 6

7 8 0
```

Program2

Implement iterative deepening search algorithm.

Algorithm:



Algorithm for IDDFS in 8-puzzle (More specific)

1. Input:
   - Initial_state: The starting configuration of the 8-puzzle.
   - Goal_state: The desired final configuration of the 8-puzzle.
   - max_depth: The maximum depth to which the search should explore.
2. Function IDDFS (Initial_State, Goal_state, Max_depth):
   1. For each depth from 0 to max_Depth:
      1. Call DLS (Initial_state, Goal_state, depth)
      2. If DLS returns true:
         1. Return solution found and exit.
   2. If loop finishes without finding a solution, return no solution found.
3. Function DLS (State, Goal_State, Depth_limit):
   1. If State is equal to Goal_state:
      1. Return true (Goal state is found).
   2. If Depth_limit == 0:
      1. Return false (stop searching further at this depth).
   3. For each move generated from State:
      1. If DLS (move, Goal_state, Depth_limit − 1) returns true:
         1. Returns true.
   4. Return false (No solution found within this depth limit)
4. Helper Functions:
   1. Generate_Moves (State):
      1. Find the position of the blank tile (0).
      2. Generate all possible valid moves by sliding tiles adjacent to the blank space.
      3. Return the list of possible new states.

Code:

```
from copy import deepcopy
```

```
DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]


class PuzzleState:
    def __init__(self, board, parent=None, move=""):
        self.board = board
        self.parent = parent
        self.move = move

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

    def generate_successors(self):
        successors = []
        x, y = self.get_blank_position()

        for dx, dy in DIRECTIONS:
            new_x, new_y = x + dx, y + dy

            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = deepcopy(self.board)
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
                successors.append(PuzzleState(new_board, parent=self))

        return successors
```

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

    def __str__(self):
        return "\n".join([" ".join(map(str, row)) for row in self.board])

def depth_limited_search(current_state, goal_state, depth):
    if depth == 0 and current_state.is_goal(goal_state):
        return current_state

    if depth > 0:
        for successor in current_state.generate_successors():
            found = depth_limited_search(successor, goal_state, depth - 1)
            if found:
                return found
    return None

def iterative_deepening_search(start_state, goal_state, max_depth):
    for depth in range(max_depth + 1):
        print(f"\nSearching at depth level: {depth}")
        result = depth_limited_search(start_state, goal_state, depth)
        if result:
            return result
    return None

def get_user_input():
    print("Enter the start state (use 0 for the blank):")
    start_state = []
    for _ in range(3):
```

```
        row = list(map(int, input().split()))
        start_state.append(row)

    print("Enter the goal state (use 0 for the blank):")
    goal_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        goal_state.append(row)

    max_depth = int(input("Enter the maximum depth for search: "))

    return start_state, goal_state, max_depth

def main():
    start_board, goal_board, max_depth = get_user_input()
    start_state = PuzzleState(start_board)
    goal_state = goal_board

    result = iterative_deepening_search(start_state, goal_state, max_depth)

    if result:
        print("\nGoal reached!")
        path = []
        while result:
            path.append(result)
            result = result.parent
        path.reverse()
        for state in path:
            print(state, "\n")
    else:
```

```
    print("Goal state not found within the specified depth.")


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

38s

```
    main()
from copy import deepcopy


DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]

class PuzzleState:
```

```
    def __init__(self, board, parent=None, move=""):
        self.board = board


        self.parent = parent

        self.move = move

    def get_blank_position(self):

        for i in range(3):
            for j in range(3):

                if self.board[i][j] == 0:


                    return i, j

    def generate_successors(self):

        successors = []


        x, y = self.get_blank_position()
        for dx, dy in DIRECTIONS:
```

```
            new_x, new_y = x + dx, y + dy

            if 0 <= new_x < 3 and 0 <= new_y < 3:

                new_board = deepcopy(self.board)

                new_board[x][y], new_board[new_x][new_y] =
new_board[new_x][new_y], new_board[x][y]

                successors.append(PuzzleState(new_board, parent=self))

        return successors
    def is_goal(self, goal_state):

        return self.board == goal_state


    def __str__(self):

        return "\n".join([" ".join(map(str, row)) for row in self.board])

def depth_limited_search(current_state, goal_state, depth):

    if depth == 0 and current_state.is_goal(goal_state):

        return current_state

    if depth > 0:
```

```
        for successor in current_state.generate_successors():

            found = depth_limited_search(successor, goal_state, depth - 1)

            if found:

                return found


    return None def iterative_deepening_search(start_state,
goal_state, max_depth):
    for depth in range(max_depth + 1):
```

```
        print(f"\nSearching at depth level: {depth}")

        result = depth_limited_search(start_state, goal_state, depth)

        if result:

            return result


    return None def
get_user_input():
    print("Enter the start state (use 0 for the blank):")

    start_state = []


    for _ in range(3):

        row = list(map(int, input().split()))

        start_state.append(row)


    print("Enter the goal state (use 0 for the blank):")
    goal_state = []
    for _ in range(3):

        row = list(map(int, input().split()))


        goal_state.append(row)

    max_depth = int(input("Enter the maximum depth for search: "))

    return start_state, goal_state, max_depth

def main():

    start_board, goal_board, max_depth = get_user_input()
```

```
    start_state = PuzzleState(start_board)
    goal_state = goal_board
    result = iterative_deepening_search(start_state, goal_state, max_depth)
    if result:
        print("\nGoal reached!")
        path = []
        while result:
            path.append(result)
            result = result.parent
        path.reverse()
        for state in path:
            print(state, "\n")
    else:
        print("Goal state not found within the specified depth.")
if __name__ == "__main__":
```

Output

```
Enter the start state (use 0 for the blank):
2 8 3
1 6 4
7 0 5
Enter the goal state (use 0 for the blank):
1 2 3
8 0 4
7 6 5
```

Enter the maximum depth for search: 5

Searching at depth level: 0

Searching at depth level: 1

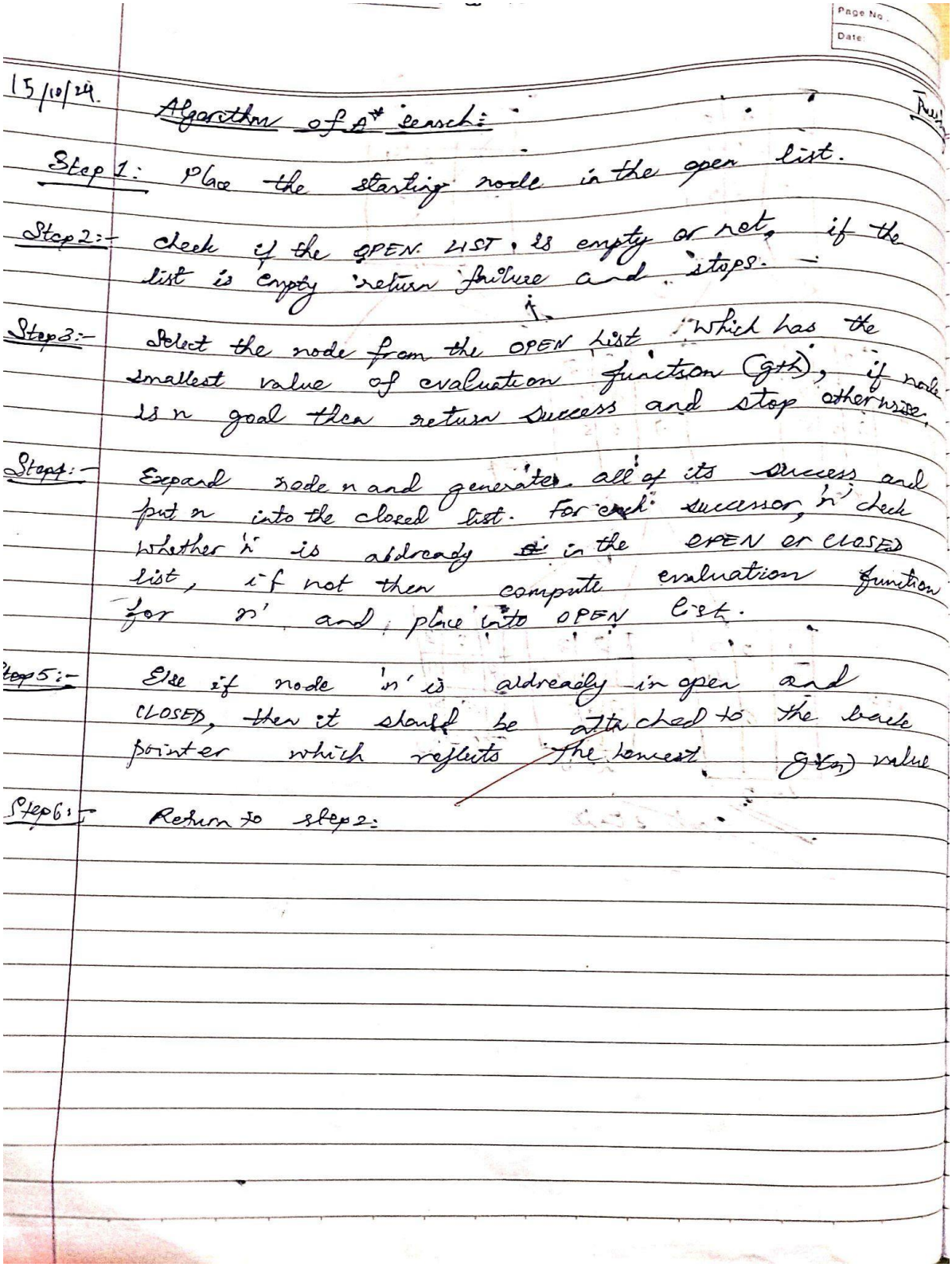
Searching at depth level: 2

```
Searching at depth level: 3

Searching at depth level: 4

Searching at depth level: 5

Goal reached!
2 8 3
1 6 4
7 0 5

2 8 3
1 0 4
7 6 5

2 0 3
1 8 4
7 6 5

0 2 3
1 8 4
7 6 5

1 2 3
0 8 4
7 6 5

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

Program3

Implement A* search algorithm

Algorithm:



15/10/24.

Algorithm of A* search:

Step 1: Place the starting node in the open list.

Step 2:- Check if the OPEN LIST is empty or not, if the list is empty return failure and stops.

Step 3:- Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node is n goal then return success and stop otherwise.

Step 4:- Expand node n and generate all of its success and put n into the closed list. For each successor, n' check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into OPEN list.

Step 5:- Else if node n' is already in open and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n) value

Step 6:- Return to step 2:

15/9/23

Page No.:
Date:

Tuesday

Algorithm for 8 puzzle problem using A* approach. Heuristic (misplaced tiles).

Step 1: Define the initial state and the final state as per the question.

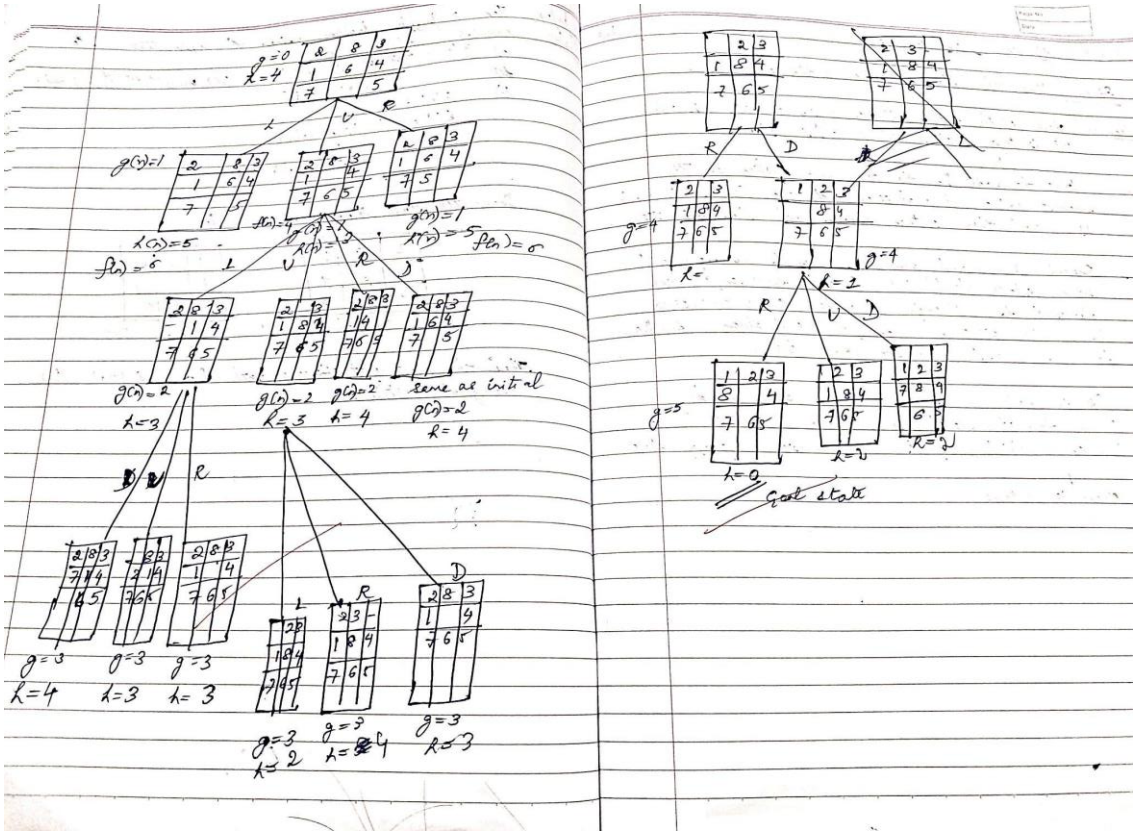
Step 2: The value of g is 0 in the first step. There are movements of blank space in right, left, up and down positions.

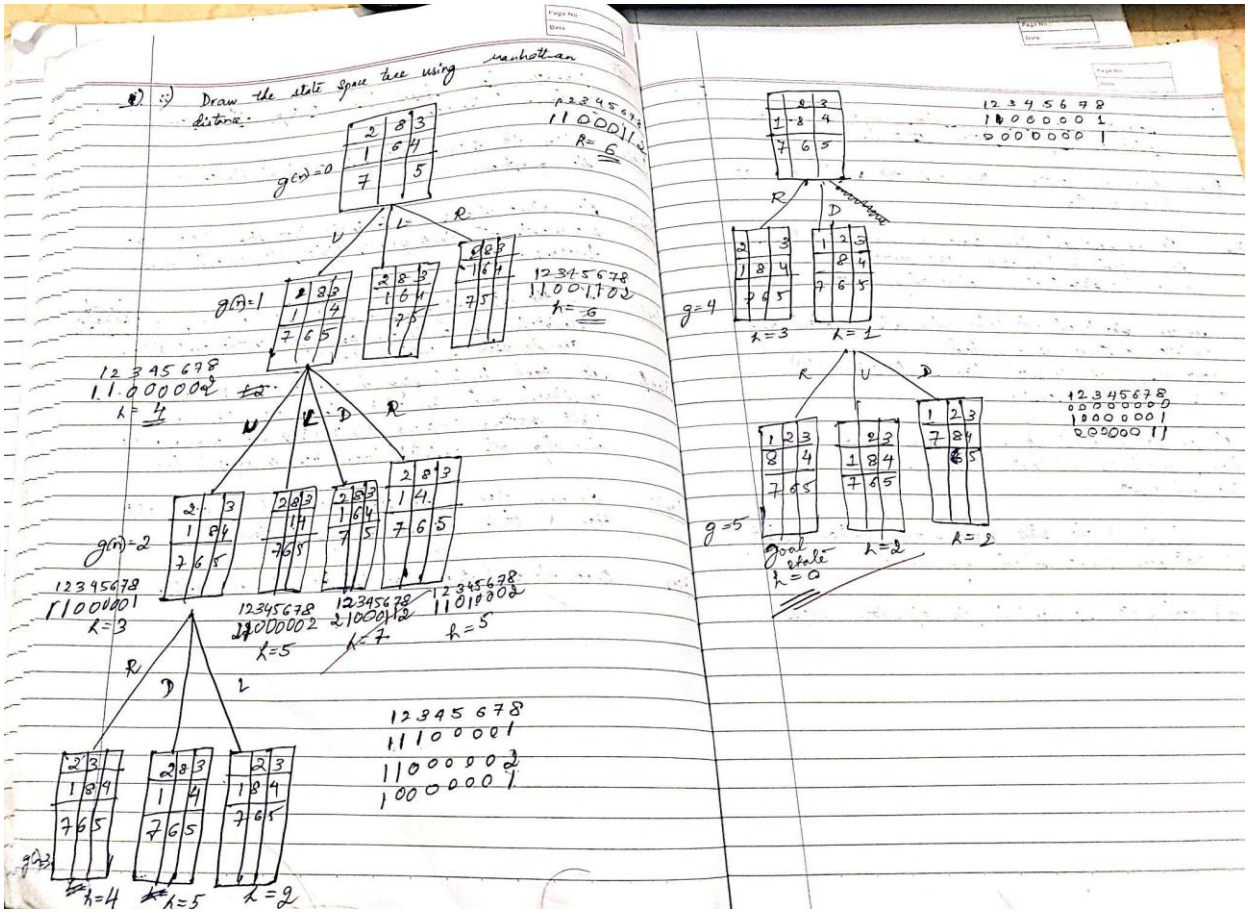
Step 3: Then the value of g gets updated to 1. The value of h is the (number of misplaced tiles) calculated by comparing each state to the goal state. Formula used is $f(n) = g(n) + h(n)$.

Step 4: At each level (using the g value), the smallest value is considered and for that state, the up, down, left, right positions (moving blank state) are drawn. The value of g gets updated here. Step 3 is repeated here.

Step 5: In any step, if the state matches the initial state then it is not solved any further.

Step 6: Once the goal state is reached, the algorithm ends.


Same as initial
Goal state


2) i) Draw the state space tree using manhattan distance.
g(n)=0
g(n)=1
g(n)=2
g=4
g=5
Goal state
h=0

15/10/24

Tuesday

Algorithm for 8 puzzle problem using A* search technique using Manhattan Distance.

Step 1:- Define the initial state and the final state as for the question.

Step 2:- The movement of blank space in upper, right, left, down lead to presence of new states.
In the first state level, the value of g (depth) is 0.

Step 3:- At each state level, the value of g gets updated and manhatan distance is computed.
After this, f is computed which is: $f(n) = g(n) + h(n)$
This is basically the cost.

Step 4:- The lowest value of f is considered to be solved further, considering all possible positions (up, down, left and right).

Step 5:- In any level, if the state matches the initial state then that step is not solved any further.

Step 6:- Once the goal state is reached, the algorithm stops there.

Code:

```
import heapq


# Function to check if the puzzle is in the goal state
def is_goal(state, goal_state):
    return state == goal_state


# Function to count the number of misplaced tiles
def misplaced_tiles(state, goal_state):
    count = 0
    for i in range(len(state)):
        if state[i] != goal_state[i] and state[i] != 0: # Exclude the blank tile (0)
            count += 1
    return count


# Function to find possible moves (successors) from the current state
def get_successors(state):
    successors = []
    blank_idx = state.index(0) # Find the index of the blank (0)

    # Possible moves: up, down, left, right
    moves = {
        "up": -3, "down": 3, "left": -1, "right": 1
    }

    for direction, move in moves.items():
        new_idx = blank_idx + move
        if 0 <= new_idx < 9: # Check if the move is within bounds
            if direction == "left" and blank_idx % 3 == 0:
                continue # Skip invalid move to the left
```

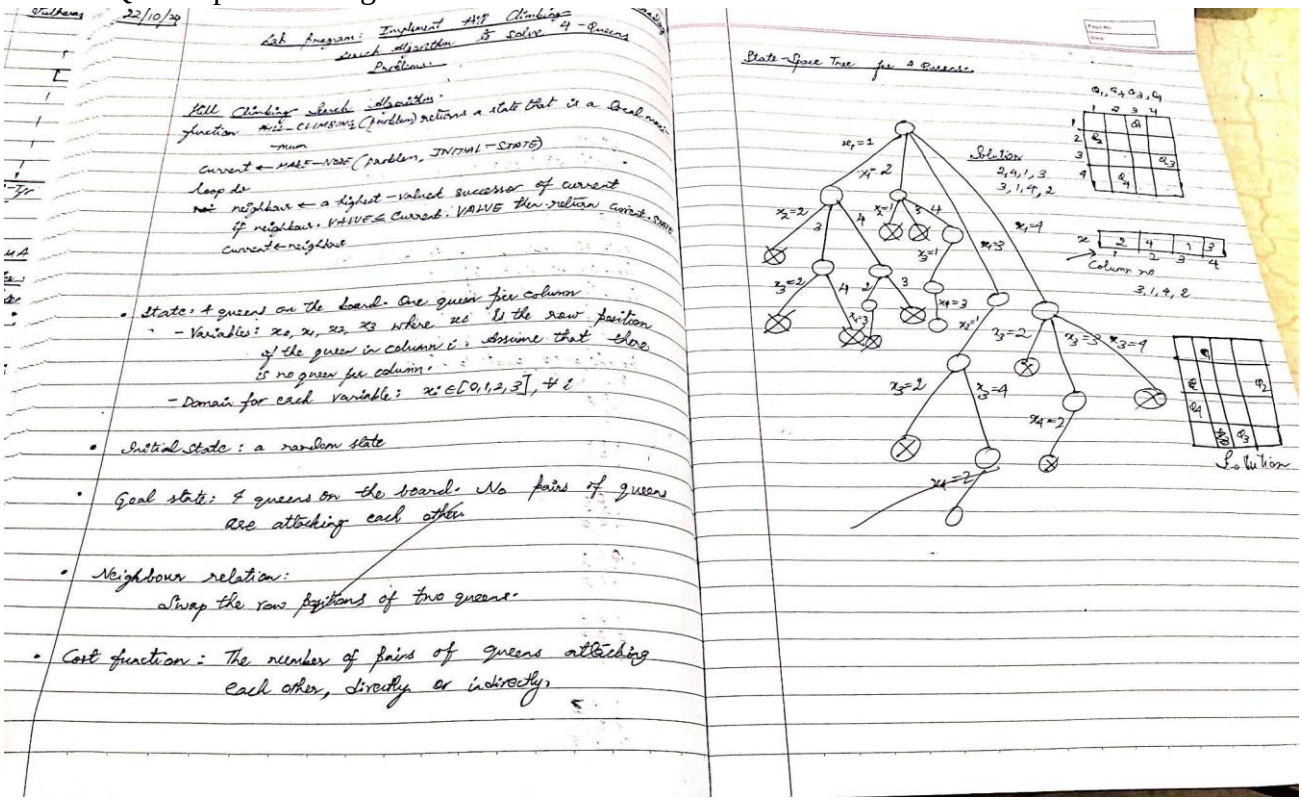
```
            if direction == "right" and blank_idx % 3 == 2:
                continue # Skip invalid move to the right
            new_state = state[:]
            new_state[blank_idx], new_state[new_idx] = new_state[new_idx], new_state[blank_idx]
            successors.append(new_state)

    return successors

# A* Algorithm using Misplaced Tiles heuristic
def astar_misplaced_tiles(start_state, goal_state):
    open_list = []
    closed_list = set()

    # Push the initial state into the priority queue (heap), with f = g + h
    heapq.heappush(open_list, (misplaced_tiles(start_state, goal_state), 0, start_state, []))

    while open_list:
        f, g, current_state, path = heapq.heappop(open_list)

        if is_goal(current_state, goal_state):
            return path + [current_state] # Return the path when goal is reached

        if tuple(current_state) in closed_list:
            continue

        closed_list.add(tuple(current_state))

        # Expand the current node (find successors)
        level_states = []
```

```
        for successor in get_successors(current_state):
            if tuple(successor) not in closed_list:
                new_g = g + 1 # Increment the cost to reach the successor
                new_f = new_g + misplaced_tiles(successor, goal_state)
                heapq.heappush(open_list, (new_f, new_g, successor, path + [current_state]))
                level_states.append(successor)

        if level_states:
            print(f"\nLevel {g+1}:")
            display_states_side_by_side(level_states)

    return None # No solution found

# Function to display the 8-puzzle in a readable format, multiple states side by side
def display_states_side_by_side(states):
    lines = [""] * 3 # Each puzzle has 3 lines

    for state in states:
        for i in range(0, 9, 3):
            lines[i // 3] += f"{state[i:i+3]}    "

    for line in lines:
        print(line)

# Main function to take input and run the A* algorithm
def main():
    print("Enter the start state of the 8-puzzle (9 numbers, use 0 for the blank tile):")
    start_state = list(map(int, input().split()))
```

```
    print("Enter the goal state of the 8-puzzle (9 numbers, use 0 for the blank tile):")
    goal_state = list(map(int, input().split()))

    print("\nSolving the 8-puzzle...\n")

    solution = astar_misplaced_tiles(start_state, goal_state)

    if solution:
        print("\nSolution Path Found!")
        for step, state in enumerate(solution):
            print(f"Step {step}:")
            display_states_side_by_side([state])
    else:
        print("No solution found.")

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

Program4

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



Code:

N = 4

```
def print_board(solution):
    for row in solution:
        print(" ".join("Q" if col else "." for col in row))
    print()

def is_safe(board, row, col): # Check
    this column on upper side for i in
    range(row):
        if board[i][col] == 1:
            return False

    # Check upper diagonal on left side for i, j in
    zip(range(row, -1, -1), range(col, -1, -1)): if j < 0:
            break
        if board[i][j] == 1:
            return False

    # Check upper diagonal on right side for i, j in
    zip(range(row, -1, -1), range(col, N)): if j >=
    N: break
        if board[i][j] == 1:
            return False

    return True

def solve_n_queens(board, row, solutions):
    if row == N:
        solutions.append([row[:] for row in board])
        return

    for col in range(N):
        if is_safe(board, row, col):
            board[row][col] = 1 # Place queen
            solve_n_queens(board, row + 1, solutions) # Recur to place rest
            board[row][col] = 0 # Backtrack

def main(): board = [[0 for _ in range(N)] for _ in
    range(N)] solutions = [] solve_n_queens(board,
    0, solutions)

    print(f"Found {len(solutions)} solutions:")
    for solution in solutions:
        print_board(solution)
```

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

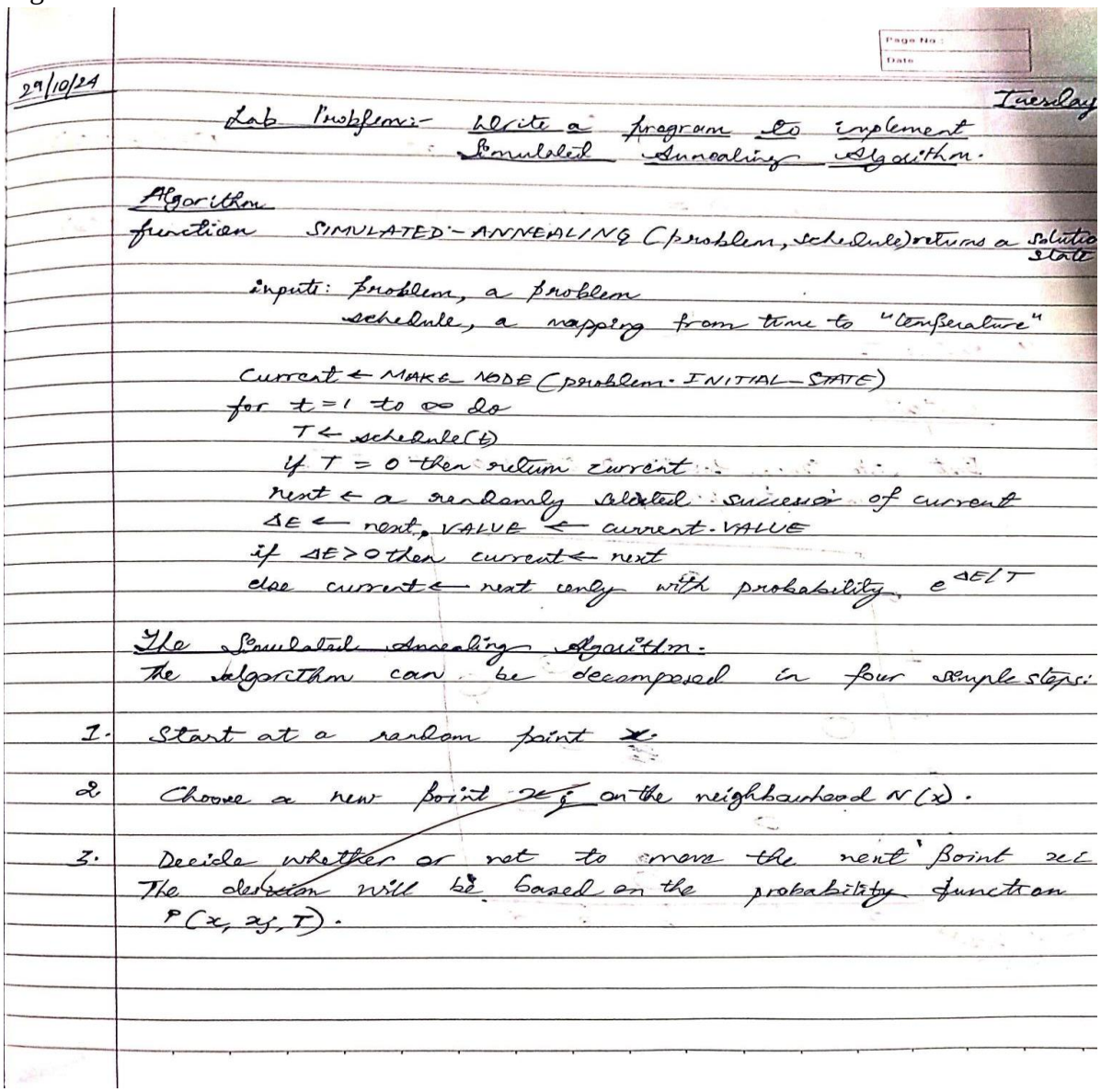
Output:

```
Found 2 solutions:
. Q . .
. . . Q
Q . . .
. . Q .

. . Q .
Q . . .
. . . Q
. Q . .
```

Program5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



29/10/24 Tuesday

Lab Problem:- Write a program to implement Simulated Annealing Algorithm.

Algorithm

function SIMULATED-ANNEALING(problem, schedule) returns a Solution state

inputs: problem, a problem
schedule, a mapping from time to "temperature"

current ← MAKE_NODE(problem.INITIAL_STATE)
for t = 1 to ∞ do
T ← schedule(t)
if T = 0 then return current
next ← a randomly selected successor of current
ΔE ← next.VALUE ← current.VALUE
if ΔE > 0 then current ← next
else current ← next only with probability $e^{\Delta E/T}$

The Simulated Annealing Algorithm:
The algorithm can be decomposed in four simple steps:

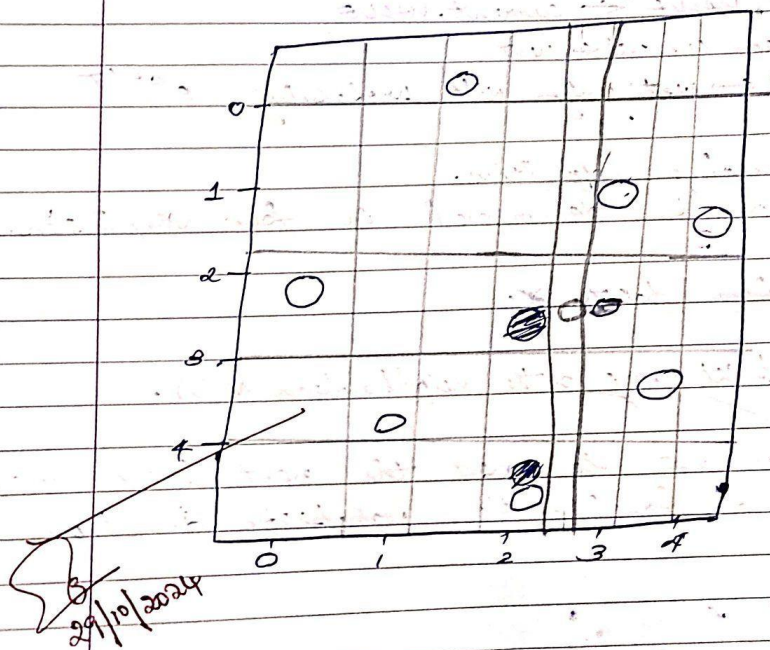
1. Start at a random point x.
2. Choose a new point xi on the neighbourhood N(x).
3. Decide whether or not to move the next point xi. The decision will be based on the probability function P(x, xi, T).

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

4. Reduce T.

Output

Best state (Queen positions): [3, 6, 0, 7, 4, 1, 5, 2],
Number of conflicts: 0



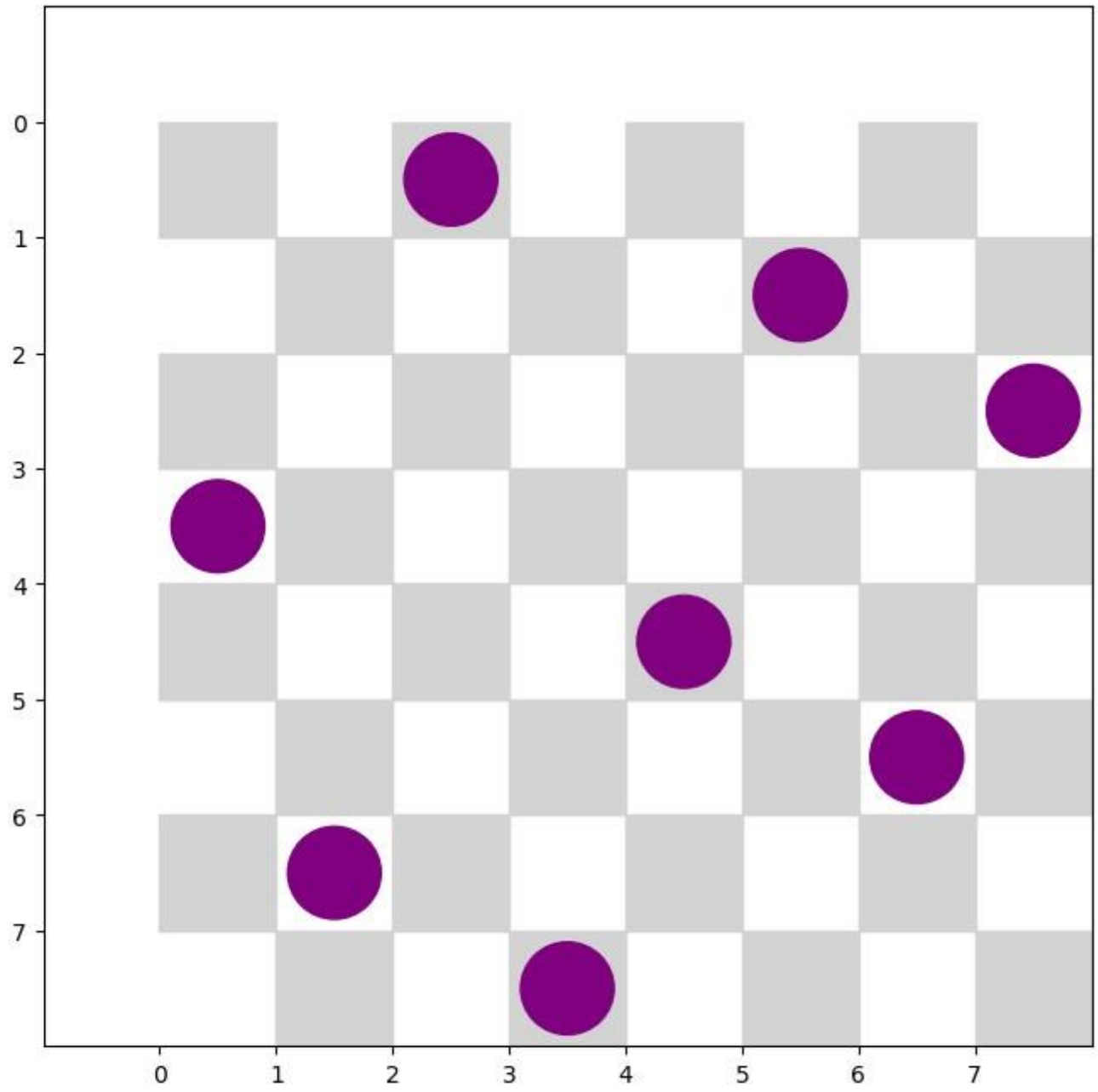
29/10/2024

Code:

```
import random
import math
import matplotlib.pyplot as plt

# Generate an initial solution with unique columns for each queen def
create_initial_solution(n): return random.sample(range(n), n) # A permutation of column
indices (unique columns)

# Calculate the number of diagonal conflicts
def calculate_fitness(state):
diagonal_conflicts = 0 n = len(state)

    for i in range(n):
        for j in range(i + 1, n):
            if abs(state[i] - state[j]) == abs(i - j): # Check if they are on the same diagonal
                diagonal_conflicts += 1

    return diagonal_conflicts

# Generate a neighboring solution by swapping two columns
def random_neighbor(state):
    neighbor = state[:] i, j = random.sample(range(len(state)), 2) # Pick two
    different rows to swap
    neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
    return neighbor

# Simulated Annealing Algorithm def simulated_annealing(n, initial_temp=1000,
cooling_rate=0.95, max_iterations=1000):
    current_solution = create_initial_solution(n)
    current_fitness =
    calculate_fitness(current_solution) best_solution =
    current_solution best_fitness = current_fitness
    temperature = initial_temp

    for iteration in range(max_iterations):
        neighbor = random_neighbor(current_solution)

        neighbor_fitness = calculate_fitness(neighbor)

        fitness_diff = neighbor_fitness - current_fitness

        # Accept the neighbor if it improves the solution or based on the annealing probability
        if fitness_diff < 0 or random.uniform(0, 1) < math.exp(-fitness_diff / temperature):
```

```
            current_solution = neighbor
            current_fitness = neighbor_fitness # Update the best solution if the current one is better if current_fitness < best_fitness:
            best_solution = current_solution
            best_fitness = current_fitness

        # Cool down the temperature

    temperature *= cooling_rate return

    best_solution, best_fitness

# Visualize the chessboard and queens def plot_solution(solution): n = len(solution) plt.figure(figsize=(n, n)) plt.xlim(-1, n) plt.ylim(-1, n)

    # Draw the chessboard
    for i in range(n):
        for j in range(n): if (i + j) % 2 == 0:
            plt.gca().add_patch(plt.Rectangle((j, i), 1, 1, color='lightgrey'))

    # Place the queens for col, row in enumerate(solution):
    plt.gca().add_patch(plt.Circle((col + 0.5, row + 0.5), 0.4, color='purple'))

    plt.xticks(range(n))
    plt.yticks(range(n))
    plt.gca().invert_yaxis()
    plt.grid(False)
    plt.show()

# Parameters
n = 8 # Number of queens
best_solution, best_fitness = simulated_annealing(n)

# Output results print(f"Best state (Queen positions): {best_solution}, Number of conflicts: {best_fitness}")

# Plot the solution
plot_solution(best_solution)
```

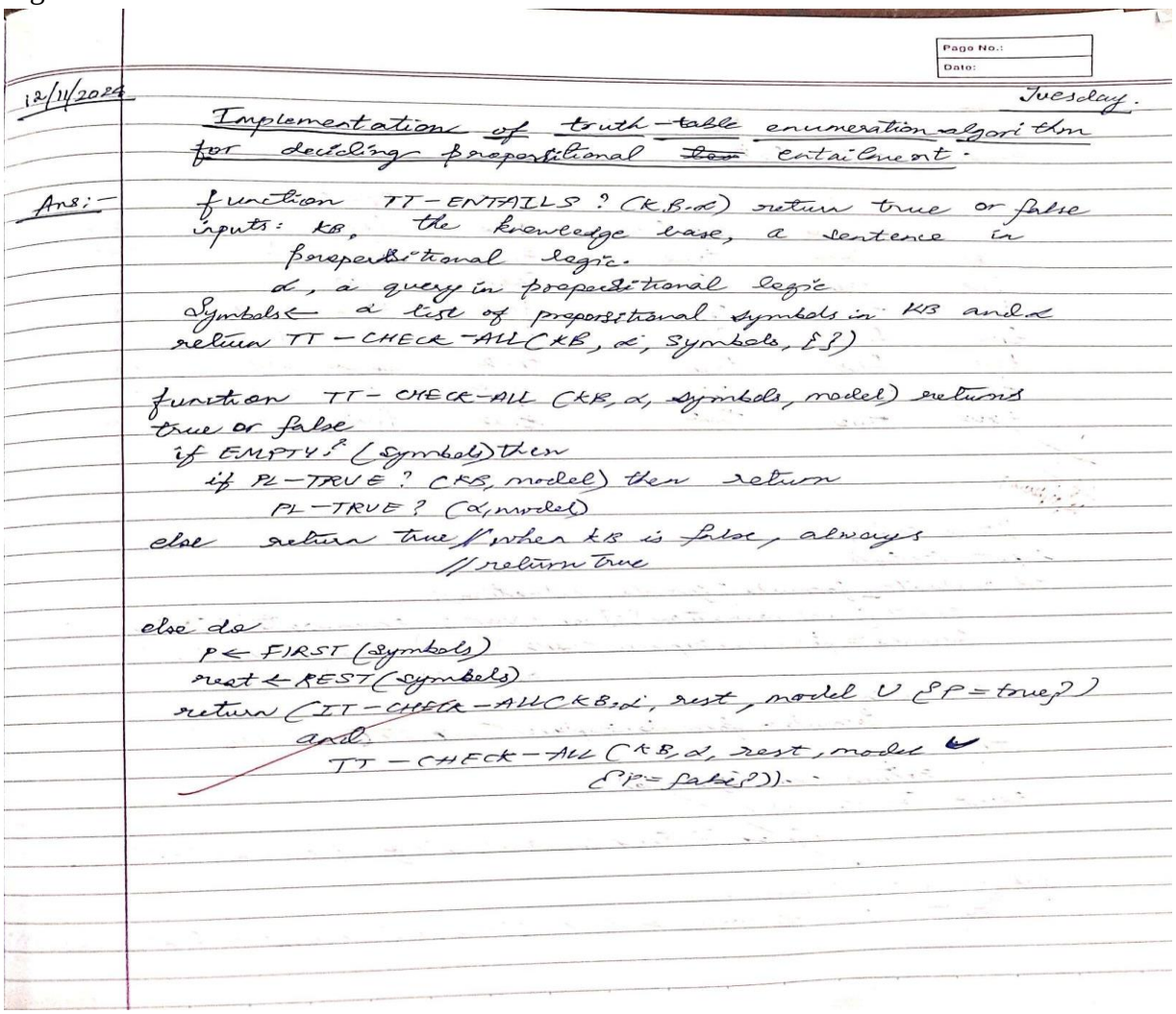
Output:
Best state (Queen positions): [3, 6, 0, 7, 4, 1, 5, 2], Number of conflicts: 0


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

Program5

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

Algorithm:



12/11/2024 Tuesday.

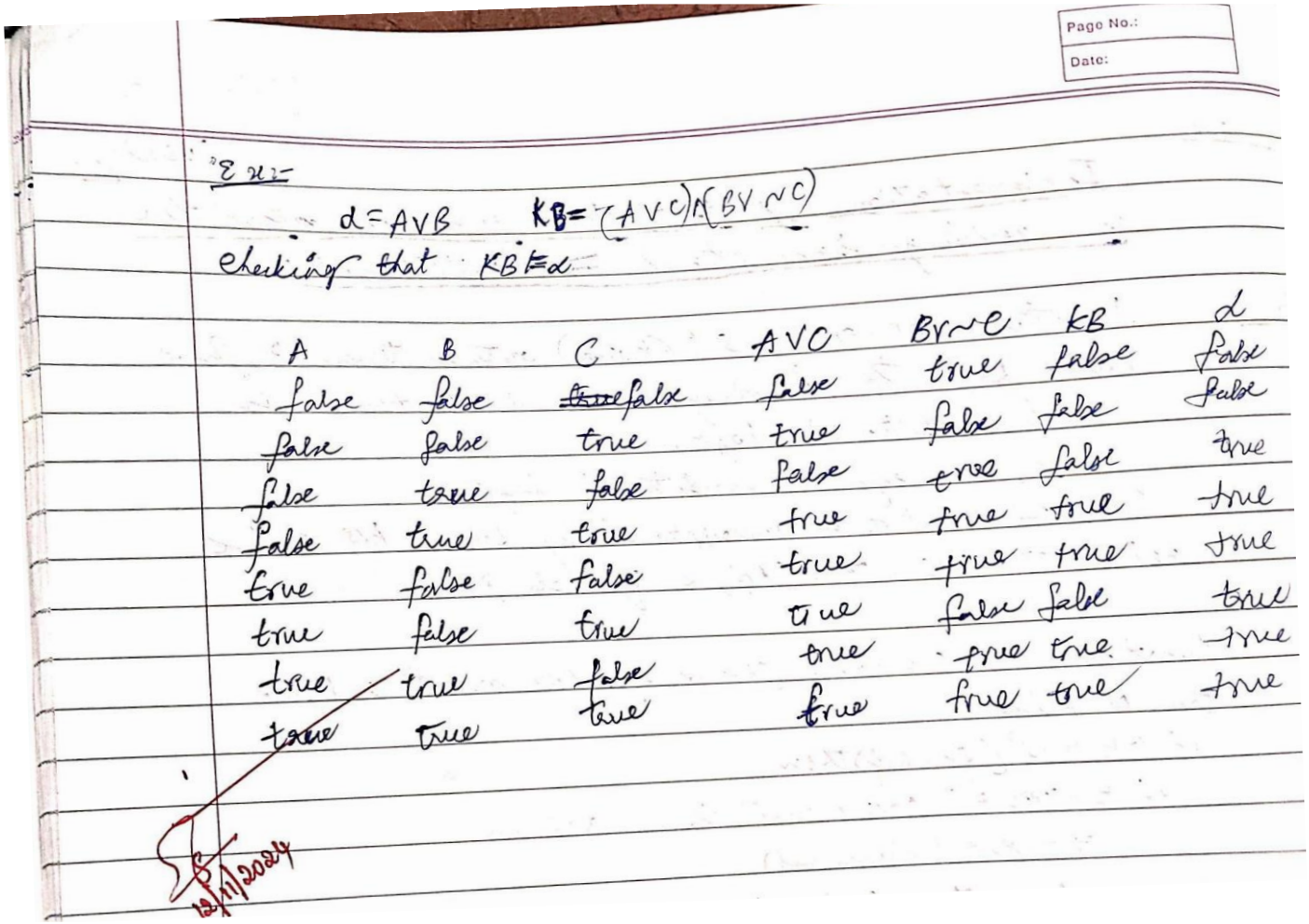
Implementation of truth-table enumeration algorithm for deciding propositional entailment.

Ans:-

function TT-ENTAILS? (KB, α) return true or false
inputs: KB, the knowledge base, a sentence in propositional logic.
α, a query in propositional logic
Symbols ← a list of propositional symbols in KB and α
return TT-CHECK-ALL(KB, α, symbols, {})

function TT-CHECK-ALL (KB, α, symbols, model) returns true or false
if EMPTY? (Symbol) then
if PL-TRUE? (KB, model) then return
PL-TRUE? (α, model)
else return true // when KB is false, always
// return true

else do
P ← FIRST (symbols)
rest ← REST (symbols)
return (TT-CHECK-ALL(KB, α, rest, model ∪ {P = true?})
and
TT-CHECK-ALL (KB, α, rest, model
{P = false?})).



Code: import itertools

```
def evaluate_formula(formula, valuation):
    """
    Evaluate the propositional formula under the given truth assignment (valuation).
    The formula is a string of logical operators like 'AND', 'OR', 'NOT', and can contain variables 'A', 'B', 'C'.
    """
    # Create a local environment (dictionary) for variable assignments
    env = {var: valuation[i] for i, var in enumerate(['A', 'B', 'C'])}

    # Replace logical operators with Python equivalents formula =
    formula.replace('AND', 'and').replace('OR', 'or').replace('NOT', 'not')

    # Replace variables in the formula with their corresponding truth values
    for var in env:
        formula = formula.replace(var, str(env[var]))
```

```
    # Evaluate the formula and return the result (True or False)
    try:
        return eval(formula)
    except Exception as e: raise ValueError(f"Error in
        evaluating formula: {e}")

def truth_table(variables):
    """
    Generate all possible truth assignments for the given variables.
    """ return list(itertools.product([False, True],
    repeat=len(variables)))

def entails(KB, alpha):
    """
    Decide if KB entails alpha using a truth-table enumeration algorithm.
    KB is a propositional formula (string), and alpha is another propositional formula (string).
    """
    # Generate all possible truth assignments for A, B, and C
    assignments = truth_table(['A', 'B', 'C'])

    print(f"{'A':<10}{'B':<10}{'C':<10}{'KB':<15}{'alpha':<15}{'KB entails alpha?'}") # Header for
the truth table print("-" * 70) # Separator
    for readability

    for assignment in assignments:
        # Evaluate KB and alpha under the current assignment
        KB_value = evaluate_formula(KB, assignment)
        alpha_value = evaluate_formula(alpha, assignment)

        # Print the current truth assignment and the results for KB and alpha

print(f"{str(assignment[0]):<10}{str(assignment[1]):<10}{str(assignment[2]):<10}{str(KB_value):<
15}{str(alpha_value):<15}{'Yes' if KB_value and alpha_value else 'No'}")

        # If KB is true and alpha is false, then KB does not entail
        alpha if KB_value and not alpha_value: return False

    # If no counterexample was found, then KB entails alpha
    return True

# Define the formulas for KB and alpha
```

```
alpha = 'A OR B'
KB = '(A OR C) AND (B OR NOT C)'

# Check if KB entails alpha
result = entails(KB, alpha)

# Print the final result of entailment
print(f"\nDoes KB entail alpha? {result}")
```

Output:

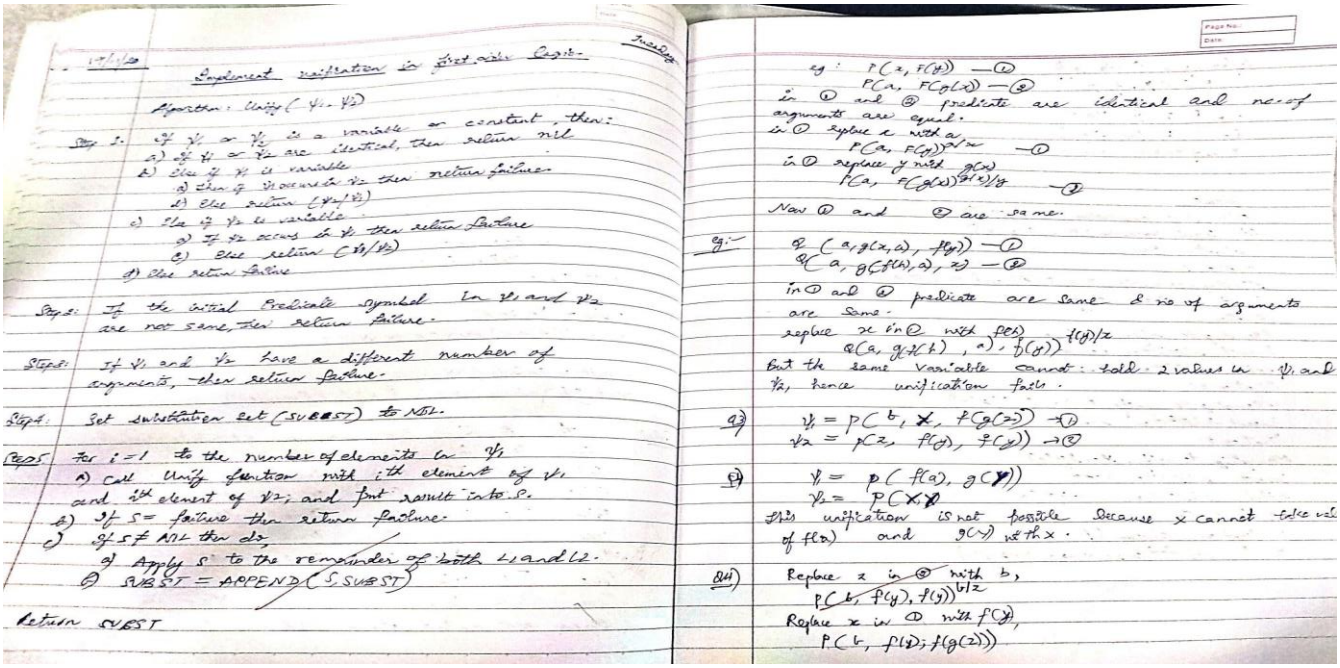
A	B	C	KB	alpha	KB entails alpha?
False	False	False	False	False	No
False	False	True	False	False	No
False	True	False	False	True	No
False	True	True	True	True	Yes
True	False	False	True	True	Yes
True	False	True	False	True	No
True	True	False	True	True	Yes
True	True	True	True	True	Yes

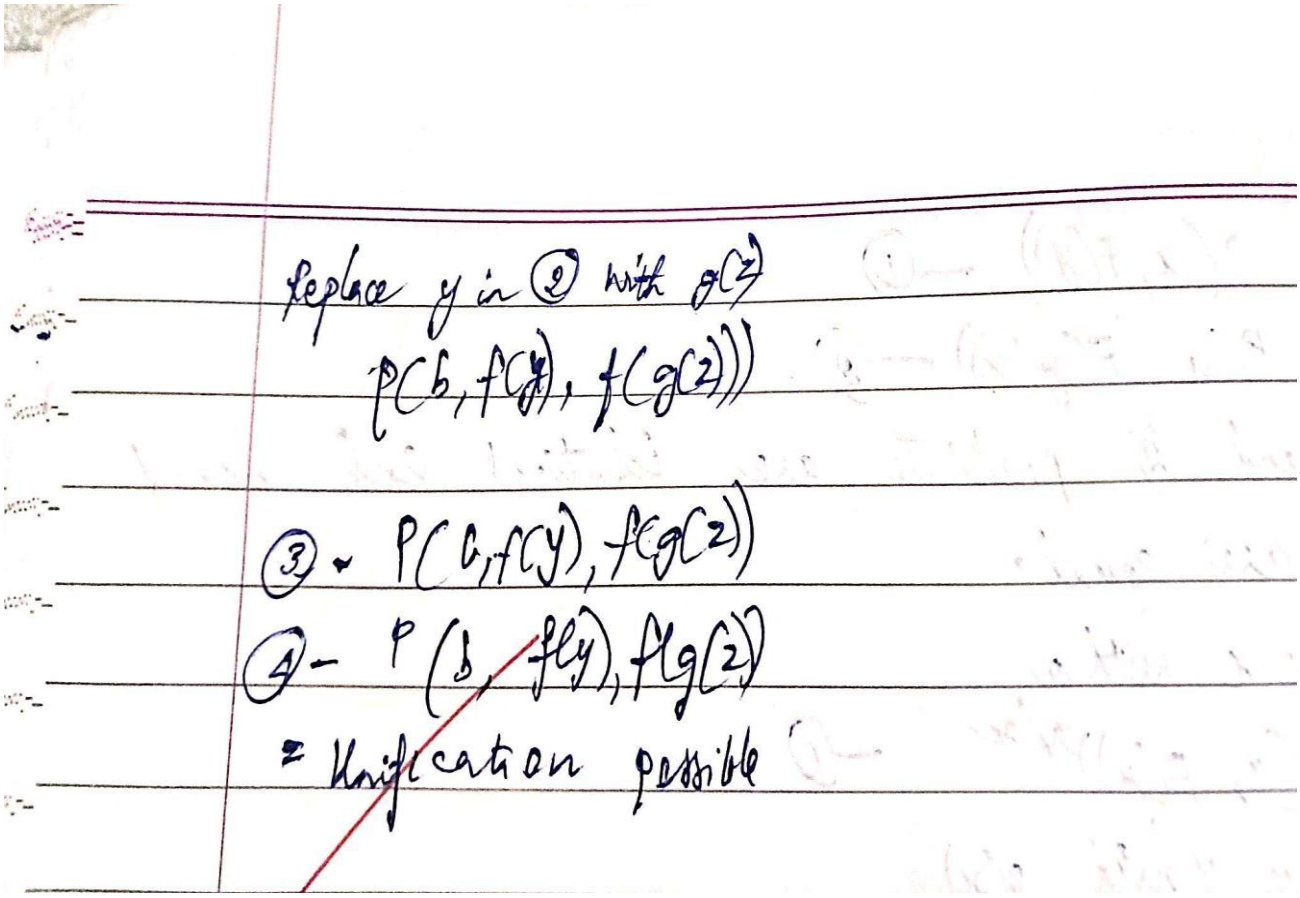
Does KB entail alpha? True

Program7

Implement unification in first order logic

Algorithm:





Code: class
Term:

```
    def __init__(self, symbol, args=None):
        self.symbol = symbol
        self.args = args if args else []

    def __str__(self):
        if not self.args:
            return str(self.symbol)
        return f"{self.symbol}({','.join(str(arg) for arg in self.args)})"

    def is_variable(self): return isinstance(self.symbol, str) and
        self.symbol.isupper() and not self.args

def occurs_check(var, term, substitution):
    """Check if variable occurs in term"""
    if term.is_variable():
        if term.symbol in substitution:
```

```
            return occurs_check(var, substitution[term.symbol], substitution)
        return var.symbol == term.symbol
    return any(occurs_check(var, arg, substitution) for arg in term.args)

def substitute(term, substitution): """Apply substitution
    to term""" if term.is_variable() and term.symbol in
    substitution:
        return substitute(substitution[term.symbol], substitution)
    if not term.args:
        return term
    return Term(term.symbol, [substitute(arg, substitution) for arg in term.args])

def unify(term1, term2, substitution=None, iteration=1):
    """Unify two terms with detailed iteration steps""" if
    substitution is None:
        substitution = {}

    print(f"\nIteration {iteration}:") print(f"Attempting to unify: {term1} and {term2}")
    print(f"Current substitution: {', '.join(f'{k}->{v}' for k,v in substitution.items()) or
    'empty'}")

    term1 = substitute(term1, substitution)
    term2 = substitute(term2, substitution)

    if term1.symbol == term2.symbol and not term1.args and not term2.args:
        print("Terms are identical - no substitution needed")
        return substitution

    if term1.is_variable():
        if occurs_check(term1, term2, substitution):
            print(f"Occurs check failed: {term1.symbol} occurs in {term2}")
            return None
        substitution[term1.symbol] = term2 print(f"Added
        substitution: {term1.symbol} -> {term2}") return
        substitution

    if term2.is_variable():
        if occurs_check(term2, term1, substitution):
            print(f"Occurs check failed: {term2.symbol} occurs in {term1}")
            return None
```

```
        substitution[term2.symbol] = term1 print(f"Added substitution:
    {term2.symbol} -> {term1}") return substitution if term1.symbol !=
    term2.symbol or len(term1.args) != len(term2.args):
    print(f"Unification failed: Different predicates or argument lengths")
    return None

    for arg1, arg2 in zip(term1.args, term2.args):
        result = unify(arg1, arg2, substitution, iteration + 1)
        if result is None:
            return None
        substitution = result

    return substitution

def parse_term(s):
    """Parse terms like P(X,f(Y)) or
    X""" s = s.strip() if '(' not in s:
        return Term(s)

    pred = s[:s.index('(')] args_str =
    s[s.index('(')+1:s.rindex(')')]

    args = [] current
    = '' depth = 0
    for c in
    args_str:
        if c == '(' or c == '[':
            depth += 1
        elif c == ')' or c == ']':
            depth -= 1
        elif c == ',' and depth == 0:
            args.append(parse_term(current.strip())
            ) current = '' continue
        current += c if current:
    args.append(parse_term(current.strip()))
    return Term(pred, args)

def print_examples():
```

```
    print("\nExample format:") print("1.
    Simple terms: P(X,Y)") print("2. Nested
    terms: P(f(X),g(Y))") print("3. Mixed
    terms: Knows(John,X)") print("4.
    Complex nested terms:
    P(f(g(X)),h(Y,Z))") print("\nNote: Use
    capital letters for variables (X,Y,Z) and
    lowercase for constants and
predicates.")

def validate_input(expr):
    """Basic validation for input expressions"""
    if not expr: return False

    # Check balanced
    parentheses count = 0 for
    char in expr:
        if char == '(':
            count += 1
        elif char == ')':
            count -= 1
        if count < 0:
            return False
    return count == 0

def main():
    while True:
        print("\n=== First Order Predicate Logic Unification
        ===") print("1. Start Unification") print("2. Show
        Examples") print("3. Exit") choice = input("\nEnter your
        choice (1-3): ")

        if choice == '1': print("\nEnter two
            expressions to unify.") print_examples()

            while True:
                expr1 = input("\nEnter first expression (or 'back' to return):
                ") if expr1.lower() == 'back': break

                if not validate_input(expr1):
```

```
            print("Invalid expression! Please check the format and try
        again.") continue expr2 = input("Enter second expression: ")
        if not validate_input(expr2):
            print("Invalid expression! Please check the format and try again.")
            continue

        try:
            term1 = parse_term(expr1)
            term2 = parse_term(expr2)

            print("\nUnification Process:")
            result = unify(term1, term2)

            print("\nFinal Result:")
            if result is None:
                print("Unification failed!")
            else:
                print("Unification successful!") print("Final substitutions:", ',
                '.join(f'{k}->{v}' for k,v in result.items()))

            retry = input("\nTry another unification? (y/n):
            ") if retry.lower() != 'y': break

        except Exception as e:
            print(f"Error processing expressions: {str(e)}")
            print("Please check your input format and try again.")

elif choice == '2':
    print("\n=== Example Expressions ===") print("1.
    P(X,h(Y)) and P(a,f(Z))") print("2. P(f(a),g(Y)) and
    P(X,X)") print("3. Knows(John,X) and
    Knows(X,Elisabeth)") print("\nPress Enter to
    continue...") input()

elif choice == '3':
    print("\nThank you for using the Unification Program!")
    break

else: print("\nInvalid choice! Please enter 1, 2, or
    3.")
```

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

Output:

```
=== First Order Predicate Logic Unification ===
1. Start Unification
2. Show Examples
3. Exit

Enter your choice (1-3): 1

Enter two expressions to unify.

Example format:
1. Simple terms: P(X,Y)
2. Nested terms: P(f(X),g(Y))
3. Mixed terms: Knows(John,X)
4. Complex nested terms: P(f(g(X)),h(Y,Z))

Note: Use capital letters for variables (X,Y,Z) and lowercase for constants and predicates.

Enter first expression (or 'back' to return): P(X,F(Y))
Enter second expression: P(a,F(g(X)))

Unification Process:

Iteration 1:
Attempting to unify: P(X,F(Y)) and P(a,F(g(X)))
Current substitution: empty

Iteration 2:
Attempting to unify: X and a
Current substitution: empty
Added substitution: X -> a

Iteration 2:
Attempting to unify: F(Y) and F(g(X))
Current substitution: X->a

Iteration 3:
Attempting to unify: Y and g(a)
```

Current substitution: X->a
Added substitution: Y -> g(a)
Final Result:
Unification successful!
Final substitutions: X->a, Y->g(a)

Try another unification? (y/n): y

Enter first expression (or 'back' to return): P(F(a),g(Y))
Enter second expression: P(X,X)

Unification Process:

Iteration 1:
Attempting to unify: P(F(a),g(Y)) and P(X,X)
Current substitution: empty

Iteration 2:
Attempting to unify: F(a) and X
Current substitution: empty
Added substitution: X -> F(a)

Iteration 2:
Attempting to unify: g(Y) and X
Current substitution: X->F(a)
Unification failed: Different predicates or argument lengths

Final Result:
Unification failed!

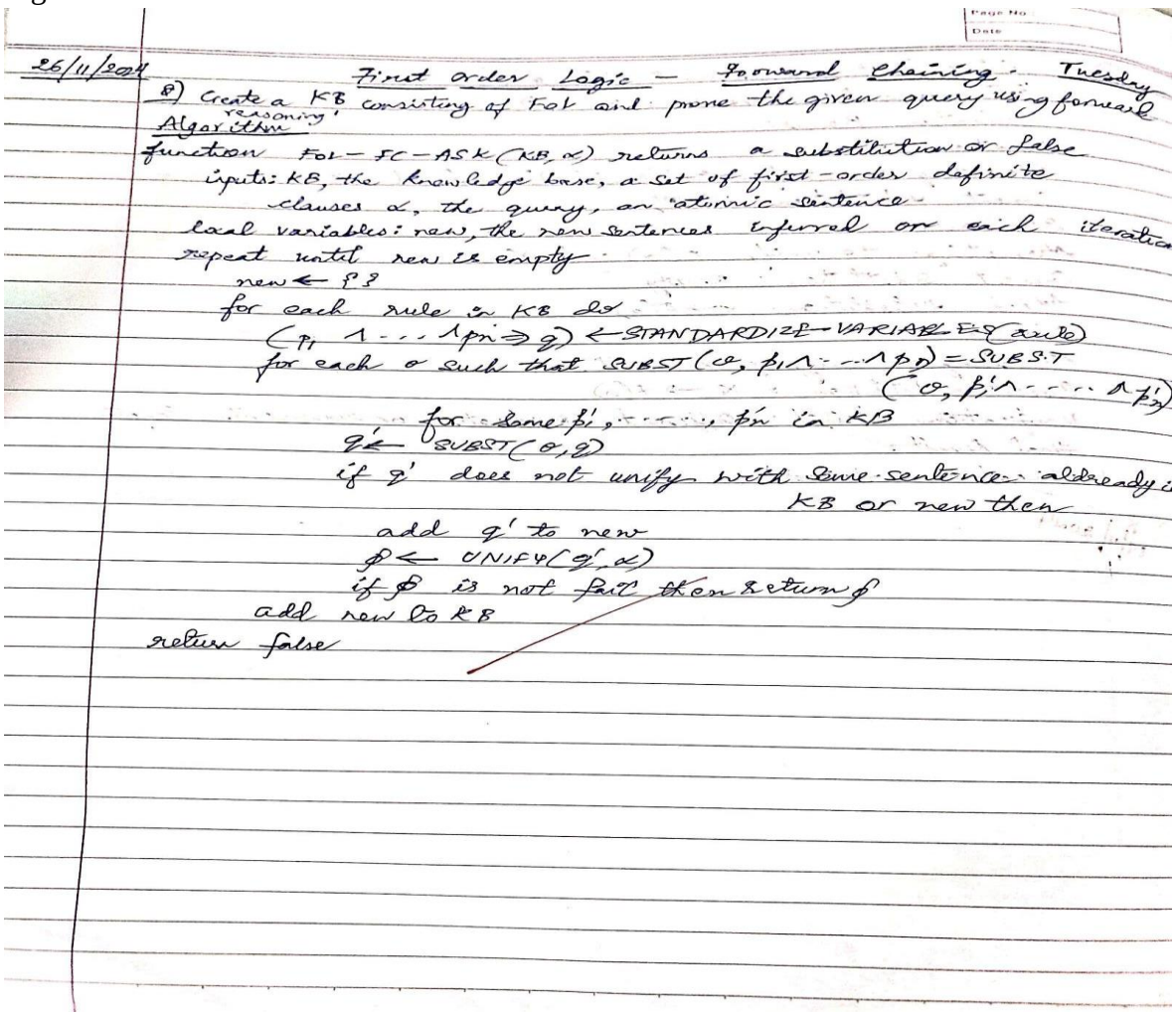
Try another unification? (y/n): n

=== First Order Predicate Logic Unification ===
1. Start Unification
2. Show Examples
3. Exit

Program8

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

Algorithm:



Code:

 # Define the knowledge base as a list of rules and facts

class KnowledgeBase:

def __init__(self):

self.facts = set() *# Set of known facts*

self.rules = [] *# List of rules*

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

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

    def infer(self): inferred =
        True while inferred:
        inferred = False for
        rule in self.rules:
                if rule.apply(self.facts):
                    inferred = True

# Define the Rule class
class Rule:
    def __init__(self, premises, conclusion):
        self.premises = premises # List of conditions
        self.conclusion = conclusion # Conclusion to add if premises are met

    def apply(self, facts):
        if all(premise in facts for premise in self.premises):
            if self.conclusion not in facts:
            facts.add(self.conclusion) print(f"Inferred:
            {self.conclusion}") return True
        return False

# Initialize the knowledge base
kb = KnowledgeBase()
```

```
# Facts in the problem
kb.add_fact("American(Robert)")
kb.add_fact("Missile(T1)")
kb.add_fact("Owns(A, T1)")
kb.add_fact("Enemy(A, America)")

# Rules based on the problem # 1. Missile(x)
implies Weapon(x)
kb.add_rule(Rule(["Missile(T1)"],
"Weapon(T1)"))

# 2. Enemy(x, America) implies Hostile(x)
kb.add_rule(Rule(["Enemy(A, America)"], "Hostile(A)"))

# 3. Missile(x) and Owns(A, x) imply Sells(Robert, x, A)
kb.add_rule(Rule(["Missile(T1)", "Owns(A, T1)"], "Sells(Robert, T1, A)"))

# 4. American(p) and Weapon(q) and Sells(p, q, r) and Hostile(r) imply Criminal(p)
kb.add_rule(Rule(["American(Robert)", "Weapon(T1)", "Sells(Robert, T1, A)", "Hostile(A)"],
"Criminal(Robert)"))

# Infer new facts based on the rules
kb.infer()

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

Output:

Inferred: Weapon(T1)

Inferred: Hostile(A)

Inferred: Sells(Robert, T1, A)

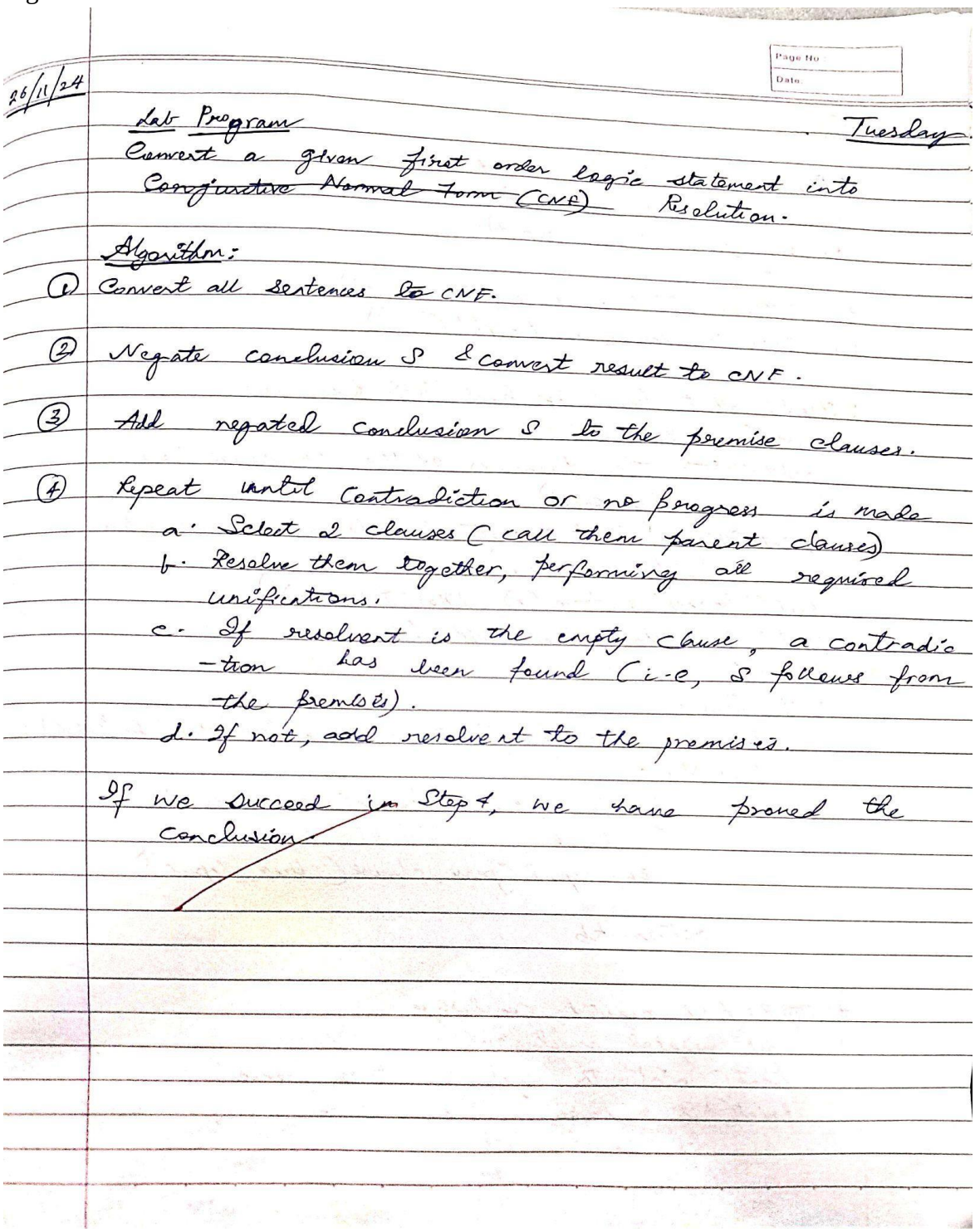
Inferred: Criminal(Robert)

Conclusion: Robert is a criminal.

Program9

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

Algorithm:



26/11/24

Lab Program — Tuesday

Convert a given first order logic statement into ~~Conjunctive Normal Form (CNF)~~ Resolution.

Algorithm:

1. Convert all sentences to CNF.
2. Negate conclusion S & convert result to CNF.
3. Add negated conclusion S to the premise clauses.
4. Repeat until Contradiction or no progress is made
   a. Select 2 clauses (call them parent clauses)
   b. Resolve them together, performing all required unifications.
   c. If resolvent is the empty clause, a contradiction has been found (i.e, S follows from the premises).
   d. If not, add resolvent to the premises.

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

26/11/24 Tuesday

Lab program 9

Continuation

Given the KB as Previous:
John likes all kind of food
Apple and vegetable are food
Anything anyone eats and not killed is food
Anil eats peanuts and still alive.
Harry eats everything that Anil eats.
Anyone who is alive implies not killed.
Anyone who is not killed implies alive.

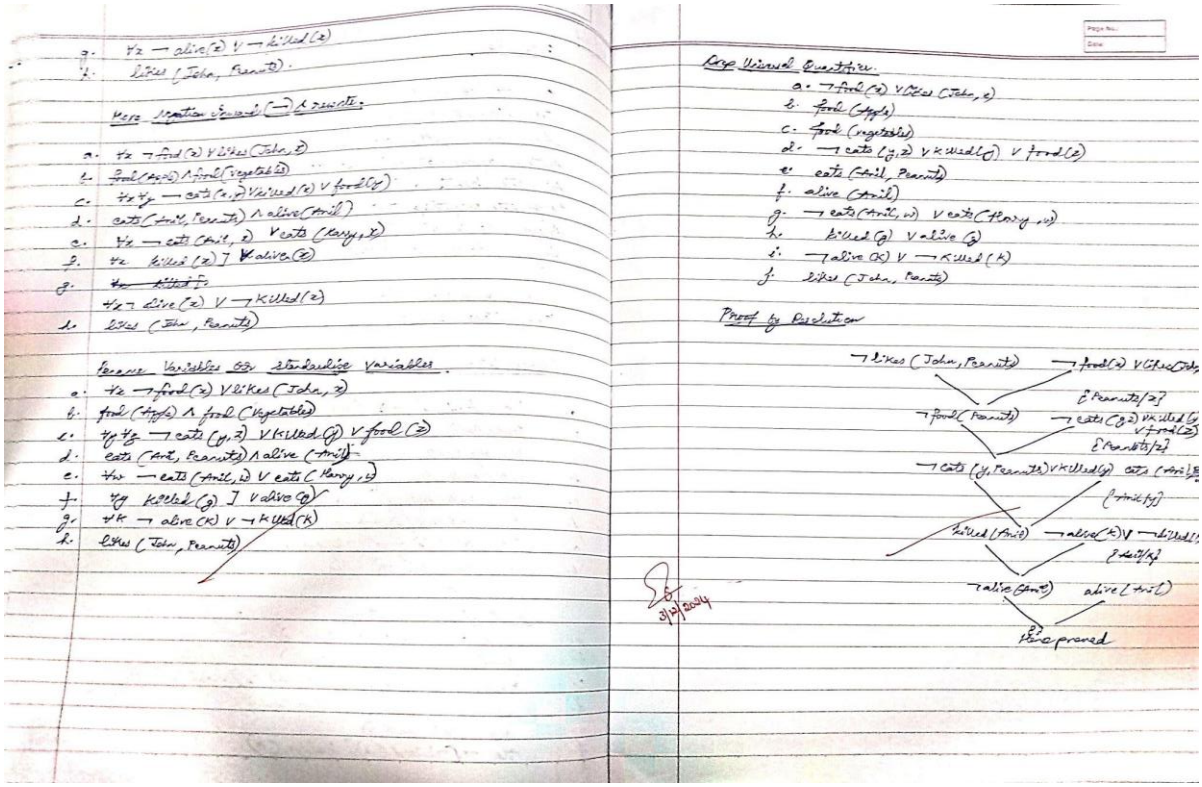
Prove by resolution that:
John likes peanuts.

Representations in FOL:

a. $\forall x : food(x) \rightarrow likes(John, x)$
b. $food(Apple) \wedge food(Vegetables)$
c. $\forall x \forall y : eats(x,y) \wedge \neg killed(x) \rightarrow food(y)$
d. $eats(Anil, Peanuts) \wedge alive(Anil)$.
e. $\forall x : eats(Anil, x) \rightarrow eats(Harry, x)$
f. $\forall x : \neg killed(x) \rightarrow alive(x)$
g. $\forall x : alive(x) \rightarrow \neg killed(x)$
h. $likes(John, Peanuts)$

Eliminate Implication $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$

a. $\forall x \neg food(x) \vee likes(John, x)$
b. $food(Apple) \wedge food(Vegetables)$
c. $\forall x \forall y \neg[eats(x,y) \wedge \neg killed(x)] \vee food(y)$
d. $eats(Anil, Peanuts) \wedge alive(Anil)$
e. $\forall x \neg eats(Anil, x)$
f. $\forall x \neg[\neg killed(x)] \vee alive(x)$



Code:

```
# Step 1: Helper function to parse user inputs into clauses (with '!' for negation)
def parse_clause(clause_input):
    """
    Parses a user input string into a tuple of literals for the clause.
    Replaces '!' with '¬' for negation handling.
    Example: "!Food(x), Likes(John, x)" -> ("¬Food(x)", "Likes(John, x)")
    """
    return tuple(literal.strip().replace("!", "¬") for literal in clause_input.split(","))

# Step 2: Collect knowledge base (KB) from user
def get_knowledge_base():
    print("Enter the premises of the knowledge base, one by one.")
    print("Use ',' to separate literals in a clause. Use '!' for
    negation.") print("Example: !Food(x), Likes(John, x)")
    print("Type 'done' when finished.")

    kb = []
    while True:
        clause_input = input("Enter a clause (or 'done' to finish):
        ").strip() if clause_input.lower() == "done": break
```

```
        kb.append(parse_clause(clause_input))

    return kb

# Step 3: Add negated conclusion
def get_negated_conclusion():
    print("\nEnter the conclusion to be proved.")
    print("It will be negated automatically for proof by contradiction.") conclusion
    = input("Enter the conclusion (e.g., Likes(John, Peanuts)): ").strip() negated =
    f"!{conclusion}" if not conclusion.startswith("!") else conclusion[1:] return
    (negated.replace("!", "¬"),) # Convert '!' to '¬' for consistency

# Step 4: Resolution algorithm
def resolve(clause1, clause2):
"""
    Resolves two clauses and returns the resolvent (new clause).
    If no resolvable literals exist, returns an empty set.
    """ resolved = set()
    for literal in
    clause1:
        if "¬" + literal in clause2: temp1
            = set(clause1) temp2 =
            set(clause2)
            temp1.remove(literal)
            temp2.remove("¬" + literal)
            resolved =
            temp1.union(temp2)
        elif literal.startswith("¬") and literal[1:] in clause2:
            temp1 = set(clause1) temp2 =
            set(clause2)
            temp1.remove(literal)
            temp2.remove(literal[1:])
            resolved =
            temp1.union(temp2)
    return tuple(resolved)

def resolution(kb):
    """
    Runs the resolution algorithm on the knowledge base (KB). Returns
    True if an empty clause is derived (proving the conclusion), or
    False if resolution fails.
```

```
    """ clauses =
    set(kb) new =
    set()

    while True:
        # Generate all pairs of clauses
        pairs = [(ci, cj) for ci in clauses for cj in clauses if ci != cj]

        for (ci, cj) in pairs:
            resolvent = resolve(ci, cj) if not
            resolvent: # Empty clause found
            return True
            new.add(resolvent)

        if new.issubset(clauses): # No new clauses
            return False
        clauses = clauses.union(new)

# Step 5: Main execution if
__name__ == "__main__":
    print("=== Resolution Proof System ===") print("Provide
    the knowledge base and conclusion to prove.")

    # Collect user inputs kb =
    get_knowledge_base() negated_conclusion =
    get_negated_conclusion()
    kb.append(negated_conclusion)

    # Show the knowledge base
    print("\nKnowledge Base (KB):")
    for clause in kb:
            print(" ", " ∨ ".join(clause)) # Join literals with OR for readability

    # Perform resolution print("\nStarting
    resolution process...")
    result = resolution(kb) if result: print("\nProof complete: The
    conclusion is TRUE.") else: print("\nResolution failed: The
    conclusion could not be proved.")
```

Output:

```
=== Resolution Proof System ===
Provide the knowledge base and conclusion to prove.
Enter the premises of the knowledge base, one by one.
```

Use ',' to separate literals in a clause. Use '!' for negation.
Example: !Food(x), Likes(John, x) Type 'done' when
finished.
Enter a clause (or 'done' to finish): !Food(x),Likes(John,x)
Enter a clause (or 'done' to finish): Food(Apple)
Enter a clause (or 'done' to finish): Food(Vegetables)
Enter a clause (or 'done' to finish): !Eats(x,y),!Killed(x),Food(y)
Enter a clause (or 'done' to finish): Eats(Anil,Peanuts)
Enter a clause (or 'done' to finish): !Killed(Anil)
Enter a clause (or 'done' to finish): done

Enter the conclusion to be proved.
It will be negated automatically for proof by contradiction.
Enter the conclusion (e.g., Likes(John, Peanuts)): Likes(John,Peanuts)

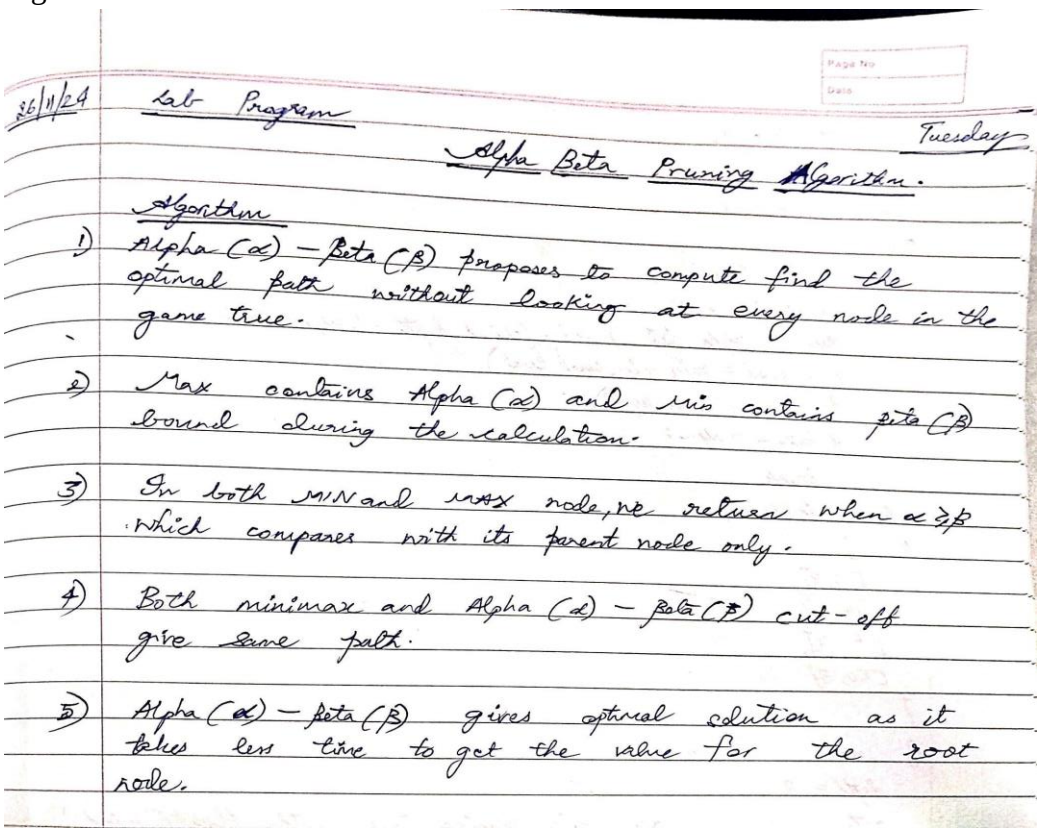
Knowledge Base (KB):
¬Food(x) ∨ Likes(John ∨ x)
Food(Apple)
Food(Vegetables)
¬Eats(x ∨ y) ∨ ¬Killed(x) ∨ Food(y)
Eats(Anil ∨ Peanuts)
¬Killed(Anil)
¬Likes(John,Peanuts)

Starting resolution process...
Proof complete: The conclusion is TRUE.

Program10

Implement Alpha-Beta Pruning.

Algorithm:



26/11/24 Lab Program

Tuesday

Alpha Beta Pruning Algorithm.

Algorithm

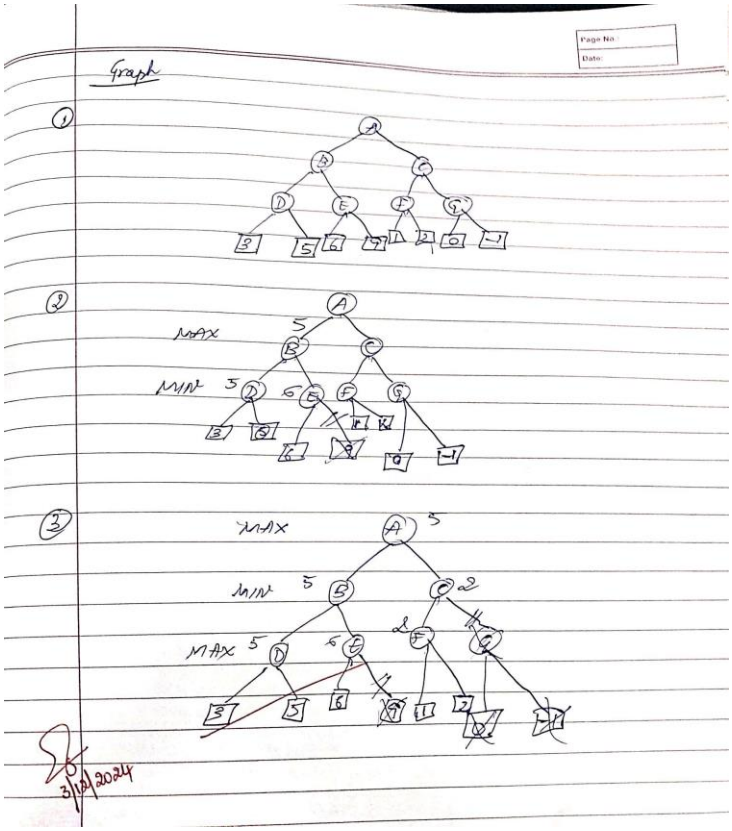
1) Alpha (α) – Beta (β) proposes to compute find the optimal path without looking at every node in the game tree.

2) Max contains Alpha (α) and Min contains Beta (β) bound during the calculation.

3) In both MIN and MAX node, we return when α ≥ β which compares with its parent node only.

4) Both minimax and Alpha (α) – Beta (β) cut-off give same path.

5) Alpha (α) – Beta (β) gives optimal solution as it takes less time to get the value for the root node.



Code:

```
def alpha_beta_pruning(node, depth, alpha, beta, maximizing_player):
    """
    Alpha-Beta Pruning Algorithm.
    Args:
        node: Current node value (can be a game state or a value in the tree).
        depth: Depth of the node in the tree (0 for leaves). alpha: Best value
        the maximizer can guarantee. beta: Best value the minimizer can
        guarantee. maximizing_player: Boolean, True if the current player is
        maximizing.
    Returns:
        Best value for the current player.
    """
    # Base case: if at a leaf node or maximum
    depth if depth == 0 or isinstance(node, int):
    return node

    if maximizing_player:
        max_eval = float('-inf')
        for child in node: # Assuming `node` is a list of child nodes
```

```
            eval = alpha_beta_pruning(child, depth - 1, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval) if
            beta <= alpha: # Beta cutoff
                break
        return max_eval
    else:
        min_eval = float('inf')
        for child in node: # Assuming `node` is a list of child nodes
            eval = alpha_beta_pruning(child, depth - 1, alpha, beta,
            True) min_eval = min(min_eval, eval) beta = min(beta, eval)
            if beta <= alpha: # Alpha cutoff
                break
        return min_eval


# Example tree from the diagram
tree = [
    [10, 9],            # First branch of the tree
    [14, 18],           # Second branch of the
                        tree
    [5, 4],            # Third branch of the tree
    [50, 3]             # Fourth branch of the tree
]

# Example usage if
__name__ == "__main__":
    # Depth of the tree (levels of decision-
    making) depth = 2 # Adjusted based on tree
    structure # Call the alpha-beta pruning
    algorithm
    optimal_value = alpha_beta_pruning(tree, depth, float('-inf'), float('inf'), True)
print("Optimal Value:", optimal_value) Output:
Optimal Value: 14
```