VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum- 590014, Karnataka.

LAB RECORD

Artificial Intelligence (23CS5PCAIN)

Submitted by

Likhith M (1BM22CS135)

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

Academic Year 2024 - 25 (odd)

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 **Likhith M (1BM22CS135),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfilment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Laboratory report has been approved as it satisfies the academic requirements of the above-mentioned subject and the work prescribed for the said degree.

Prameetha Pai	Dr. Kavitha Sooda
Assistant Professor	Professor & HOD
Department of CSE, BMSCE	Department of CSE, BMSCE

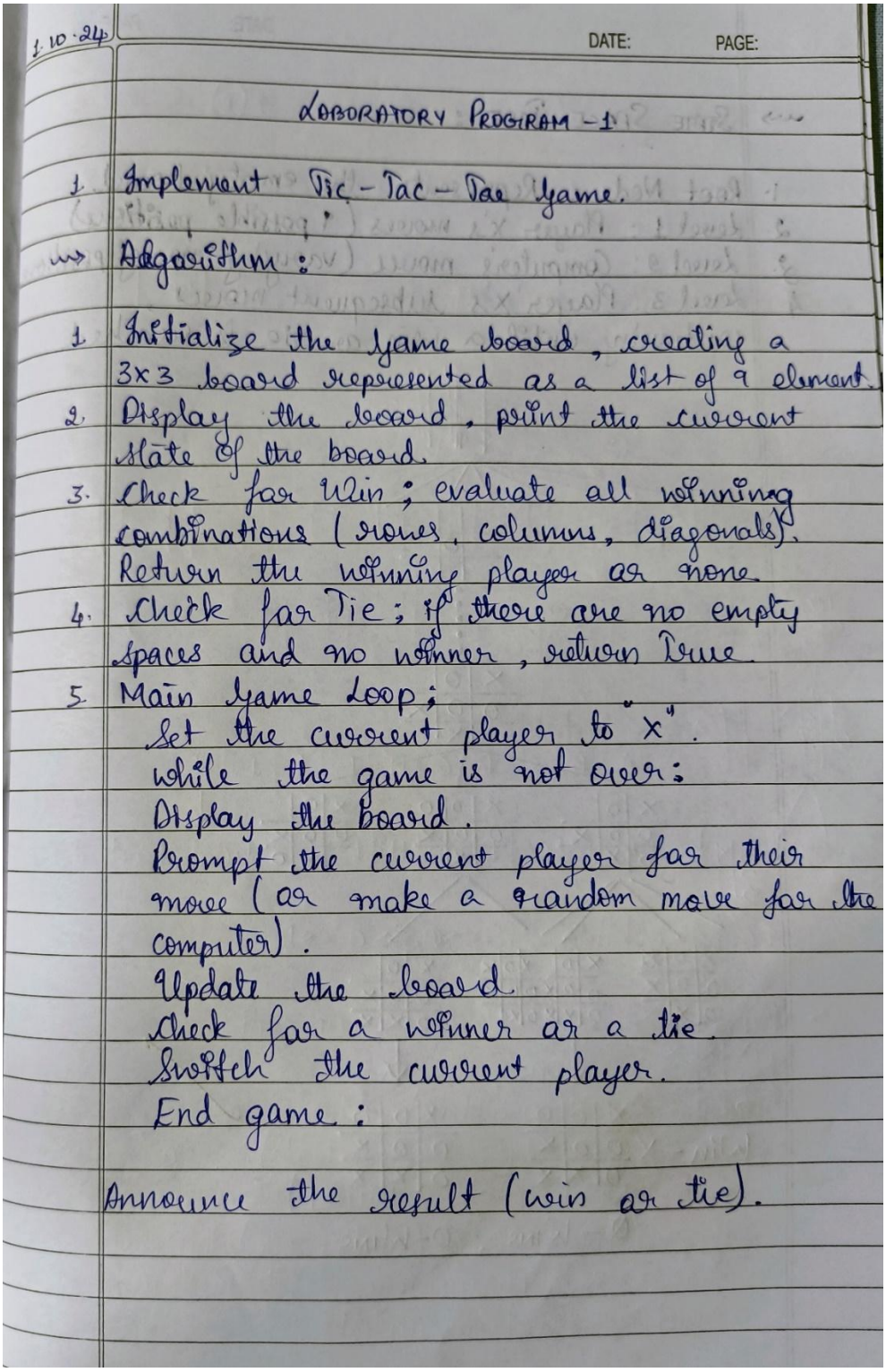
INDEX

Github Link: https://github.com/01-BLUELOTUS/AI-1BM22CS135.git

LABORATORY PROGRAM – 1

Implement Tic –Tac –Toe Game

PSEUDOCODE OR ALGORITHM



1.10.24 DATE: PAGE:

LABORATORY PROGRAM - 1

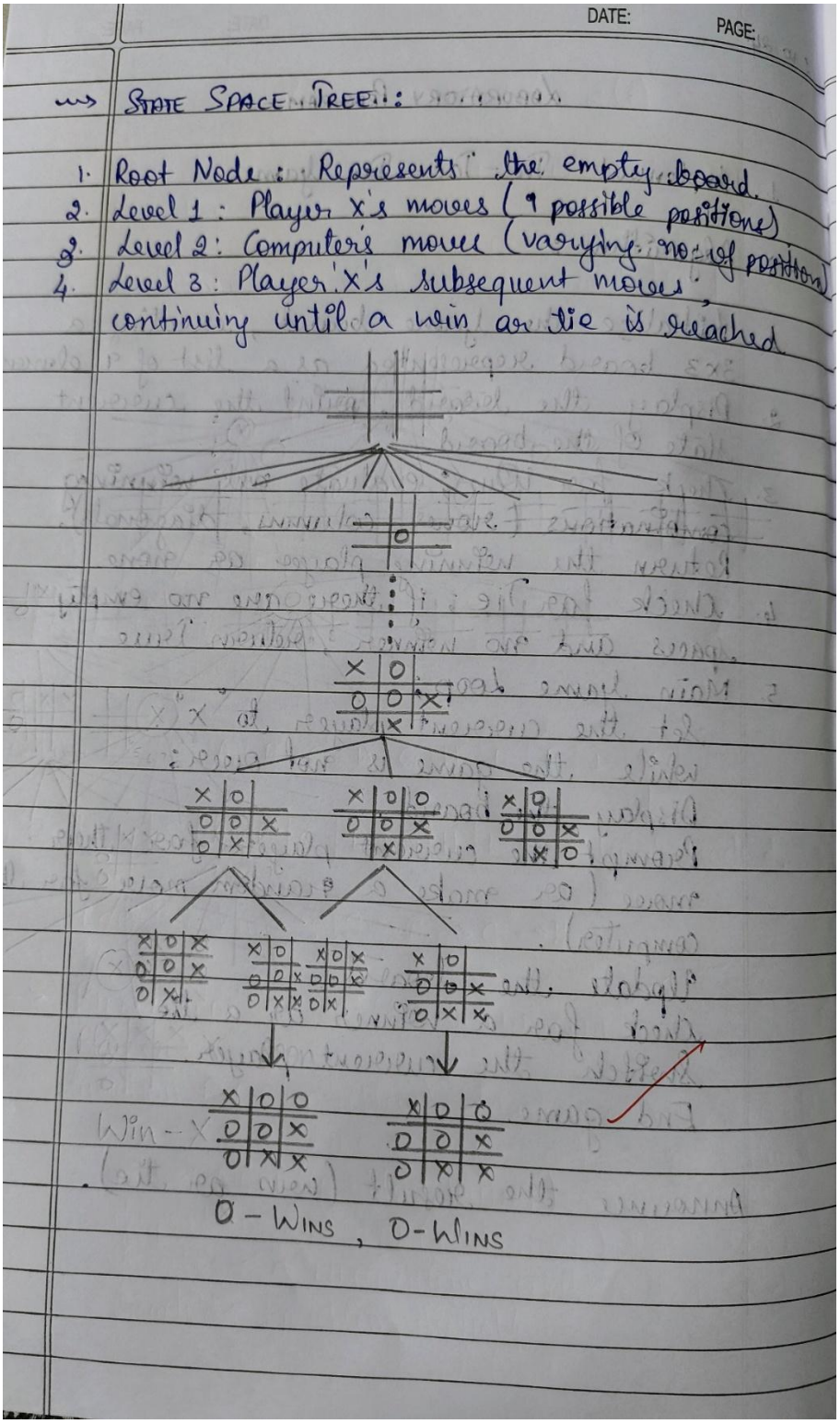
1. Implement Tic - Tac - Toe game.

⇒ Algorithm :

1. Initialize the game board, creating a 3x3 board represented as a list of 9 element.
2. Display the board, print the current state of the board.
3. Check for Win ; evaluate all winning combinations (rows, columns, diagonals). Return the winning player or none.
4. Check for Tie ; if there are no empty spaces and no winner, return True.
5. Main game loop ;
   Set the current player to "X".
   while the game is not over :
   Display the board.
   Prompt the current player for their move (or make a random move for the computer).
   Update the board.
   Check for a winner or a tie.
   Switch the current player.
   End game :

Announce the result (win or tie).

STATE SPACE TREE



CODE

```
import random

board = ["-", "-", "-",
         "-", "-", "-",
         "-", "-", "-"]

def p_board(player):
    print(board[0] + "|" + board[1] + "|" + board[2])
    print(board[3] + "|" + board[4] + "|" + board[5])
    print(board[6] + "|" + board[7] + "|" + board[8])

def check_win():

  for i in range(0, 7, 3):
    if board[i] == board[i + 1] == board[i + 2] and board[i] != '-':
      return board[i]

  for i in range(3):
    if board[i] == board[i + 3] == board[i + 6] and board[i] != '-':
      return board[i]

  if board[0] == board[4] == board[8] and board[0] != '-':
    return board[0]
  if board[2] == board[4] == board[6] and board[2] != '-':
    return board[2]

  return None

def check_tie():
  if '-' not in board:
    return True
  return False

def play_game():
  current_player = "X"
  game_over = False

  while not game_over:
    p_board(current_player)

    if current_player == "X":
      print("It's your turn (X).")
      try:
        position = int(input("Choose a position from 1-9: ")) - 1
        if position < 0 or position > 8 or board[position] != '-':
          print("Invalid move. Try again.")
          continue
      except ValueError:
        print("Invalid input. Please enter a number between 1 and 9.")
        continue
    else:
```

```
            print("Computer's turn (O).")
            available_moves = [i for i, spot in enumerate(board) if spot == '-']
            position = random.choice(available_moves)

        board[position] = current_player
        winner = check_win()

        if winner:
            p_board(current_player)
            print(winner + " won!")
            game_over = True
        elif check_tie():
            p_board(current_player)
            print("It's a tie!")
            game_over = True
        else:
            current_player = "O" if current_player == "X" else "X"

play_game()
```

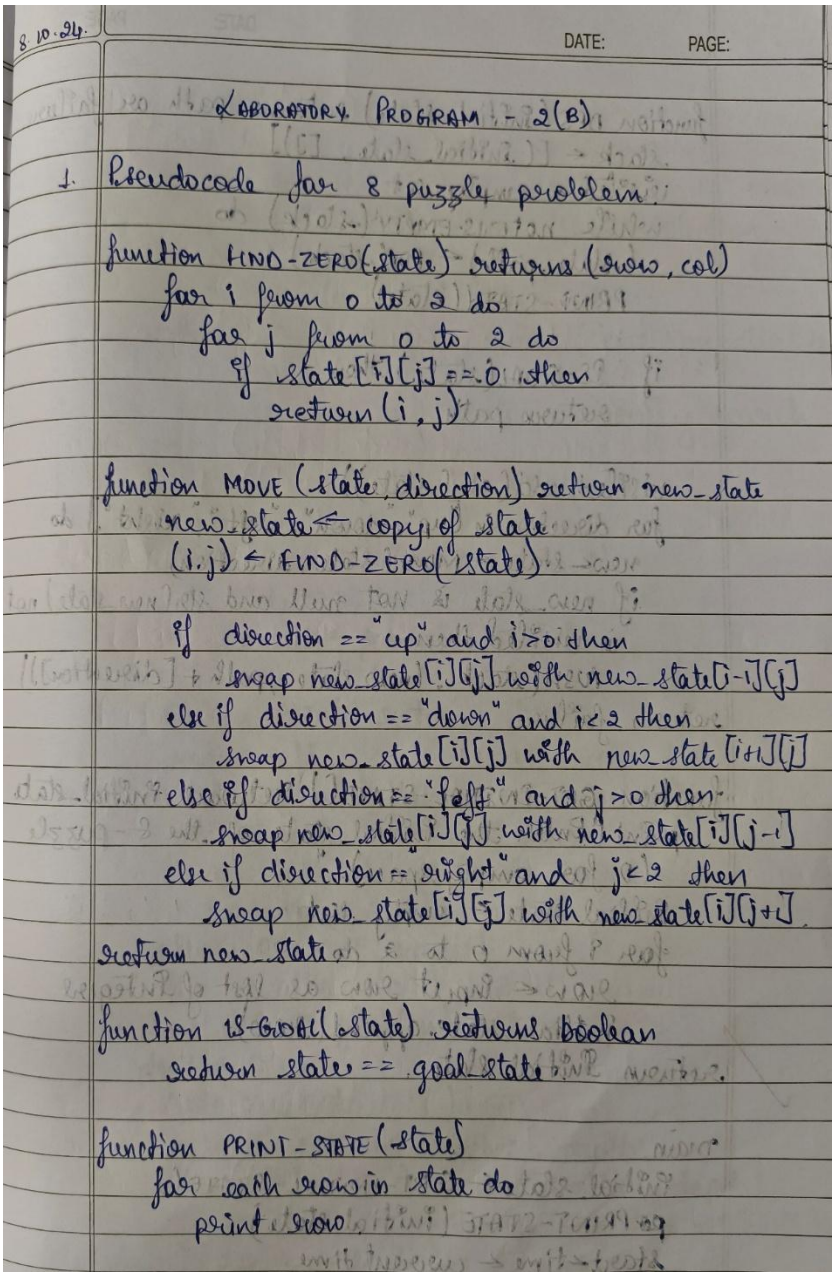
OUTPUT

```
-|-|-
-|-|-
-|-|-
It's your turn (X).
Choose a position from 1-9: 5
-|-|-
-|X|-
-|-|-
Computer's turn (O).
-|O|-
-|X|-
-|-|-
It's your turn (X).
Choose a position from 1-9: 7
-|O|-
-|X|-
X|-|-
Computer's turn (O).
-|O|O
-|X|-
X|-|-
It's your turn (X).
Choose a position from 1-9: 1
X|O|O
-|X|-
X|-|-
Computer's turn (O).
X|O|O
-|X|O
X|-|-
It's your turn (X).
Choose a position from 1-9: 4
X|O|O
X|X|O
X|-|-
X won!
```

LABORATORY PROGRAM – 2

Solve 8 puzzle problems

PSEUDOCODE OR ALGORITHM

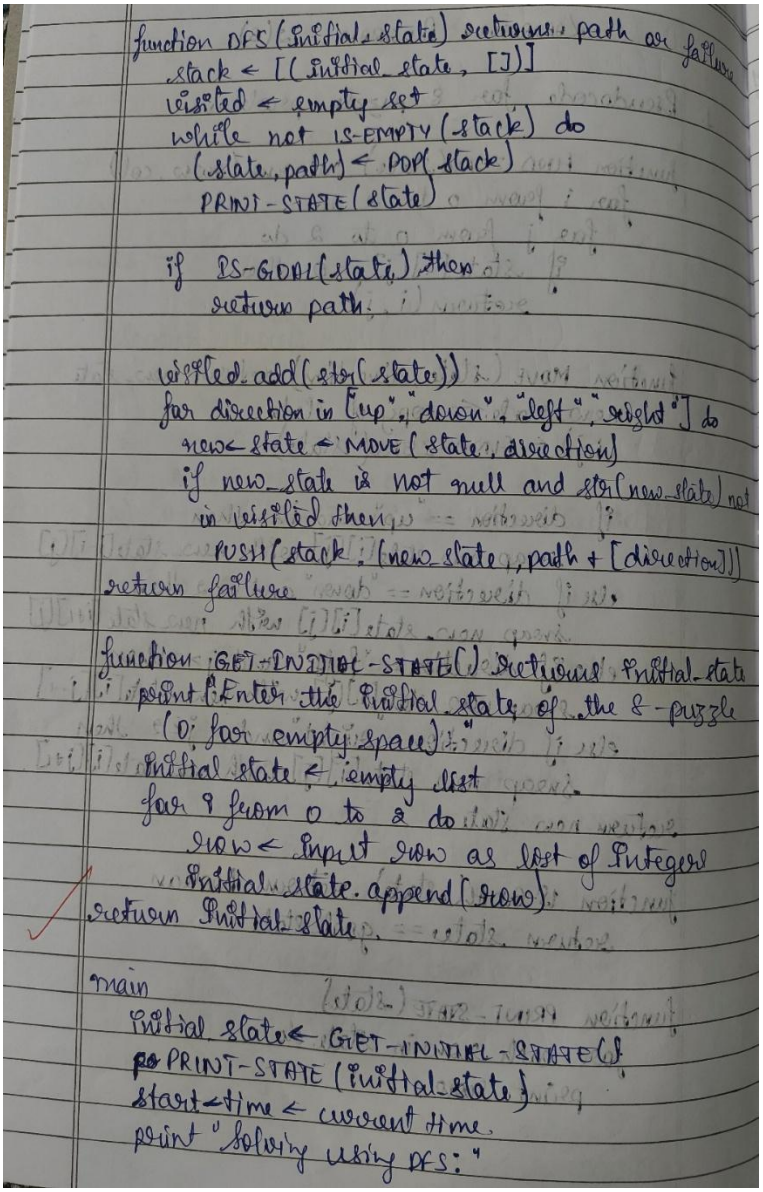


8.10.24 DATE: PAGE:

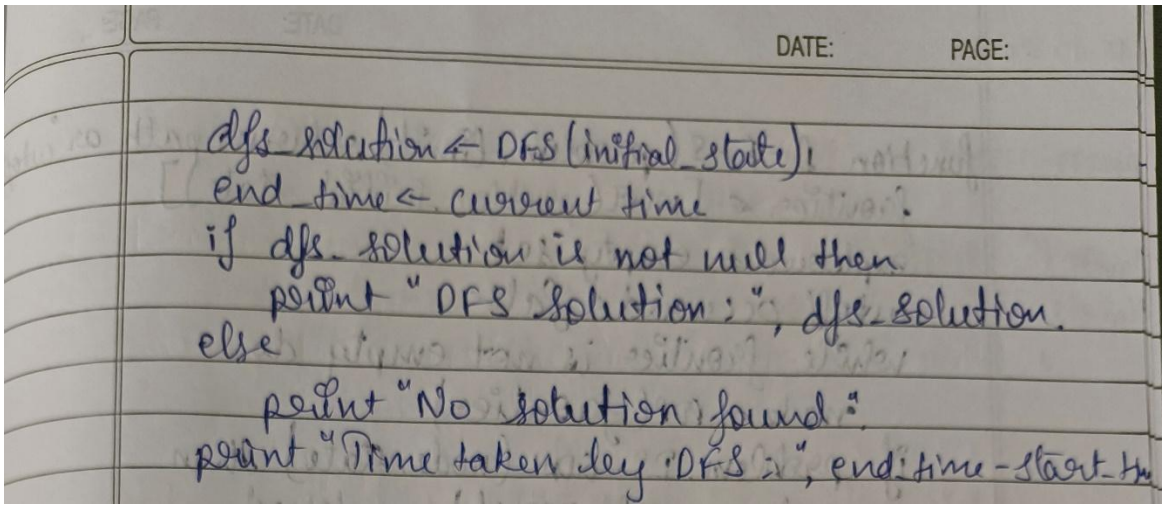
LABORATORY PROGRAM :- 2(B)

1. Pseudocode for 8 puzzle problem:

```
function FIND-ZERO(state) returns (row, col)
    for i from 0 to 2 do
        for j from 0 to 2 do
            if state[i][j] == 0 then
                return (i, j)

function MOVE (state, direction) return new_state
    new_state ← copy of state
    (i,j) ← FIND-ZERO(state)
    if direction == "up" and i>0 then
        swap new_state[i][j] with new_state[i-1][j]
    else if direction == "down" and i<2 then
        swap new_state[i][j] with new_state[i+1][j]
    else if direction == "left" and j>0 then
        swap new_state[i][j] with new_state[i][j-1]
    else if direction == "right" and j<2 then
        swap new_state[i][j] with new_state[i][j+1]
    return new_state

function IS-GOAL(state) returns boolean
    return state == goal_state

function PRINT-STATE(state)
    for each row in state do
        print row
```

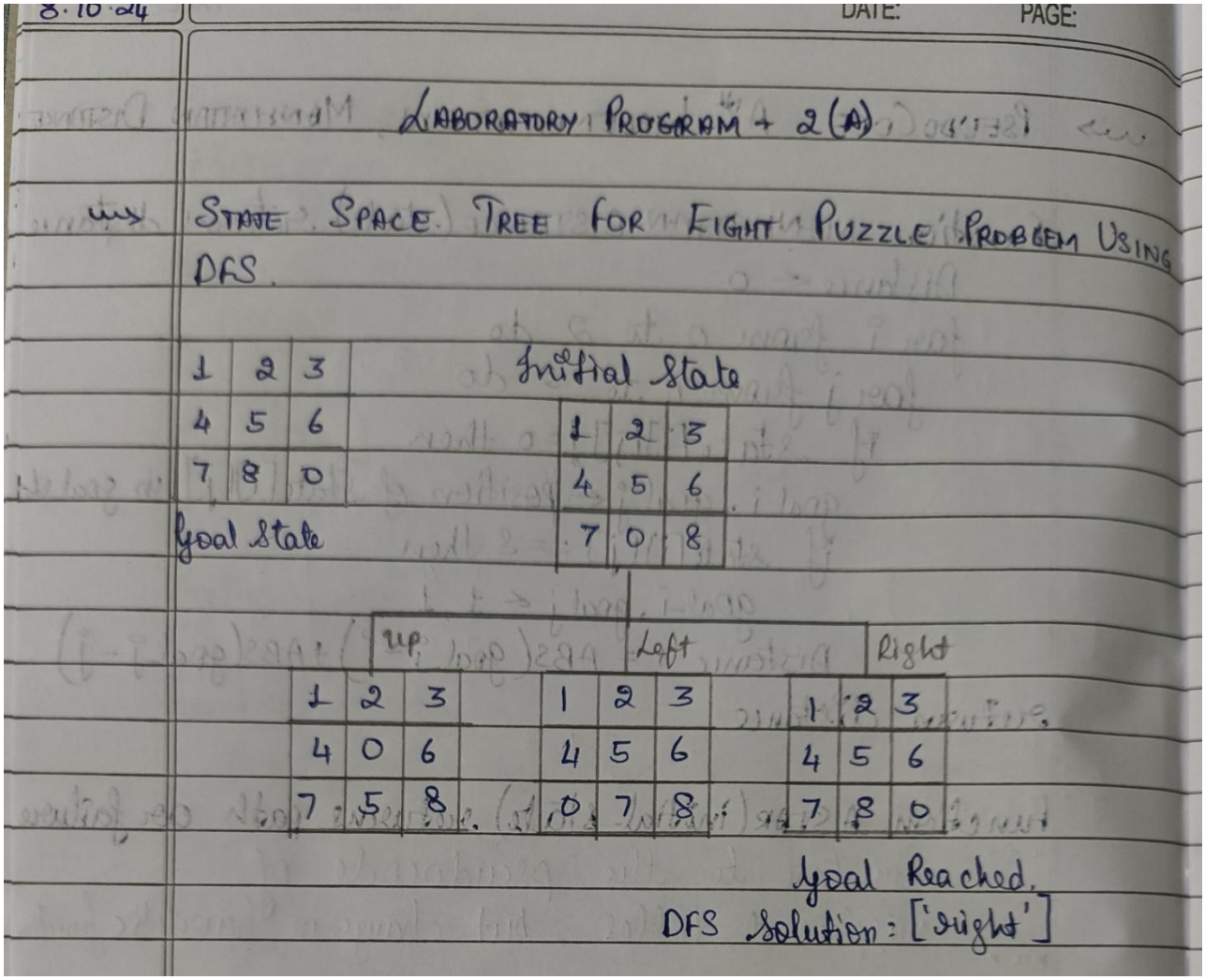


```
function DFS(Initial_state) returns path or failure
    stack ← [(initial_state, [])]
    visited ← empty set
    while not IS-EMPTY(stack) do
        (state, path) ← POP(stack)
        PRINT-STATE(state)

        if IS-GOAL(state) then
            return path

        visited.add(str(state))
        for direction in ["up", "down", "left", "right"] do
            new_state ← MOVE(state, direction)
            if new_state is not null and str(new_state) not
            in visited then
                PUSH(stack, (new_state, path + [direction]))
    return failure

function GET-INITIAL-STATE() returns initial_state
    print("Enter the initial state of the 8-puzzle
        (0 for empty space)")
    initial_state ← empty list
    for i from 0 to 2 do
        row ← input row as list of integers
        initial_state.append(row)
    return initial_state

main
    initial_state ← GET-INITIAL-STATE()
    PRINT-STATE(initial_state)
    start_time ← current time
    print("Solving using DFS:")
```



```
DATE:                PAGE:

dfs_solution ← DFS(initial_state)
end_time ← current time
if dfs_solution is not null then
    print("DFS Solution:", dfs_solution)
else
    print("No solution found")
print("Time taken by DFS:", end_time - start_ti
```

STATE SPACE TREE



8.10.24

DATE: PAGE:

LABORATORY PROGRAM - 2(A)

STATE SPACE TREE FOR EIGHT PUZZLE PROBLEM USING DFS.

1	2	3
4	5	6
7	8	0

Goal State

Initial State

1	2	3
4	5	6
7	0	8

Up

1	2	3
4	0	6
7	5	8

Left

1	2	3
4	5	6
0	7	8

Right

1	2	3
4	5	6
7	8	0

Goal Reached.

DFS Solution: ['right']

CODE

```
from collections import deque
import time

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

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

def move(state, direction):
    new_state = [row[:] for row in state]
    zero_pos = find_zero(state)
    i, j = zero_pos

    if direction == "up" and i > 0:
        new_state[i][j], new_state[i-1][j] = new_state[i-1][j], new_state[i][j]
    elif direction == "down" and i < 2:
        new_state[i][j], new_state[i+1][j] = new_state[i+1][j], new_state[i][j]
    elif direction == "left" and j > 0:
        new_state[i][j], new_state[i][j-1] = new_state[i][j-1], new_state[i][j]
    elif direction == "right" and j < 2:
        new_state[i][j], new_state[i][j+1] = new_state[i][j+1], new_state[i][j]
    else:
        return None

    return new_state

def is_goal(state):
    return state == goal_state

def print_state(state):
    for row in state:
        print(row)
    print("\n")

def dfs(initial_state):
    stack = [(initial_state, [])]
    visited = set()

    while stack:
        state, path = stack.pop()
        print("Exploring state in DFS:")
        print_state(state)

        if is_goal(state):
            return path
```

```
            visited.add(str(state))
            for direction in ["up", "down", "left", "right"]:
                new_state = move(state, direction)
                if new_state and str(new_state) not in visited:
                    stack.append((new_state, path + [direction]))

    return None

def get_initial_state():
    print("Enter the initial state of the 8-puzzle (0 for empty space):")
    initial_state = []
    for i in range(3):
        row = list(map(int, input(f"Enter row {i+1} (space-separated): ").strip().split()))
        if len(row) != 3:
            raise ValueError("Each row must contain exactly 3 numbers.")
        initial_state.append(row)
    return initial_state

if __name__ == "__main__":
    initial_state = get_initial_state()

    print("Initial State:")
    print_state(initial_state)

    start_time_dfs = time.time()
    print("Solving using DFS:")
    dfs_solution = dfs(initial_state)
    end_time_dfs = time.time()
    if dfs_solution:
        print("DFS Solution:", dfs_solution)
    else:
        print("No solution found with DFS.")
    print(f"Time taken by DFS: {end_time_dfs - start_time_dfs:.6f} seconds")
```

OUTPUT

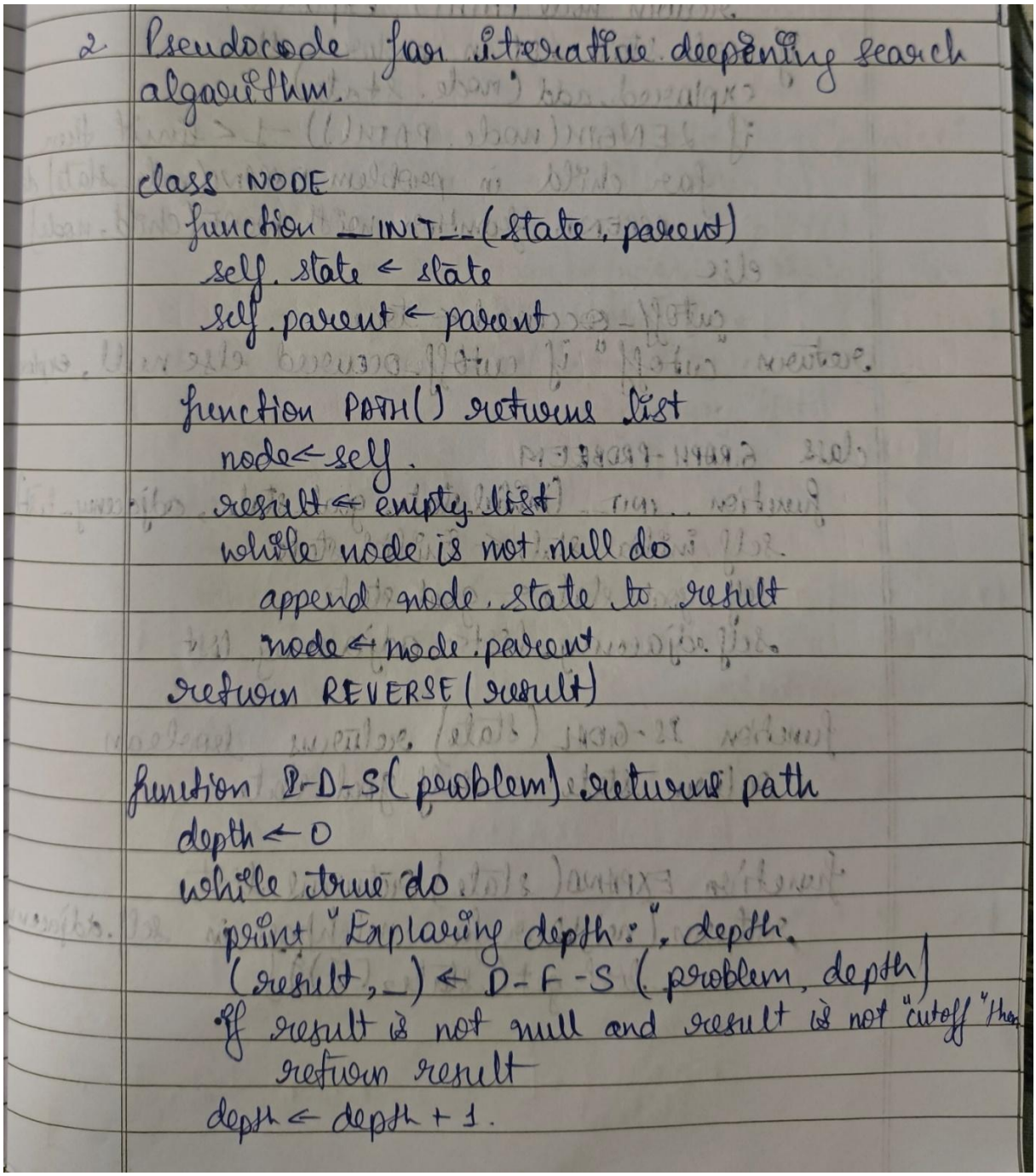
```
Enter the initial state of the 8-puzzle (0 for empty space):
Enter row 1 (space-separated): 1 2 3
Enter row 2 (space-separated): 4  6 5
Enter row 3 (space-separated): 8 7 0
Initial State:
[1, 2, 3]
[4, 6, 5]
[8, 7, 0]


Solving using DFS:
Exploring state in DFS:
[1, 2, 3]
[4, 6, 5]
[8, 7, 0]


Exploring state in DFS:
[1, 2, 3]
[4, 6, 5]
[8, 0, 7]


Exploring state in DFS:
[1, 2, 3]
[4, 6, 5]
[0, 8, 7]


Exploring state in DFS:
[1, 2, 3]
[0, 6, 5]
[4, 8, 7]
```

```
 Exploring state in DFS:
 [1, 2, 3]
 [4, 8, 5]
 [7, 0, 6]


 Exploring state in DFS:
 [1, 2, 3]
 [4, 8, 5]
 [7, 6, 0]


 Exploring state in DFS:
 [1, 2, 3]
 [4, 0, 5]
 [7, 8, 6]


 Exploring state in DFS:
 [1, 2, 3]
 [4, 5, 0]
 [7, 8, 6]


 Exploring state in DFS:
 [1, 2, 3]
 [4, 5, 6]
 [7, 8, 0]


 DFS Solution: ['left', 'left', 'up', 'right', 'right', 'down', 'left', 'le
 Time taken by DFS: 0.183756 seconds
```

LABORATORY PROGRAM – 3

Implement Iterative deepening search algorithm

PSEUDOCODE OR ALGORITHM



2 Pseudocode for iterative deepening search algorithm.

```
class NODE
    function __INIT__(state, parent)
        self.state ← state
        self.parent ← parent

    function PATH() returns list
        node ← self.
        result ← empty list
        while node is not null do
            append node.state to result
            node ← node.parent
        return REVERSE(result)

function I-D-S(problem) returns path
    depth ← 0
    while true do
        print "Exploring depth:", depth.
        (result, _) ← D-F-S(problem, depth)
        if result is not null and result is not "cutoff" then
            return result
        depth ← depth + 1.
```

DATE: PAGE:

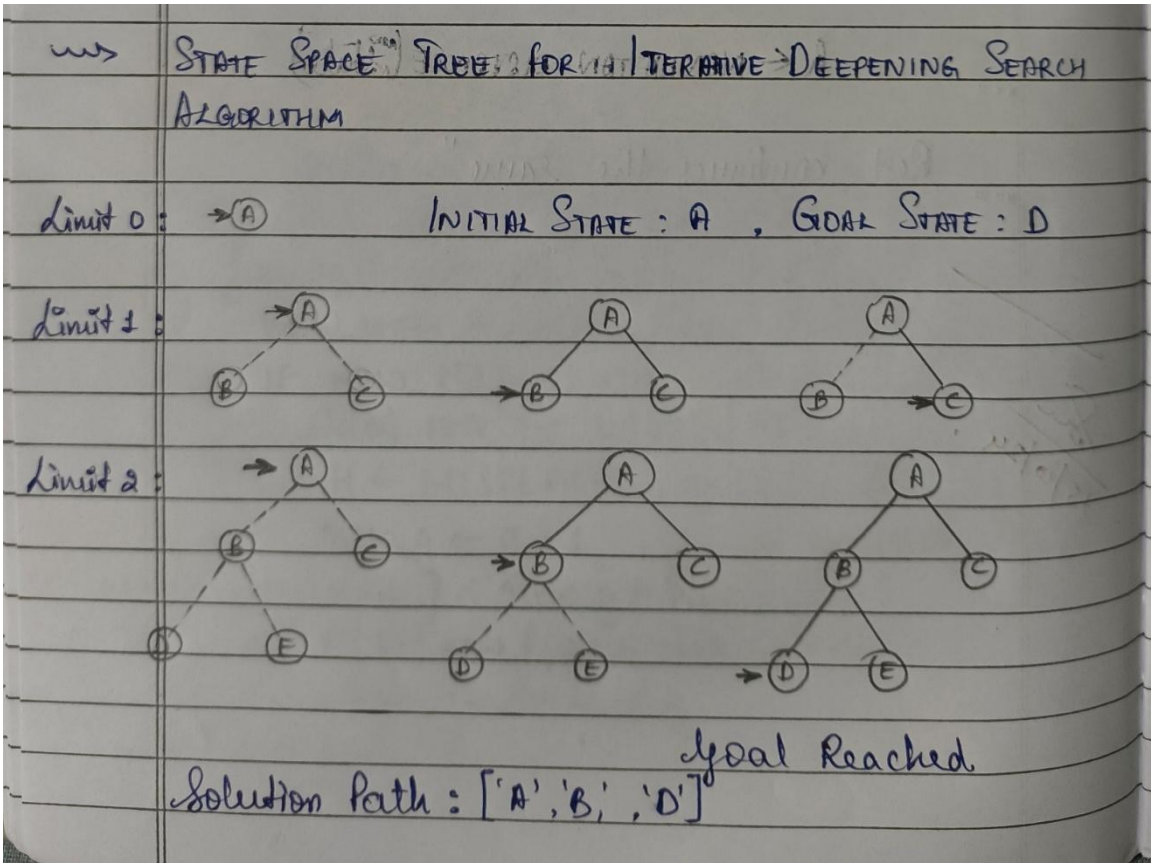
```
function D-F-S(problem, limit) returns path or cutoff
    frontier ← [NODE(problem.initial_state)]
    explored ← empty_set
    cutoff_occured ← false
    while frontier is not empty do
        node ← POP(frontier)
        if problem.IS-GOAL(node.state) then
            return node.PATH(), explored.
        if node.state not in explored then
            explored.add(node.state)
            if LENGTH(node.PATH()) - 1 < limit then
                for child in problem.EXPAND(node.state) do
                    APPEND frontier with NODE(child, node)
            else
                cutoff_occured ← true
    return "cutoff" if cutoff_occured else null, explored
```

```
class GRAPH-PROBLEM
    function __INIT__(initial_state, goal_state, adjacency_list)
        self.initial_state ← initial_state
        self.goal_state ← goal_state
        self.adjacency_list ← adjacency_list

    function IS-GOAL(state) returns boolean
        return state == self.goal_state

    function EXPAND(state) returns list,
        return [neighbor for neighbor in self.adjacency
            _list.get(state, [])].
```

DATE: PAGE:

```
function GET-GRAPH-FROM-INPUT() returns GRAPH-PROBLEM
    adjacency_list ← empty_dictionary
    initial_state ← INPUT("Enter the initial state:").strip()
    goal_state ← INPUT("Enter the goal state:").strip()
    print "Enter the AD list for graph."
    print "Type 'done' when finished."
    while true do
        node ← INPUT("Enter node:").strip()
        if node.lower() == "done" then
            break
        neighbors_input ← INPUT("Enter N of "+node+"
        separated by spaces:").strip()
        neighbors ← neighbors_input.split()
        adjacency_list[node] ← [neighbor.strip() for
            neighbor in neighbors]
    return GRAPH-PROBLEM(initial_state, goal_state,
                         adjacency_list)

main
    problem ← GET-GRAPH-FROM-INPUT()
    solution ← I-D-S(problem)
    if solution is not null then
        print "Solution Path:", solution
    else
        print "No solution found."
```

STATE SPACE TREE



CODE

```
class Node:
    def __init__(self, state, parent=None):
        self.state = state
        self.parent = parent

    def path(self):
        node, result = self, []
        while node:
            result.append(node.state)
            node = node.parent
        return result[::-1]

def iterative_deepening_search(problem):
    depth = 0
    while True:
        print(f"Exploring depth: {depth}")
        result, _ = depth_limited_search(problem, depth)
        if result is not None and result != 'cutoff':
            return result
        depth += 1

def depth_limited_search(problem, limit):
    frontier = [Node(problem.initial_state)]
    explored = set()
    cutoff_occurred = False

    while frontier:
        node = frontier.pop()

        if problem.is_goal(node.state):
            return node.path(), explored

        if node.state not in explored:
            explored.add(node.state)
            if len(node.path()) - 1 < limit:
                for child in problem.expand(node.state):
                    frontier.append(Node(child, node))
            else:
                cutoff_occurred = True

    return 'cutoff' if cutoff_occurred else None, explored

class GraphProblem:
    def __init__(self, initial_state, goal_state, adjacency_list):
        self.initial_state = initial_state
        self.goal_state = goal_state
        self.adjacency_list = adjacency_list

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

    def expand(self, state):
        return [neighbor for neighbor in self.adjacency_list.get(state, [])]
```

```
def get_graph_from_input():
    adjacency_list = {}
    initial_state = input("Enter the initial state: ").strip()
    goal_state = input("Enter the goal state: ").strip()

    print("Enter the adjacency list for the graph (neighbors of each node).")
    print("Type 'done' when finished.")

    while True:
        node = input("Enter node (or 'done' to finish): ").strip()
        if node.lower() == 'done':
            break
        neighbors_input = input(f"Enter neighbors of {node} separated by spaces: ").strip()
        neighbors = neighbors_input.split()
        adjacency_list[node] = [neighbor.strip() for neighbor in neighbors]

    return GraphProblem(initial_state, goal_state, adjacency_list)

if __name__ == "__main__":
    problem = get_graph_from_input()
    solution = iterative_deepening_search(problem)

    if solution:
        print("Solution Path:", solution)
    else:
        print("No solution found.")
```

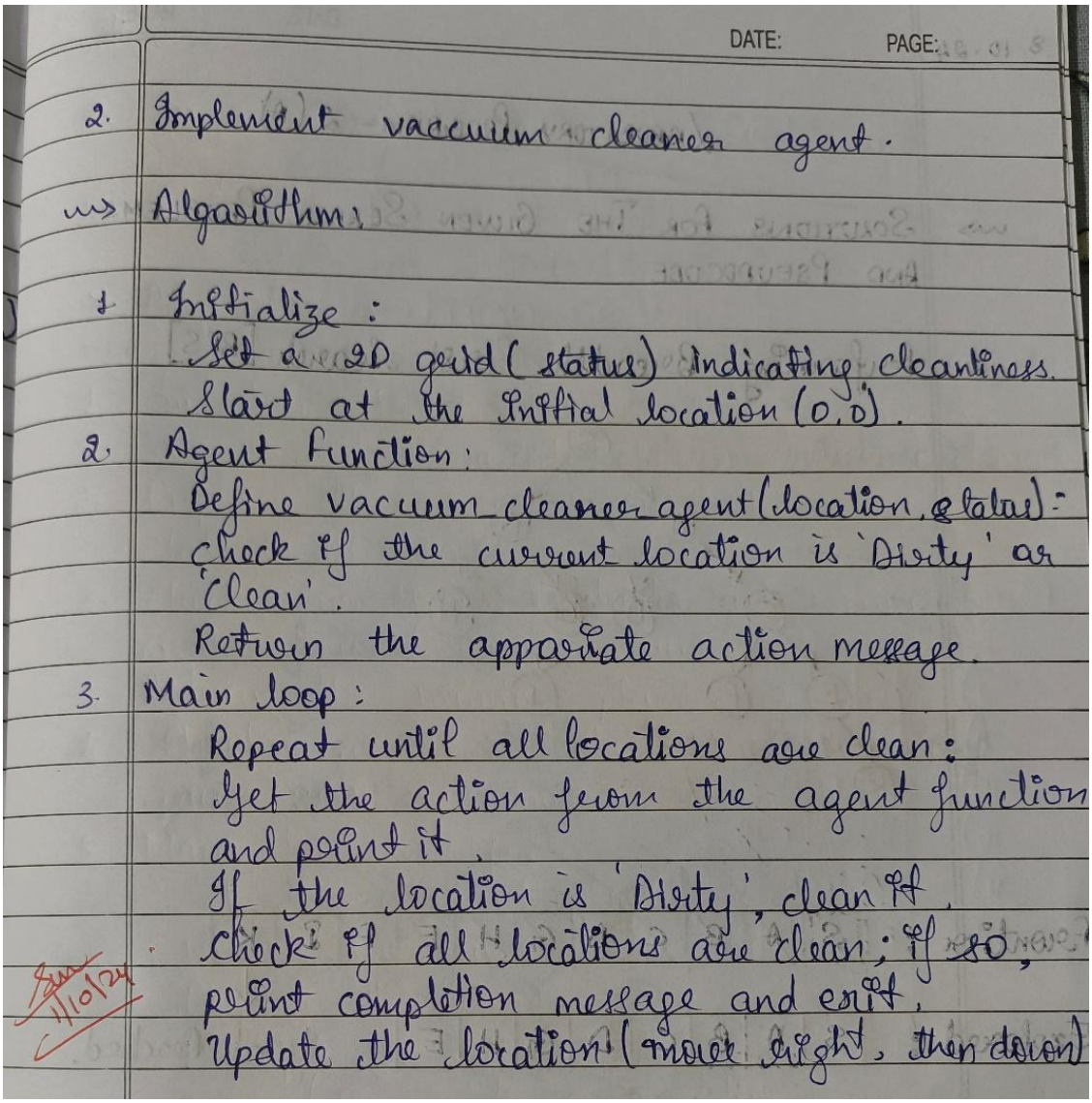
OUTPUT

```
Enter the initial state: A
Enter the goal state: G
Enter the adjacency list for the graph (neighbors of each node).
Type 'done' when finished.
Enter node (or 'done' to finish): A
Enter neighbors of A separated by spaces: B C
Enter node (or 'done' to finish): B
Enter neighbors of B separated by spaces: D E
Enter node (or 'done' to finish): D
Enter neighbors of D separated by spaces: H I
Enter node (or 'done' to finish): C
Enter neighbors of C separated by spaces: F G
Enter node (or 'done' to finish): F
Enter neighbors of F separated by spaces: K
Enter node (or 'done' to finish): done
Exploring depth: 0
Exploring depth: 1
Exploring depth: 2
Solution Path: ['A', 'C', 'G']
```

LABORATORY PROGRAM – 4

Implement vacuum cleaner agent

PSEUDOCODE OR ALGORITHM



DATE: PAGE:

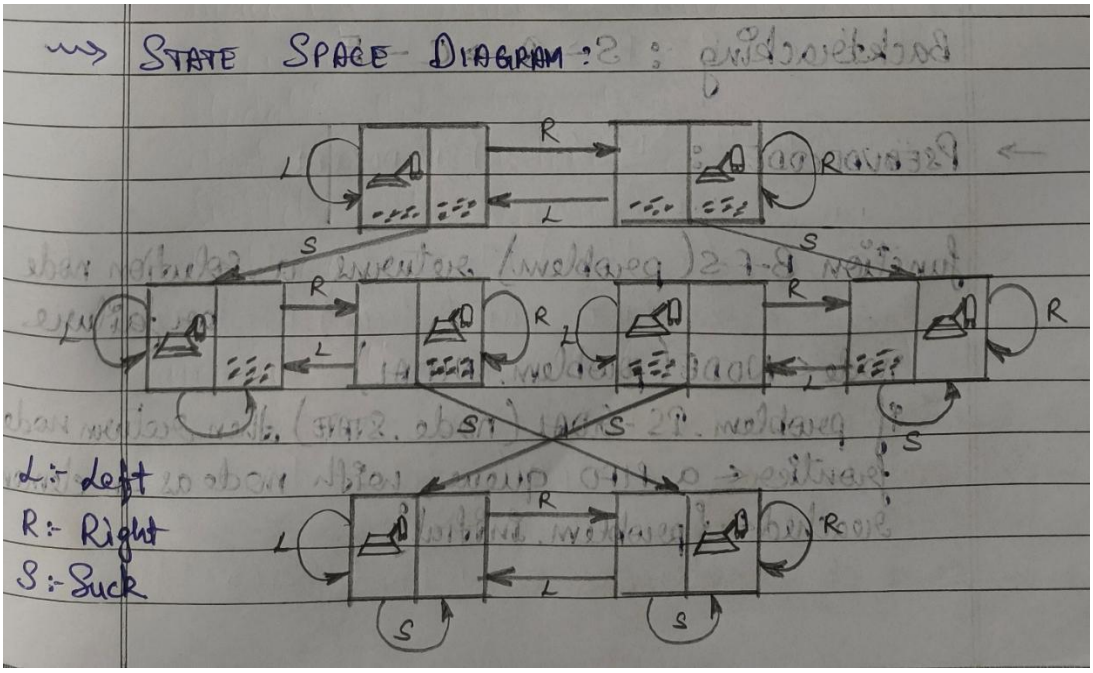
2. Implement vaccuum cleaner agent.

=> Algorithm:

1. Initialize:
   Set a 2D grid (status) indicating cleanliness.
   Start at the initial location (0,0).
2. Agent Function:
   Define vacuum_cleaner_agent(location, status):
   Check if the current location is 'Dirty' or 'Clean'.
   Return the appropriate action message.
3. Main loop:
   Repeat until all locations are clean:
   Get the action from the agent function and print it.
   If the location is 'Dirty', clean it.
   Check if all locations are clean; if so, print completion message and exit.
   Update the location (move right, then down)

8 11/10/24

STATE SPACE TREE



CODE

```
def vacuum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter the initial location of the vacuum cleaner (A or B): ")
    status_input = input(f"Enter the status of room {location_input} (0 for Clean, 1 for Dirty): ")
    status_input_complement = input(f"Enter the status of the other room ({'B' if location_input == 'A'
else 'A'}) (0 for Clean, 1 for Dirty): ")

    # Set the initial state based on user input
    initial_state = {
        'A': status_input if location_input == 'A' else status_input_complement,
        'B': status_input_complement if location_input == 'A' else status_input
    }

    print("\nInitial Location Condition:", initial_state)

    if location_input == 'A':
        print("\nVacuum cleaner is placed in room A.")

        if status_input == '1':
            print("Room A is Dirty. Cleaning room A...")
            goal_state['A'] = '0'
            cost += 1
            print("Cost for cleaning room A:", cost)

            if status_input_complement == '1':
                print("\nRoom B is also Dirty. Moving to room B...")
                cost += 1
                print("Cost for moving to room B:", cost)
                print("Cleaning room B...")
                goal_state['B'] = '0'
                cost += 1
                print("Cost for cleaning room B:", cost)
            else:
                print("\nRoom B is already Clean. No further action needed.")

        else:
            print("Room A is already Clean.")

            if status_input_complement == '1':
                print("\nRoom B is Dirty. Moving to room B...")
                cost += 1
                print("Cost for moving to room B:", cost)
                print("Cleaning room B...")
                goal_state['B'] = '0'
                cost += 1
                print("Cost for cleaning room B:", cost)
            else:
                print("\nRoom B is already Clean. No further action needed.")

    else:
        print("\nVacuum cleaner is placed in room B.")
```

```
        if status_input == '1':
            print("Room B is Dirty. Cleaning room B...")
            goal_state['B'] = '0'
            cost += 1
            print("Cost for cleaning room B:", cost)

            if status_input_complement == '1':
                print("\nRoom A is also Dirty. Moving to room A...")
                cost += 1
                print("Cost for moving to room A:", cost)
                print("Cleaning room A...")
                goal_state['A'] = '0'
                cost += 1
                print("Cost for cleaning room A:", cost)
            else:
                print("\nRoom A is already Clean. No further action needed.")

        else:
            print("Room B is already Clean.")

            if status_input_complement == '1':
                print("\nRoom A is Dirty. Moving to room A...")
                cost += 1
                print("Cost for moving to room A:", cost)
                print("Cleaning room A...")
                goal_state['A'] = '0'
                cost += 1
                print("Cost for cleaning room A:", cost)
            else:
                print("\nRoom A is already Clean. No further action needed.")

    print("\nGOAL STATE:", goal_state)
    print("Total cost for cleaning:", cost)

vacuum_world()
```

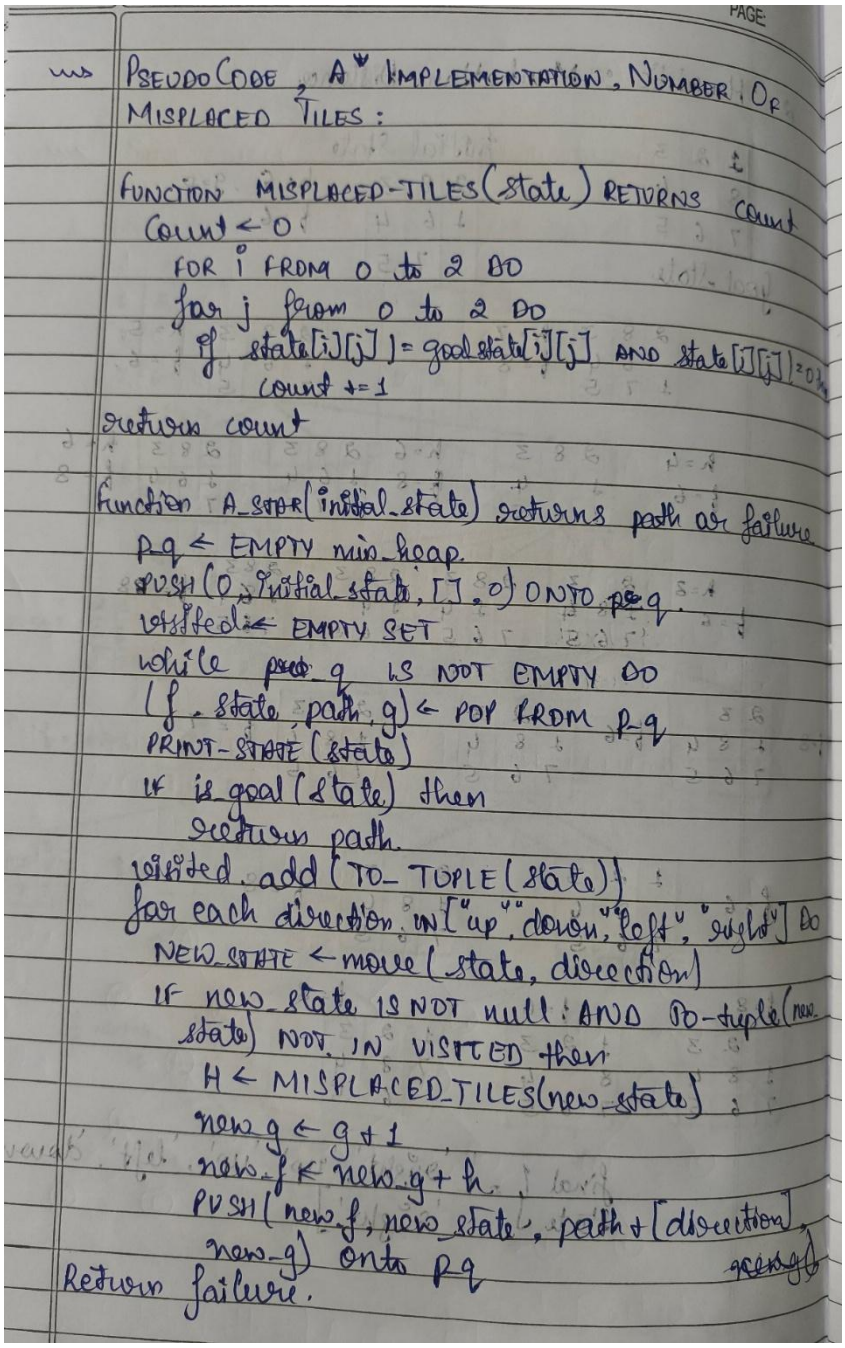
OUTPUT

```
Enter the initial location of the vacuum cleaner (A or B): A
Enter the status of room A (0 for Clean, 1 for Dirty): 1
Enter the status of the other room (B) (0 for Clean, 1 for Dirty): 1

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

Vacuum cleaner is placed in room A.
Room A is Dirty. Cleaning room A...
Cost for cleaning room A: 1

Room B is also Dirty. Moving to room B...
Cost for moving to room B: 2
Cleaning room B...
Cost for cleaning room B: 3

GOAL STATE: {'A': '0', 'B': '0'}
Total cost for cleaning: 3
```

LABORATORY PROGRAM – 5(A)

Implement A* search algorithm

PSEUDOCODE OR ALGORITHM



```
PSEUDO CODE, A* IMPLEMENTATION, NUMBER OF
MISPLACED TILES:

FUNCTION MISPLACED-TILES(state) RETURNS count
  count ← 0
    FOR i FROM 0 to 2 DO
    for j from 0 to 2 DO
      if state[i][j] = goal state[i][j] AND state[i][j] ≠ 0
        count += 1
return count

function A_star(initial_state) returns path or failure
  pq ← EMPTY min_heap
  PUSH (0, initial_state, [], 0) ONTO pq
  visited ← EMPTY SET
  while pq IS NOT EMPTY DO
  (f, state, path, g) ← POP FROM pq
  PRINT-STATE (state)
  IF is_goal (state) then
    return path
  visited.add (TO_TUPLE (state))
  for each direction in ["up", "down", "left", "right"] DO
    NEW_STATE ← move (state, direction)
    IF new state IS NOT null AND to-tuple(new
    state) NOT IN VISITED then
      H ← MISPLACED_TILES(new_state)
      new g ← g + 1
      new_f ← new_g + h
      PUSH (new_f, new_state, path + [direction],
      new_g) onto pq
Return failure.
```

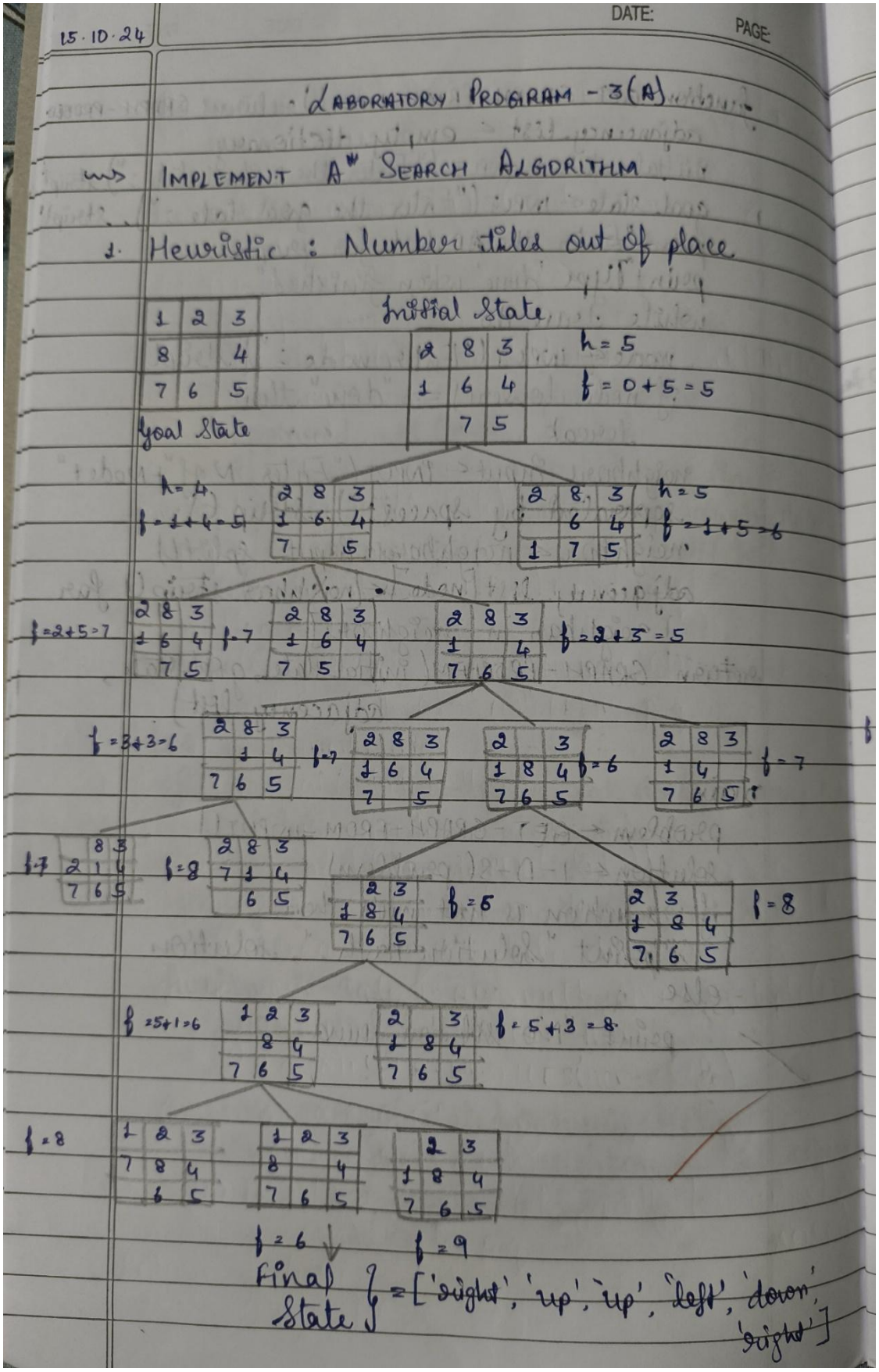
DATE: PAGE:

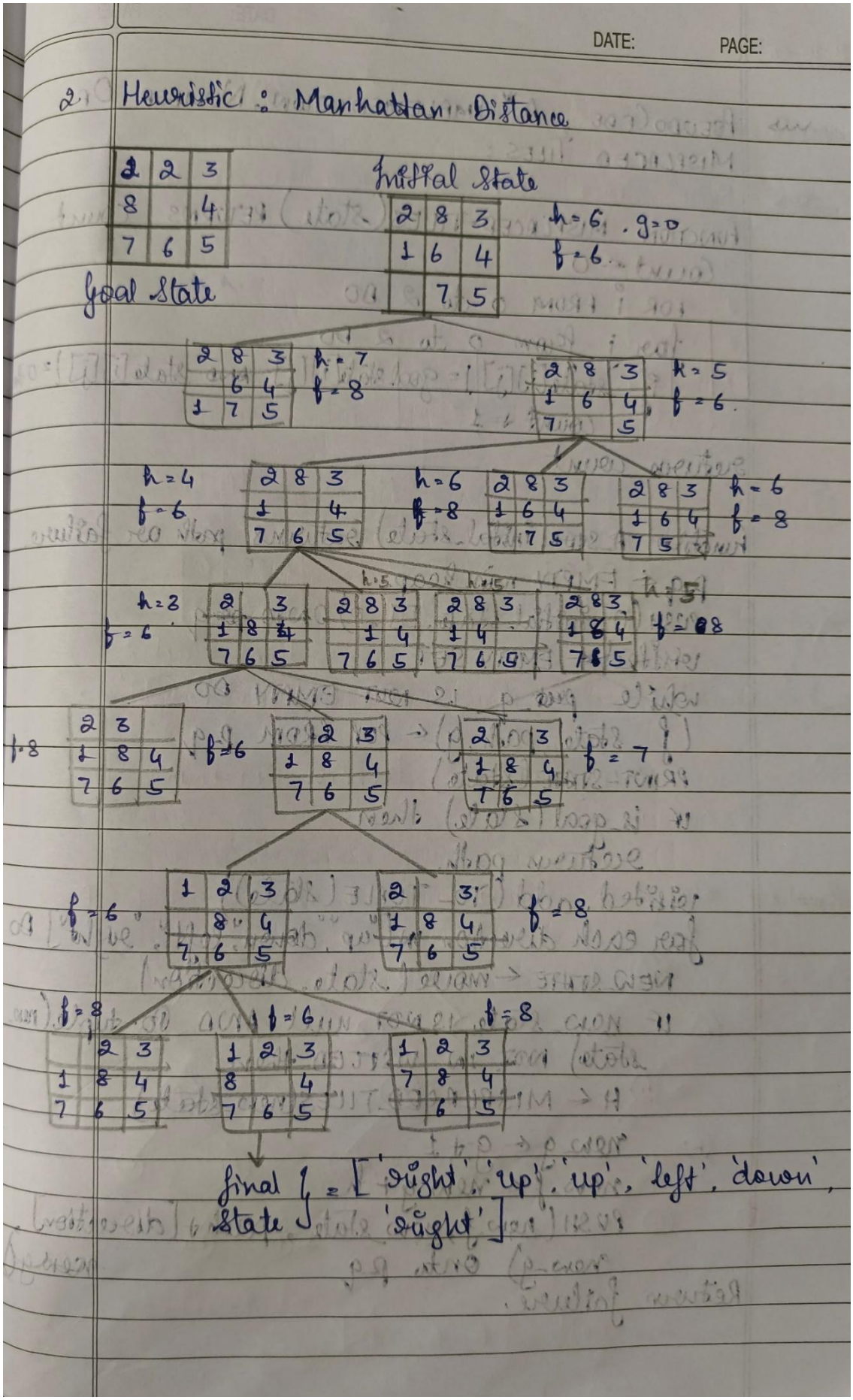
Pseudo Code, A* Implementation, Manhattan Distance

```
Function MANHATTAN-DISTANCE(state) returns distance
    Distance ← 0
    for i from 0 to 2 do
        for j from 0 to 2 do
            if state[i][j] != 0 then
                goal_i, goal_j ← position of state[i][j] in goal_state
                if state[i][j] == 8 then
                    goal_i, goal_j ← 1, 1
                Distance += ABS(goal_i - i) + ABS(goal_j - j)
    return distance

Function A_STAR(initial_state) returns path or failure
    Similar to the pseudocode of
    misplaced_tiles, but change in heuristic functio

        h ← MANHATTAN-DISTANCE(node.state)

    Rest continues the same
```

15/10/24.

STATE SPACE TREE




DATE:
PAGE:
2. Heuristic : Manhattan Distance
Goal State
Initial State
final State = ['right', 'up', 'up', 'left', 'down', 'right']

CODE - G(N): DEPTH OF THE NODE, H(N): NUMBER OF MISPLACED TILES

```
import heapq

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

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

def move(state, direction):
    new_state = [row[:] for row in state]
    i, j = find_zero(state)
    if direction == "up" and i > 0:
        new_state[i][j], new_state[i-1][j] = new_state[i-1][j], new_state[i][j]
    elif direction == "down" and i < 2:
        new_state[i][j], new_state[i+1][j] = new_state[i+1][j], new_state[i][j]
    elif direction == "left" and j > 0:
        new_state[i][j], new_state[i][j-1] = new_state[i][j-1], new_state[i][j]
    elif direction == "right" and j < 2:
        new_state[i][j], new_state[i][j+1] = new_state[i][j+1], new_state[i][j]
    else:
        return None
    return new_state

def is_goal(state):
    return state == goal_state

def misplaced_tiles(state):
    return sum(1 for i in range(3) for j in range(3)
               if state[i][j] != goal_state[i][j] and state[i][j] != 0)

def a_star(initial_state):
    priority_queue = []
    heapq.heappush(priority_queue, (0, initial_state, [], 0))
    visited = set()

    while priority_queue:
        f, state, path, g = heapq.heappop(priority_queue)
        print("Exploring state in A*:")
        print_state(state)

        if is_goal(state):
            return path

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

```
        for direction in ["up", "down", "left", "right"]:
            new_state = move(state, direction)
            if new_state and tuple(map(tuple, new_state)) not in visited:
                h = misplaced_tiles(new_state)
                new_g = g + 1
                new_f = new_g + h
                heapq.heappush(priority_queue, (new_f, new_state, path + [direction], new_g))

    return None

def print_state(state):
    for row in state:
        print(row)
    print()

def get_initial_state():
    print("Enter the initial state of the 8-puzzle (0 for empty space):")
    initial_state = []
    for i in range(3):
        row = list(map(int, input(f"Enter row {i+1} (space-separated): ").strip().split()))
        if len(row) != 3:
            raise ValueError("Each row must contain exactly 3 numbers.")
        initial_state.append(row)
    return initial_state

if __name__ == "__main__":
    initial_state = get_initial_state()
    print("Initial State:")
    print_state(initial_state)
    print("Solving using A* search:")
    a_star_solution = a_star(initial_state)

    if a_star_solution:
        print("A* Solution:", a_star_solution)
    else:
        print("No solution found with A*.")
```

OUTPUT - G(N): DEPTH OF THE NODE, H(N): NUMBER OF MISPLACED TILES

```
Enter the initial state of the 8-puzzle (0 for empty space):
Enter row 1 (space-separated): 2 8 3
Enter row 2 (space-separated): 1 6 4
Enter row 3 (space-separated): 0 7 5
Initial State:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Solving using A* search:
Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

Exploring state in A*:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

Exploring state in A*:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

Exploring state in A*:
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

Exploring state in A*:
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

Exploring state in A*:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

A* Solution: ['right', 'up', 'up', 'left', 'down', 'right']
```

CODE - G(N): DEPTH OF THE NODE, H(N): MANHATTAN DISTANCE

```
import heapq

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

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

def move(state, direction):
    new_state = [row[:] for row in state]
    i, j = find_zero(state)
    if direction == "up" and i > 0:
        new_state[i][j], new_state[i-1][j] = new_state[i-1][j], new_state[i][j]
    elif direction == "down" and i < 2:
        new_state[i][j], new_state[i+1][j] = new_state[i+1][j], new_state[i][j]
    elif direction == "left" and j > 0:
        new_state[i][j], new_state[i][j-1] = new_state[i][j-1], new_state[i][j]
    elif direction == "right" and j < 2:
        new_state[i][j], new_state[i][j+1] = new_state[i][j+1], new_state[i][j]
    else:
        return None
    return new_state

def is_goal(state):
    return state == goal_state

def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                goal_i, goal_j = divmod(state[i][j] - 1, 3)
                if state[i][j] == 8:
                    goal_i, goal_j = 1, 1
                distance += abs(goal_i - i) + abs(goal_j - j)
    return distance

def a_star(initial_state):
    priority_queue = []
    heapq.heappush(priority_queue, (0, initial_state, [], 0))
    visited = set()

    while priority_queue:
        f, state, path, g = heapq.heappop(priority_queue)
        print("Exploring state in A*:")
        print_state(state)

        if is_goal(state):
            return path
```

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

        for direction in ["up", "down", "left", "right"]:
            new_state = move(state, direction)
            if new_state and tuple(map(tuple, new_state)) not in visited:
                h = manhattan_distance(new_state)
                new_g = g + 1
                new_f = new_g + h
                heapq.heappush(priority_queue, (new_f, new_state, path + [direction], new_g))

    return None

def print_state(state):
    for row in state:
        print(row)
    print()

def get_initial_state():
    print("Enter the initial state of the 8-puzzle (0 for empty space):")
    initial_state = []
    for i in range(3):
        row = list(map(int, input(f"Enter row {i+1} (space-separated): ").strip().split()))
        if len(row) != 3:
            raise ValueError("Each row must contain exactly 3 numbers.")
        initial_state.append(row)
    return initial_state

if __name__ == "__main__":
    initial_state = get_initial_state()
    print("Initial State:")
    print_state(initial_state)
    print("Solving using A* search:")
    a_star_solution = a_star(initial_state)

    if a_star_solution:
        print("A* Solution:", a_star_solution)
    else:
        print("No solution found with A*.")
```

OUTPUT - G(N): DEPTH OF THE NODE, H(N): MANHATTAN DISTANCE

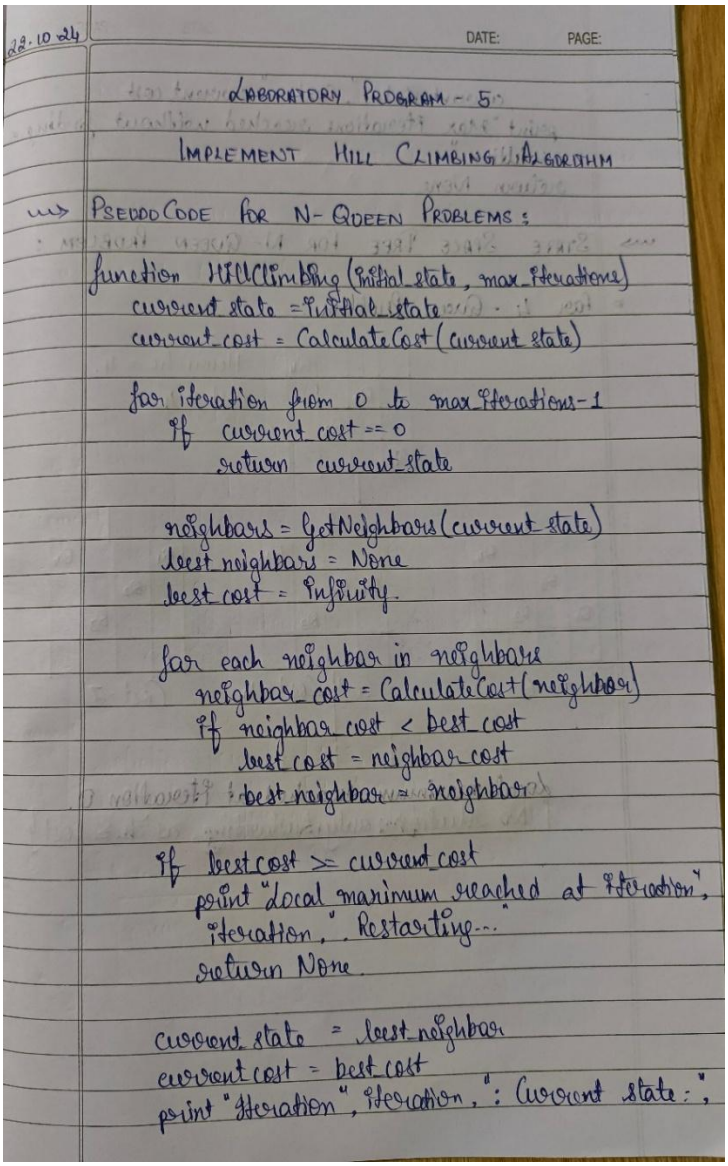
```
Enter the initial state of the 8-puzzle (0 for empty space):
Enter row 1 (space-separated): 2 8 3
Enter row 2 (space-separated): 1 6 4
Enter row 3 (space-separated): 0 7 5
Initial State:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Solving using A* search:
Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[7, 5, 0]

Exploring state in A*:
[2, 8, 3]
[0, 6, 4]
[1, 7, 5]

Exploring state in A*:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
```

```
Exploring state in A*:
[2, 8, 3]
[1, 5, 6]
[7, 4, 0]

Exploring state in A*:
[0, 8, 3]
[2, 6, 4]
[1, 7, 5]

Exploring state in A*:
[1, 2, 3]
[7, 8, 4]
[0, 6, 5]

Exploring state in A*:
[1, 2, 3]
[7, 8, 6]
[0, 5, 4]

Exploring state in A*:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

A* Solution: ['right', 'up', 'up', 'left', 'down', 'right']
```

LABORATORY PROGRAM – 5(B)

Implement Hill Climbing Algorithm

PSEUDOCODE OR ALGORITHM



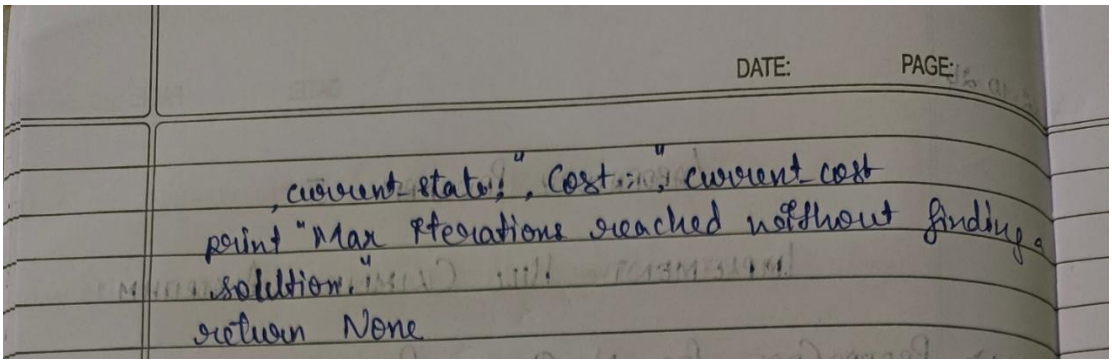
28.10.24 DATE: PAGE:

LABORATORY PROGRAM - 5

IMPLEMENT HILL CLIMBING ALGORITHM

⇒ PSEUDOCODE FOR N-QUEEN PROBLEMS:

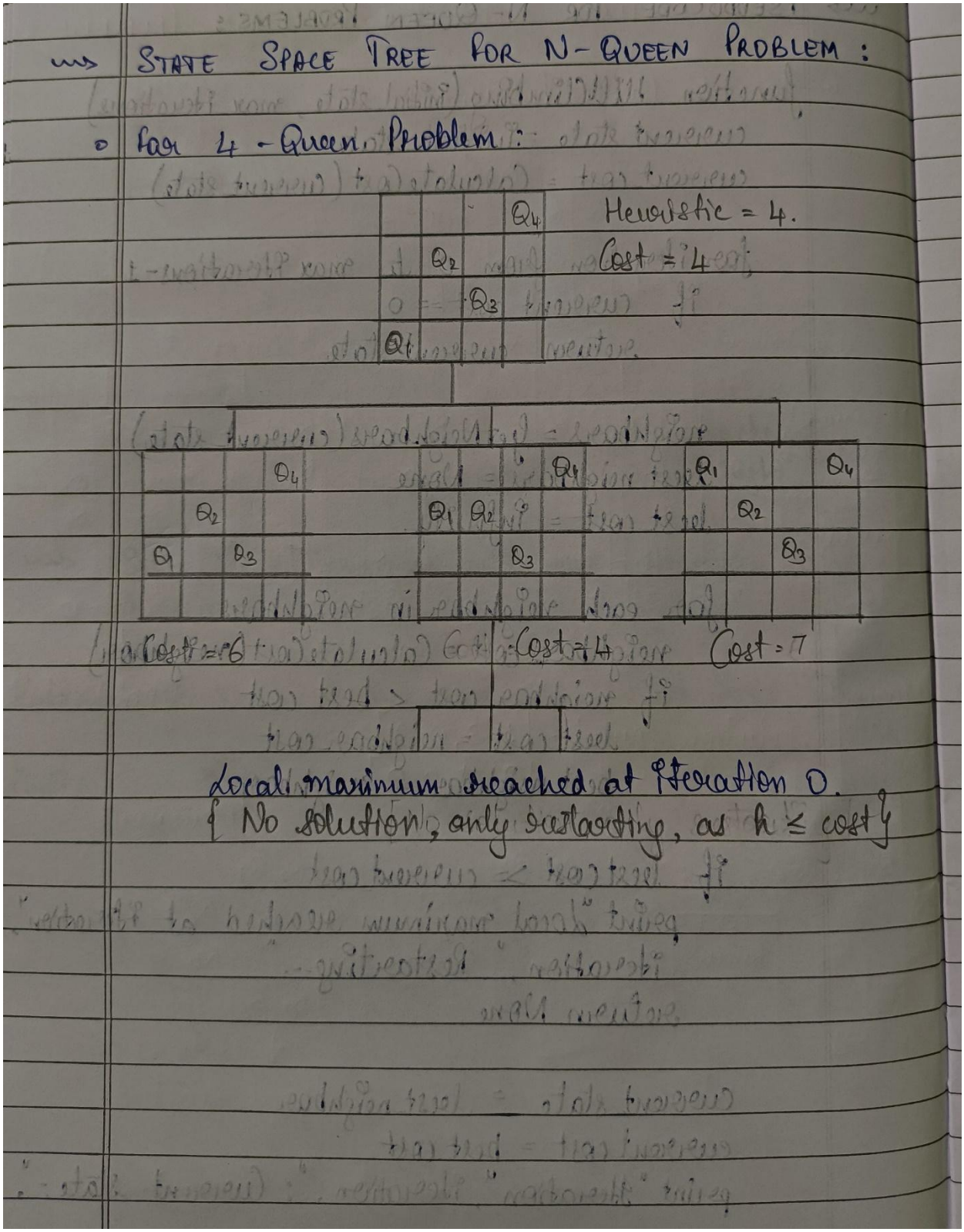
```
function HillClimbing(initial_state, max_iterations)
    current state = initial_state
    current_cost = CalculateCost(current state)

    for iteration from 0 to max_iterations-1
        if current_cost == 0
            return current_state

        neighbors = GetNeighbors(current state)
        best neighbors = None
        best cost = Infinity.

        for each neighbor in neighbors
            neighbor_cost = CalculateCost(neighbor)
            if neighbor cost < best_cost
                best cost = neighbor_cost
                best_neighbor = neighbor

        if best cost >= current_cost
            print "Local maximum reached at iteration",
            "iteration.", Restarting...
            return None.

        current state = best_neighbor
        current cost = best cost
        print "Iteration", iteration, ": Current state:",
```



DATE: PAGE:

```
current_state:", Cost:", current_cost
print "Max iterations reached without finding a
solution."
return None
```

STATE SPACE TREE



CODE

```
import random

def calculate_cost(state):
    """Calculate the number of conflicts in the current state."""
    cost = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                cost += 1
    return cost

def get_neighbors(state):
    """Generate all possible neighbors by moving each queen in its column."""
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if state[col] != row:  # Move the queen in column `col` to a different row
                new_state = list(state)
                new_state[col] = row
                neighbors.append(new_state)
    return neighbors

def hill_climbing(initial_state, max_iterations=1000):
    """Perform hill climbing search to solve the N-Queens problem."""
    current_state = initial_state
    current_cost = calculate_cost(current_state)

    for iteration in range(max_iterations):
        if current_cost == 0:
            return current_state

        neighbors = get_neighbors(current_state)
        neighbor_costs = [(neighbor, calculate_cost(neighbor)) for neighbor in neighbors]
        next_state, next_cost = min(neighbor_costs, key=lambda x: x[1])

        if next_cost >= current_cost:
            print(f"Local maximum reached at iteration {iteration}. Restarting...")
            return None
        current_state, current_cost = next_state, next_cost
        print(f"Iteration {iteration}: Current state: {current_state}, Cost: {current_cost}")

    print(f"Max iterations reached without finding a solution.")
    return None

try:
    n = int(input("Enter the number of queens (N): "))
    if n <= 0:
        raise ValueError("N must be a positive integer.")

    initial_state = list(map(int, input(f"Enter the initial state as a list of {n} integers (rows for each
column): ").split()))
```

```
    if len(initial_state) != n or any(not (0 <= row < n) for row in initial_state):
        raise ValueError(f"Invalid initial state. Please provide {n} integers between 0 and {n-1}.")
except ValueError as e:
    print(e)
    n = 4
    initial_state = [random.randint(0, n - 1) for _ in range(n)]
    print(f"Using random initial state: {initial_state}")

solution = None

while solution is None:
    solution = hill_climbing(initial_state)

print(f"Solution found: {solution}")
```

OUTPUT

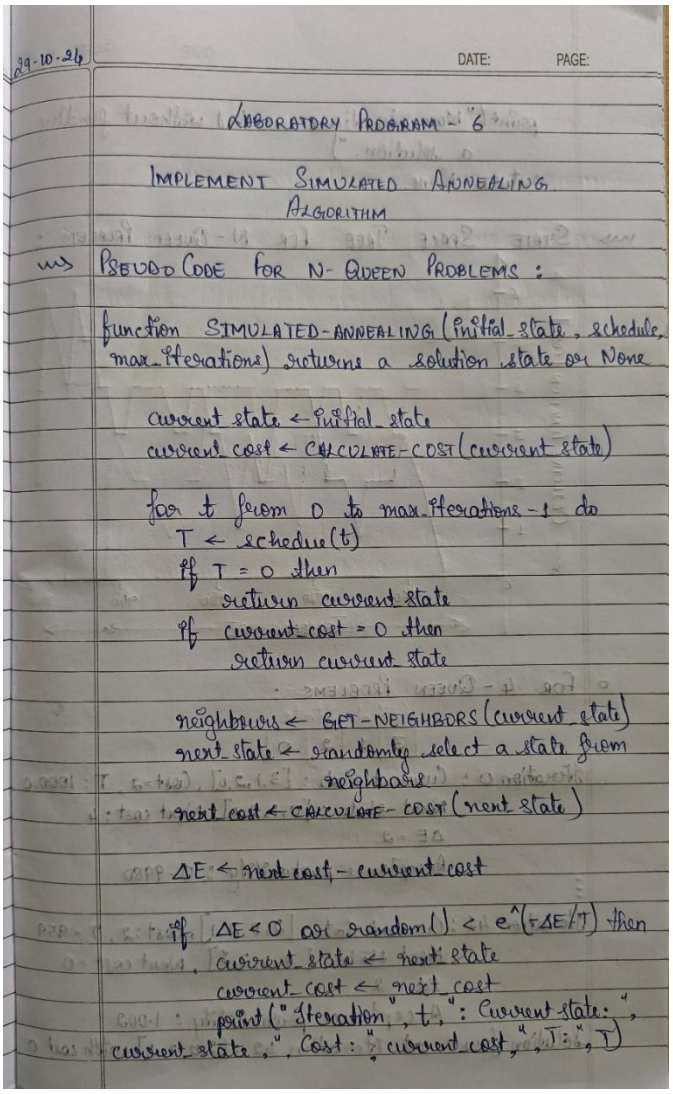
```
Enter the number of queens (N): 4
Enter the initial state as a list of 4 integers (rows for each column): 3 1 2 0
Local maximum reached at iteration 0. Restarting...
Local maximum reached at iteration 0. Restarting...
Local maximum reached at iteration 0. Restarting...
Local maximum reached at iteration 0. Restarting...
Local maximum reached at iteration 0. Restarting...
Local maximum reached at iteration 0. Restarting...
Local maximum reached at iteration 0. Restarting...
Local maximum reached at iteration 0. Restarting...
Local maximum reached at iteration 0. Restarting...
Local maximum reached at iteration 0. Restarting...
Local maximum reached at iteration 0. Restarting...
Local maximum reached at iteration 0. Restarting...
```

```
Enter the number of queens (N): 4
Enter the initial state as a list of 4 integers (rows for each column): 0 0 0 0
Iteration 0: Current state: [0, 3, 0, 0], Cost: 3
Iteration 1: Current state: [1, 3, 0, 0], Cost: 1
Iteration 2: Current state: [1, 3, 0, 2], Cost: 0
Solution found: [1, 3, 0, 2]
```

LABORATORY PROGRAM – 6

implement Simulated Annealing Algorithm

PSEUDOCODE OR ALGORITHM



29-10-24 DATE: PAGE:

LABORATORY PROGRAM - 6

IMPLEMENT SIMULATED ANNEALING ALGORITHM

PSEUDO CODE FOR N-QUEEN PROBLEMS :

function SIMULATED-ANNEALING (initial_state, schedule, max_iterations) returns a solution state or None

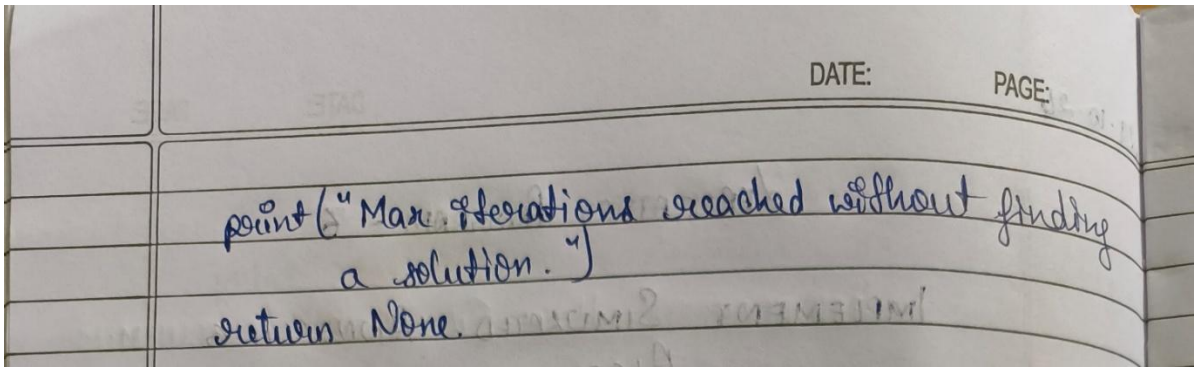
current_state ← initial_state
current_cost ← CALCULATE-COST(current_state)

for t from 0 to max_iterations - 1 do
T ← schedule(t)
if T = 0 then
return current_state
if current_cost = 0 then
return current_state

neighbours ← GET-NEIGHBORS(current_state)
next_state ← randomly select a state from neighbors
next_cost ← CALCULATE-COST(next_state)

ΔE ← next_cost - current_cost

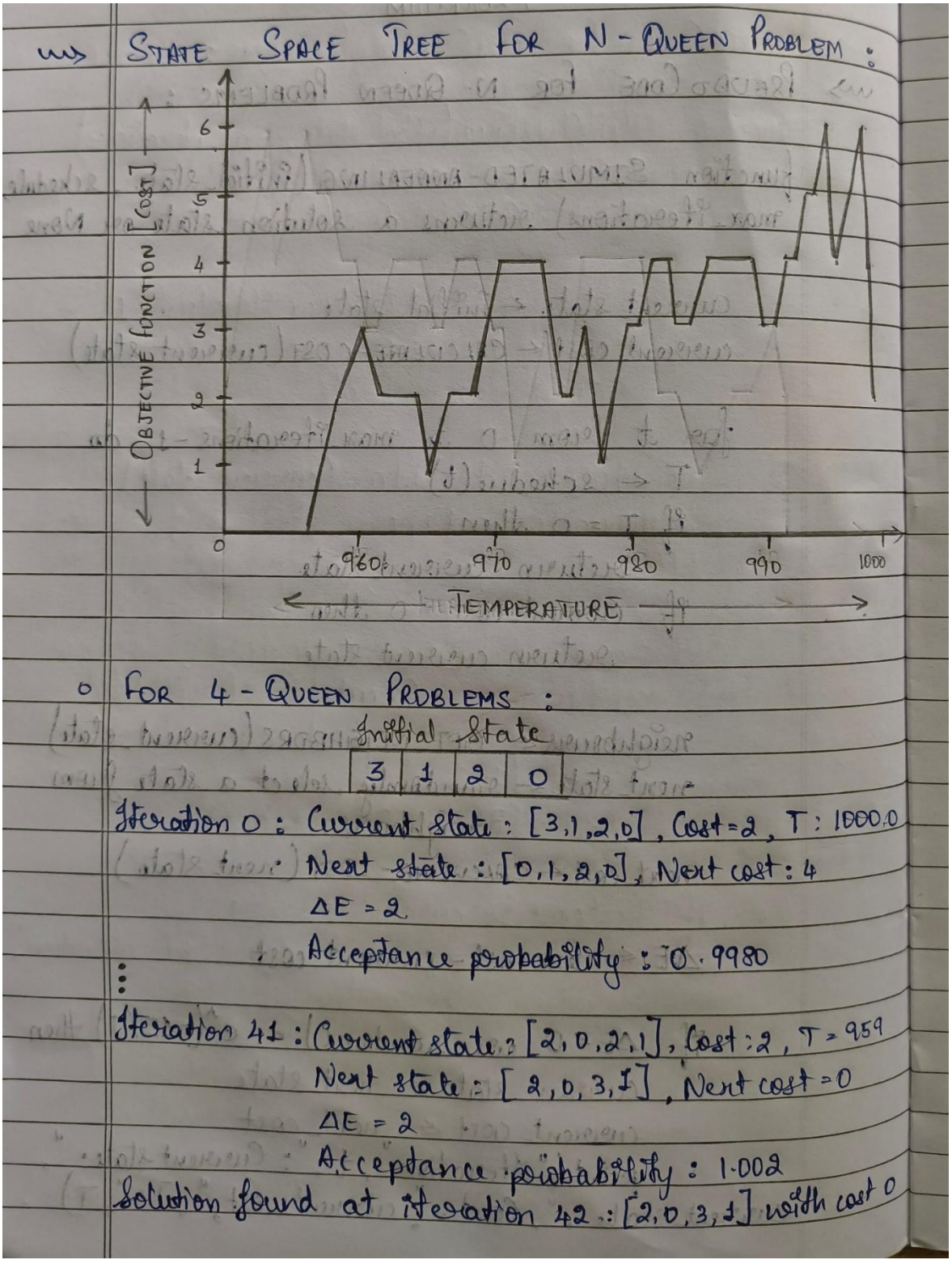
if ΔE < 0 or random() < e^(-ΔE/T) then
current_state ← next_state
current_cost ← next_cost
print("Iteration", t, ": Current state:", current_state, ". Cost:", current_cost, ", T:", T)



DATE: PAGE:

print("Max iterations reached without finding a solution.")
return None.

STATE SPACE TREE



CODE

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

def calculate_cost(state):
    cost = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                cost += 1
    return cost

def get_neighbors(state):
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if state[col] != row:
                new_state = list(state)
                new_state[col] = row
                neighbors.append(new_state)
    return neighbors

def simulated_annealing_with_tracking(initial_state, schedule, max_iterations=1000):
    current_state = initial_state
    current_cost = calculate_cost(current_state)
    costs = []
    temperatures = []

    for t in range(max_iterations):
        T = schedule(t)
        if T == 0:
            break

        if current_cost == 0:
            costs.append(current_cost)
            temperatures.append(T)
            print(f"Solution found at iteration {t}: {current_state} with cost {current_cost}")
            break

        neighbors = get_neighbors(current_state)
        next_state = random.choice(neighbors)
        next_cost = calculate_cost(next_state)

        ΔE = next_cost - current_cost
        acceptance_probability = math.exp(-ΔE / T) if T > 0 else 0
        accept = ΔE < 0 or random.random() < acceptance_probability

        print(f"Iteration {t}:")
        print(f"  Current state: {current_state}, Cost: {current_cost}, Temperature (T): {T}")
```

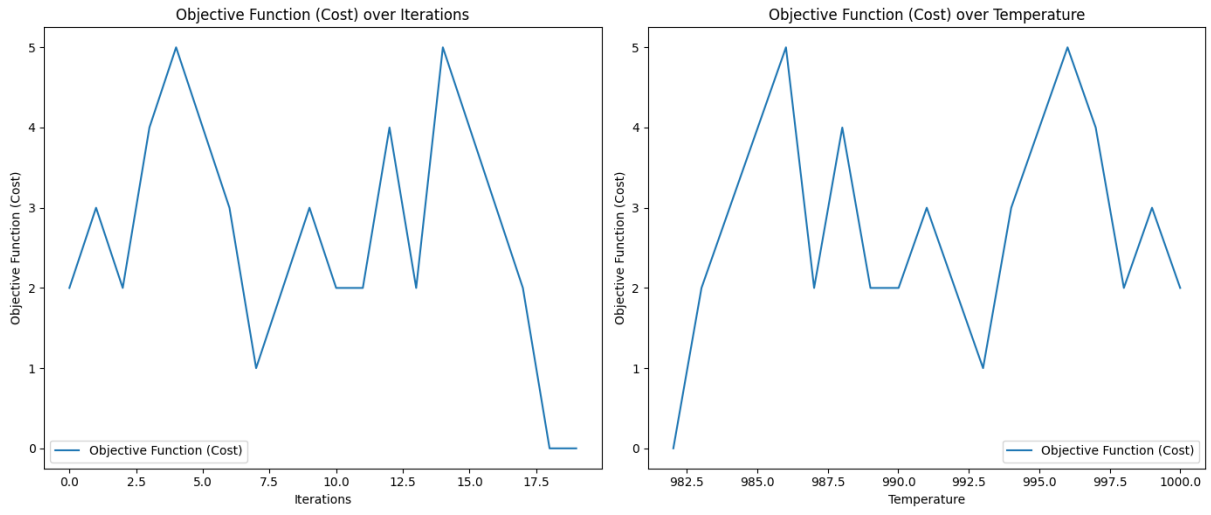
```
        print(f"  Next state: {next_state}, Next cost: {next_cost}")
        print(f"  ΔE = {ΔE}")
        print(f"  Acceptance probability: {acceptance_probability}")
        print(f"  Acceptance condition met: {accept}")

        costs.append(current_cost)
        temperatures.append(T)

        if accept:
            current_state, current_cost = next_state, next_cost

    costs.append(current_cost)
    temperatures.append(T)
    return costs, temperatures

def linear_schedule(t, initial_temp=1000, final_temp=1, max_iter=1000):
    return max(final_temp, initial_temp - (initial_temp - final_temp) * (t / max_iter))

try:
    n = int(input("Enter the number of queens (N): "))
    if n <= 0:
        raise ValueError("N must be a positive integer.")

    initial_state = list(map(int, input(f"Enter the initial state as a list of {n} integers (rows for each
column): ").split()))
    if len(initial_state) != n or any(not (0 <= row < n) for row in initial_state):
        raise ValueError(f"Invalid initial state. Please provide {n} integers between 0 and {n-1}.")
except ValueError as e:
    print(e)
    n = 4
    initial_state = [random.randint(0, n - 1) for _ in range(n)]
    print(f"Using random initial state: {initial_state}")

costs, temperatures = simulated_annealing_with_tracking(initial_state, linear_schedule)

plt.figure(figsize=(14, 6))

plt.subplot(1, 2, 1)
plt.plot(costs, label="Objective Function (Cost)")
plt.xlabel("Iterations")
plt.ylabel("Objective Function (Cost)")
plt.title("Objective Function (Cost) over Iterations")
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(temperatures, costs, label="Objective Function (Cost)")
plt.xlabel("Temperature")
plt.ylabel("Objective Function (Cost)")
plt.title("Objective Function (Cost) over Temperature")
plt.legend()

plt.tight_layout()
plt.show()

if costs[-1] == 0:
```

```
    print(f"Solution found: {initial_state}")
else:
    print("Max iterations reached without finding a solution.")
```

OUTPUT

```
Enter the number of queens (N): 4
Enter the initial state as a list of 4 integers (rows for each column): 3 1 2 0
Iteration 0:
  Current state: [3, 1, 2, 0], Cost: 2, Temperature (T): 1000.0
  Next state: [3, 1, 0, 0], Next cost: 3
  ΔE = 1
  Acceptance probability: 0.999000499833375
  Acceptance condition met: True
Iteration 1:
  Current state: [3, 1, 0, 0], Cost: 3, Temperature (T): 999.001
  Next state: [3, 1, 3, 0], Next cost: 2
  ΔE = -1
  Acceptance probability: 1.001001501166707
  Acceptance condition met: True
Iteration 2:
  Current state: [3, 1, 3, 0], Cost: 2, Temperature (T): 998.002
  Next state: [3, 3, 3, 0], Next cost: 4
  ΔE = 2
  Acceptance probability: 0.9979980026753383
  Acceptance condition met: True
Iteration 3:
  Current state: [3, 3, 3, 0], Cost: 4, Temperature (T): 997.003
  Next state: [3, 2, 3, 0], Next cost: 5
  ΔE = 1
  Acceptance probability: 0.998997496833386
  Acceptance condition met: True
```

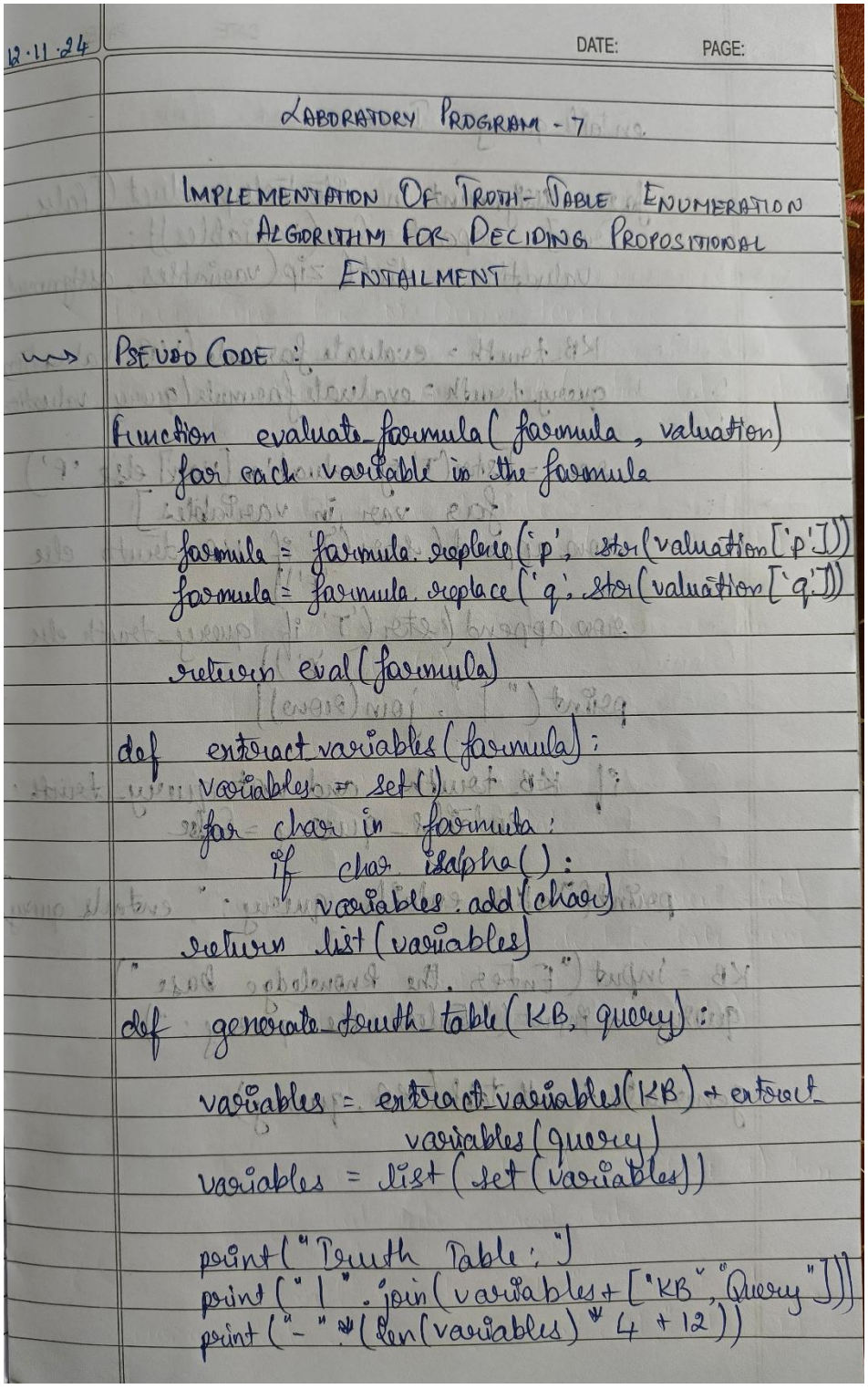
```
Iteration 16:
  Current state: [3, 2, 3, 1], Cost: 3, Temperature (T): 984.016
  Next state: [2, 2, 3, 1], Next cost: 2
  ΔE = -1
  Acceptance probability: 1.0010167601888467
  Acceptance condition met: True
Iteration 17:
  Current state: [2, 2, 3, 1], Cost: 2, Temperature (T): 983.017
  Next state: [2, 0, 3, 1], Next cost: 0
  ΔE = -2
  Acceptance probability: 1.0020366239173017
  Acceptance condition met: True
Solution found at iteration 18: [2, 0, 3, 1] with cost 0
Solution found for the initial state: [3, 1, 2, 0]
```



LABORATORY PROGRAM – 7

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

PSEUDOCODE OR ALGORITHM



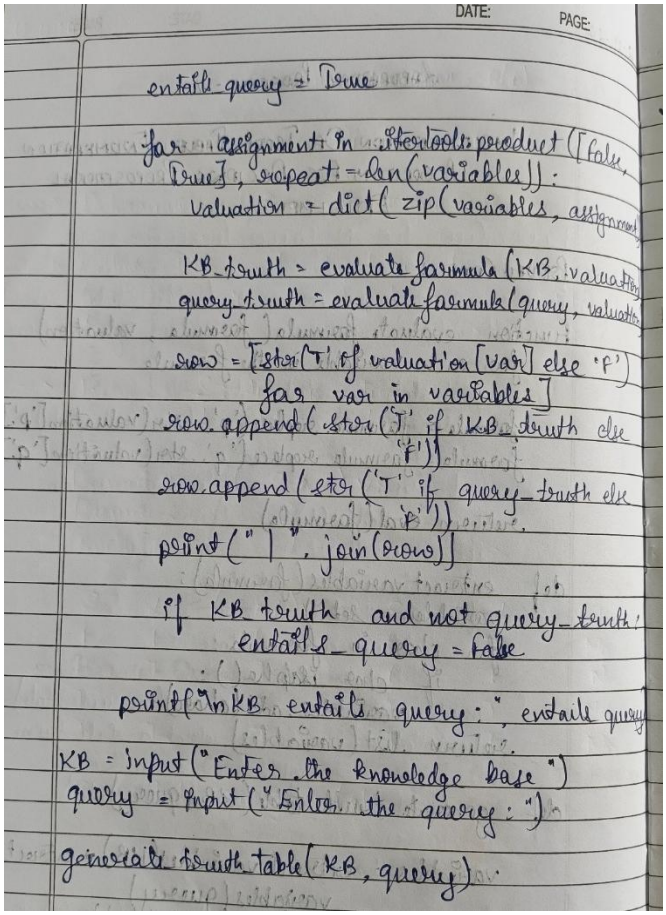
12.11.24 DATE: PAGE:

LABORATORY PROGRAM - 7

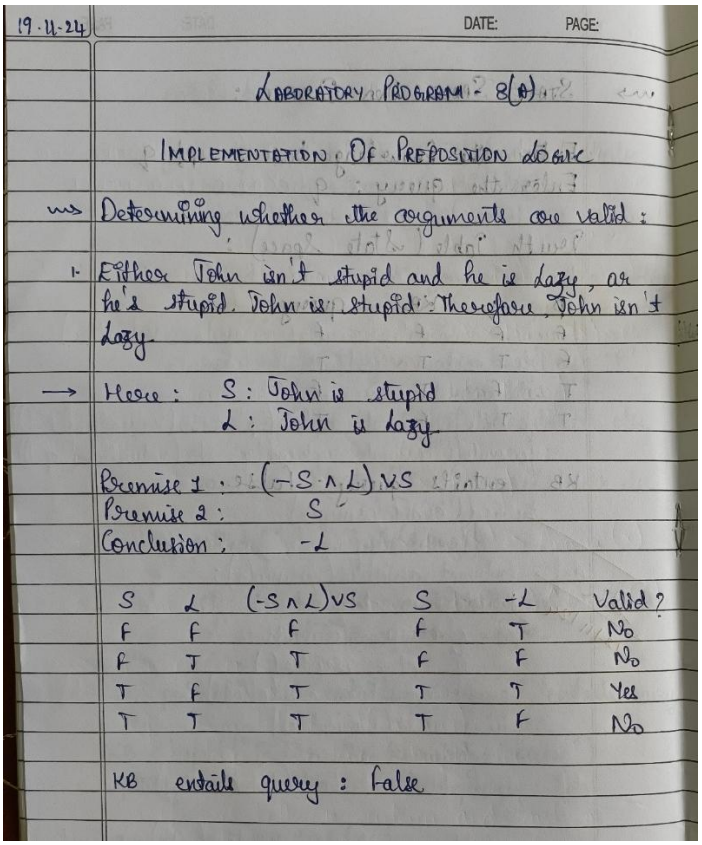
IMPLEMENTATION OF TRUTH-TABLE ENUMERATION ALGORITHM FOR DECIDING PROPOSITIONAL ENTAILMENT

PSEUDO CODE :

```
Function evaluate_formula(formula, valuation)
    for each variable in the formula
        formula = formula.replace('p', str(valuation['p']))
        formula = formula.replace('q', str(valuation['q']))
    return eval(formula)

def extract_variables(formula):
    variables = set()
    for char in formula:
        if char.isalpha():
            variables.add(char)
    return list(variables)
def generate_truth_table(KB, query):
    variables = extract_variables(KB) + extract_
                variables(query)
    variables = list(set(variables))

    print("Truth Table:")
    print(" | ".join(variables + ["KB", "Query"]))
    print("-" * (len(variables) * 4 + 12))
```



```
entails_query = True

for assignment in itertools.product([False, True], repeat = len(variables)):
    valuation = dict(zip(variables, assignment))

    KB_truth = evaluate_formula(KB, valuation)
    query_truth = evaluate_formula(query, valuation)
    row = [str('T' if valuation[var] else 'F')
           for var in variables]
    row.append(str('T' if KB_truth else 'F'))
    row.append(str('T' if query_truth else 'F'))
    print("|", join(row))

    if KB_truth and not query_truth:
        entails_query = False

print("In KB entails query:", entails_query)

KB = input("Enter the knowledge base")
query = input("Enter the query:")

generate_truth_table(KB, query)
```



19-11-24

LABORATORY PROGRAM - 8(a)

IMPLEMENTATION OF PREPOSITION LOGIC

⇝ Determining whether the arguments are valid:

1. Either John isn't stupid and he is lazy, or he's stupid. John is stupid. Therefore, John isn't lazy.

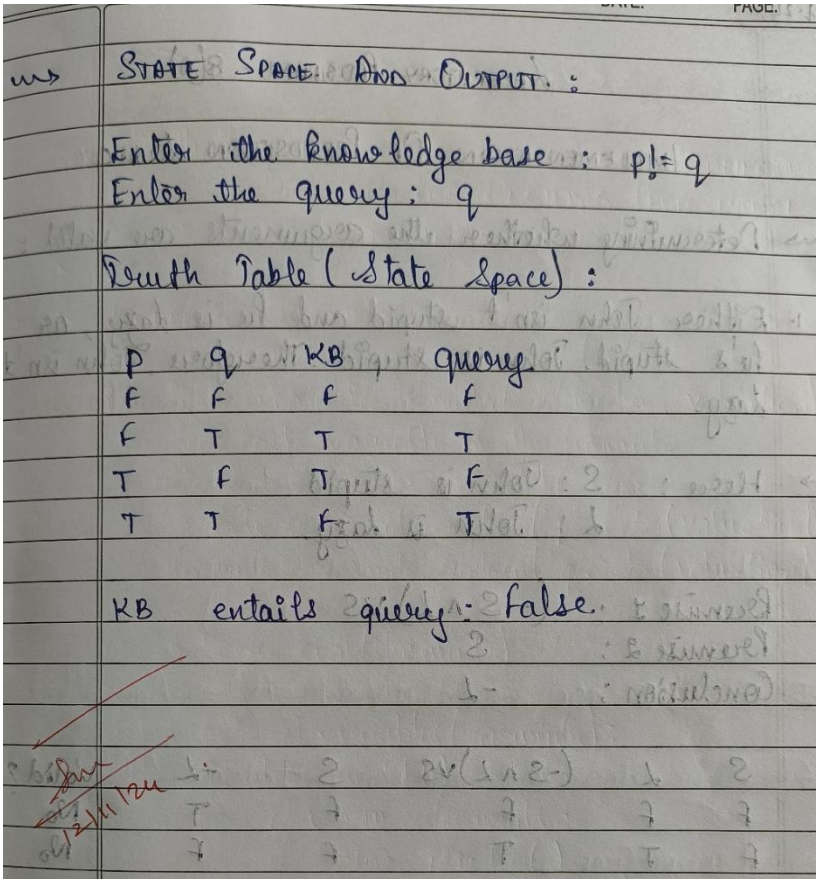
→ Here: S: John is stupid
L: John is lazy

Premise 1: $(\neg S \wedge L) \vee S$
Premise 2: S
Conclusion: $\neg L$

S	L	$(\neg S \wedge L) \vee S$	S	$\neg L$	Valid?
F	F	F	F	T	No
F	T	T	F	F	No
T	F	T	T	T	Yes
T	T	T	T	F	No

KB entails query: False.

STATE SPACE TREE



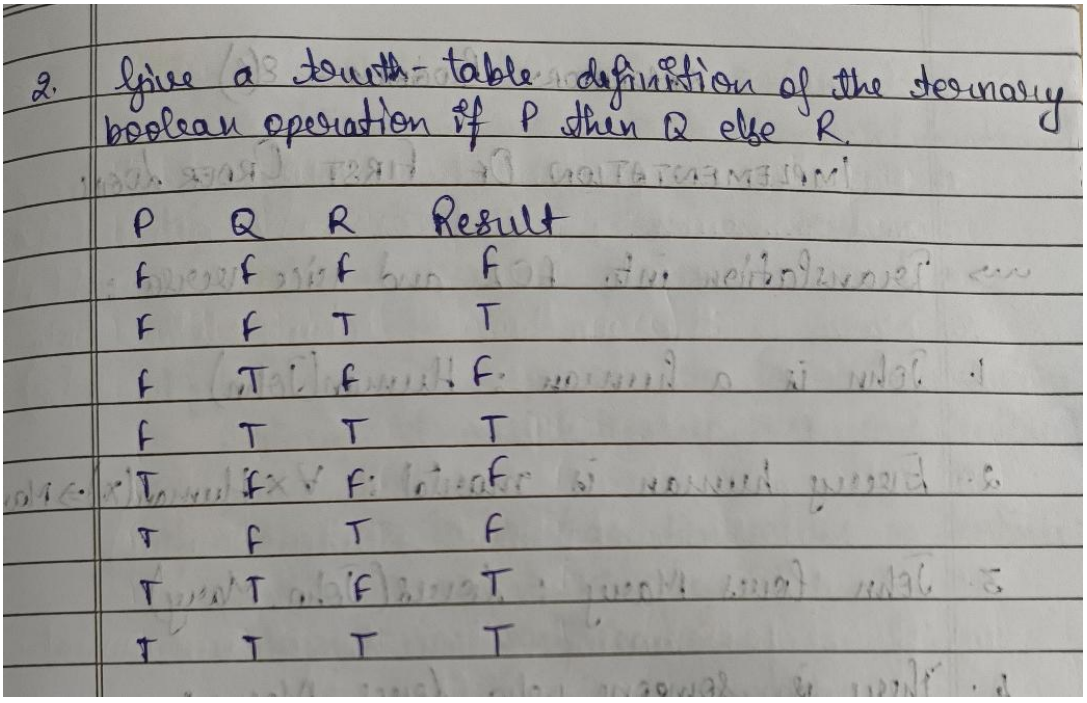
STATE SPACE AND OUTPUT :

Enter the knowledge base : p|= q
Enter the query : q

Truth Table (State Space) :

p	q	KB	query
F	F	F	F
F	T	T	T
T	F	T	F
T	T	F	T

KB entails query : false.



2. Give a truth-table definition of the ternary boolean operation if P then Q else R.

P	Q	R	Result
F	F	F	F
F	F	T	T
F	T	F	F
F	T	T	T
T	F	F	F
T	F	T	F
T	T	F	T
T	T	T	T

CODE

```
from itertools import product

# Evaluate a logical formula using the assignment of truth values
def evaluate_formula(formula, assignment):
    return eval(formula, {}, assignment)

# Generate all possible truth assignments for the given variables
def generate_all_assignments(variables):
    return [dict(zip(variables, values)) for values in product([True, False],
repeat=len(variables))]

# Create knowledge base and query from user input
def create_knowledge_base():
    print("Please enter the meanings of the following propositions:")
    p = input("Enter the meaning of proposition p : ")
    q = input("Enter the meaning of proposition q : ")
    r = input("Enter the meaning of proposition r : ")

    print(f"\nYou defined the following propositions:")
    print(f"p: {p}")
    print(f"q: {q}")
    print(f"r: {r}")

    print("\nNow, define the knowledge base (KB) and query (Q) using these propositions.")
    print("You can use 'p', 'q', 'r', 'not', 'or', 'and', and parentheses in your formulas.")

    KB = input("\nEnter the knowledge base (KB) formula : ")
    Q = input("\nEnter the query (Q) formula: ")

    return p, q, r, KB, Q

# Check if the knowledge base (KB) entails the query (Q) by evaluating the truth table
def truth_table_entailment(KB, Q, variables):
    all_assignments = generate_all_assignments(variables)

    print(f"\n{'p':<8}{'q':<8}{'r':<8}{'KB':<8}{'Q':<8}{'Entails'}")

    for assignment in all_assignments:
        KB_value = evaluate_formula(KB, assignment)
        Q_value = evaluate_formula(Q, assignment)
        entails = "Yes" if KB_value == True and Q_value == True else "No"

print(f"{assignment['p']:<8}{assignment['q']:<8}{assignment['r']:<8}{KB_value:<8}{Q_valu
e:<8}{entails}")

        if KB_value == True and Q_value == False:
            return False
    return True
```

```
# Main execution
if __name__ == "__main__":
    p, q, r, KB, Q = create_knowledge_base()
    variables = ['p', 'q', 'r']

    # Check if the KB entails the query Q
    result = truth_table_entailment(KB, Q, variables)

    if result:
        print("\nKB entails Q.")
    else:
        print("\nKB does not entail Q.")
```

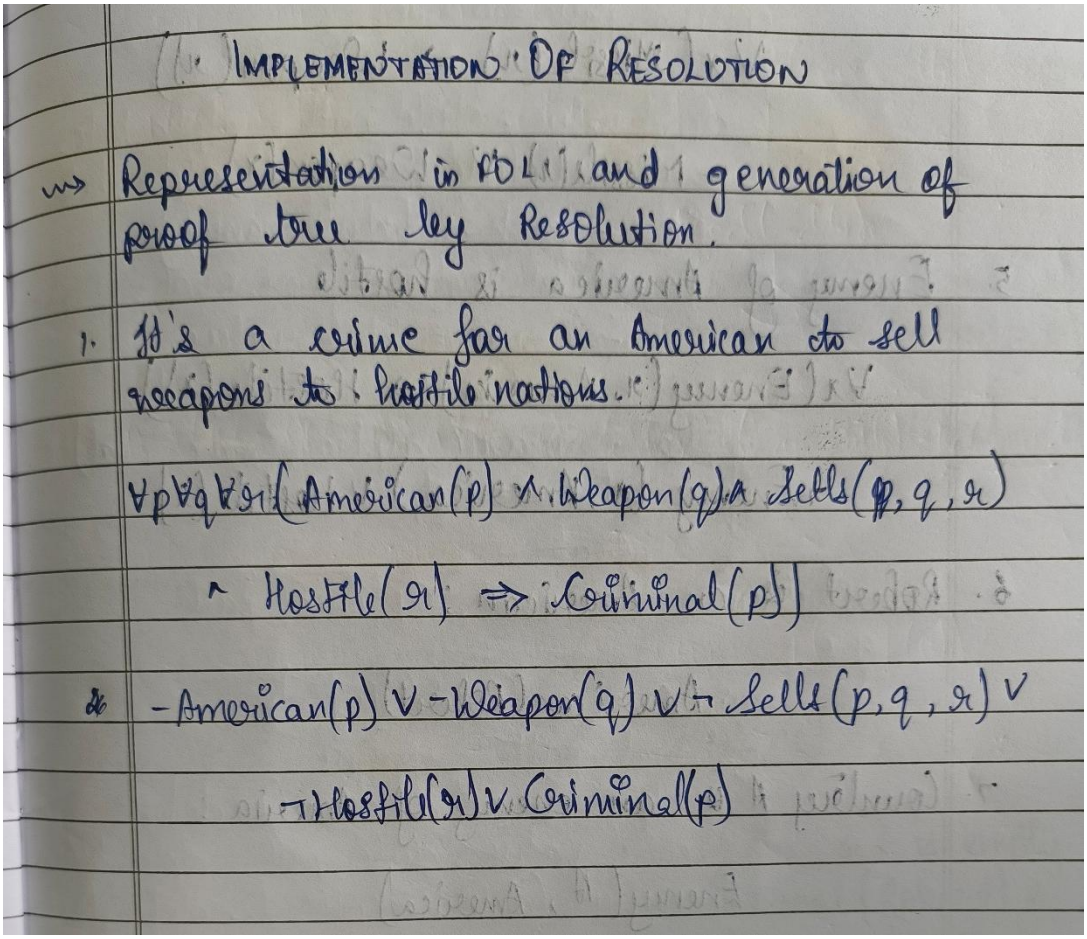
OUTPUT

```
Please enter the meanings of the following propositions:
Enter the meaning of proposition p : "It is raining"
Enter the meaning of proposition q : "The ground is wet"
Enter the meaning of proposition r : "The sun is shining"

You defined the following propositions:
p: "It is raining"
q: "The ground is wet"
r: "The sun is shining"

Now, define the knowledge base (KB) and query (Q) using these propositions.
You can use 'p', 'q', 'r', 'not', 'or', 'and', and parentheses in your formulas.

Enter the knowledge base (KB) formula : p and q

Enter the query (Q) formula: q

p       q       r       KB      Q       Entails
1       1       1       1       1       Yes
1       1       0       1       1       Yes
1       0       1       0       0       No
1       0       0       0       0       No
0       1       1       0       1       No
0       1       0       0       1       No
0       0       1       0       0       No
0       0       0       0       0       No

KB entails Q.
```

LABORATORY PROGRAM – 8

Create a knowledge base using prepositional logic and prove the given query using resolution.

PSEUDOCODE OR ALGORITHM



IMPLEMENTATION OF RESOLUTION

Representation in FOL and generation of proof tree by Resolution.

1. It's a crime for an American to sell weapons to hostile nations.

$\forall p \forall q \forall r (American(p) \wedge Weapon(q) \wedge Sells(p,q,r) \wedge Hostile(r) \Rightarrow Criminal(p))$

$\neg American(p) \vee \neg Weapon(q) \vee \neg Sells(p,q,r) \vee \neg Hostile(r) \vee Criminal(p)$

2. Country A has some missiles.

$$\exists x\ (Owns(A, x) \wedge Missile(x))$$

$$Owns(A, T_1) \wedge Missile(T_1)$$

3. All missiles were sold to country A by Robert.

$$\forall x\ (Missile(x) \wedge Owns(A, x) \Rightarrow Sells(Robert, x, A))$$

$$\neg Missile(x) \vee \neg Owns(A, x) \vee Sells(Robert, x, A)$$

4. Missiles are weapons.

$$\forall x\ (Missile(x) \Rightarrow Weapon(x))$$

$$\neg Missile(x) \vee Weapon(x)$$

5. Enemy of America is hostile.

$$\forall x\ (Enemy(x, America) \Rightarrow Hostile(x))$$

$$\neg Enemy(x, America) \vee Hostile(x)$$

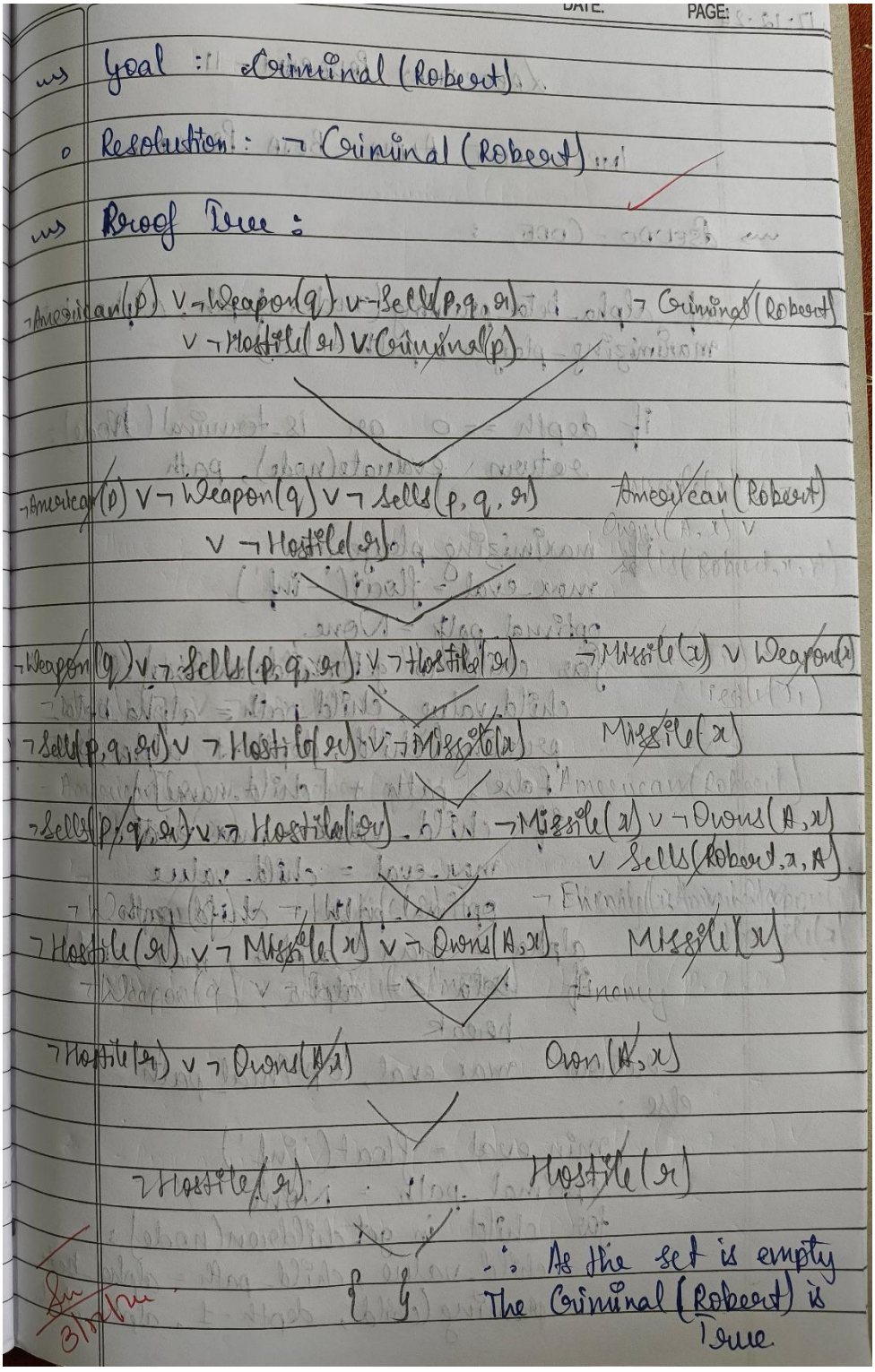
6. Robert is an American.

$$American(Robert)$$

7. Country A is an enemy of America.

$$Enemy(A, America)$$

STATE SPACE TREE



CODE

```
from typing import List, Set, Dict, Union

def unify(literal1: str, literal2: str) -> Union[Dict[str, str], None]:
"""
Unify two literals and return a substitution dictionary, or None if they cannot be unified.
"""
if literal1 == literal2:
return {}
if literal1.startswith("~") and literal2.startswith("~"):
return None
if literal1.startswith("~"):
neg, pos = literal1, literal2
else:
neg, pos = literal2, literal1

if neg[1:] == pos:
return {}
return None

def apply_substitution(clause: Set[str], substitution: Dict[str, str]) -> Set[str]:
"""
Apply a substitution to a clause.
"""
new_clause = set()
for literal in clause:
for var, value in substitution.items():
literal = literal.replace(var, value)
new_clause.add(literal)
return new_clause

def resolve(clause1: Set[str], clause2: Set[str]) -> Union[Set[str], None]:
"""
Resolves two clauses. Returns the resolvent clause or None if resolution is not possible.
"""
for lit1 in clause1:
for lit2 in clause2:
substitution = unify(lit1, lit2)
if substitution is not None:
# Create a new clause with unified literals removed
new_clause = (clause1 - {lit1}) | (clause2 - {lit2})
return apply_substitution(new_clause, substitution)
return None

def resolution(knowledge_base: List[Set[str]], query: Set[str]) -> bool:
"""
Implements the resolution algorithm.
"""
```

```
# Negate the query and add it to the knowledge base
negated_query = {f"~{literal}" if not literal.startswith("~") else literal[1:] for literal in query}
clauses = knowledge_base + [negated_query]

new_clauses = set()

while True:
pairs = [(clauses[i], clauses[j]) for i in range(len(clauses)) for j in range(i + 1, len(clauses))]
for clause1, clause2 in pairs:
resolvent = resolve(clause1, clause2)
if resolvent is not None:
if not resolvent:  # Empty clause found
return True
new_clauses.add(frozenset(resolvent))

# If no new clauses are generated, resolution has failed
if all(frozenset(c) in new_clauses for c in clauses):
return False

# Add new clauses to the set of clauses
clauses.extend(map(set, new_clauses))

if __name__ == "__main__":
print("Enter knowledge base (clauses) as sets of literals (comma-separated).")
print("Example: Likes(John, Food), Food(Apple). Enter 'done' when finished.")

knowledge_base = []
while True:
clause = input("Clause: ").strip()
if clause.lower() == "done":
break
knowledge_base.append(set(lit.strip() for lit in clause.split(',')))

print("Enter query as a set of literals (comma-separated).")
query = set(lit.strip() for lit in input("Query: ").strip().split(','))

result = resolution(knowledge_base, query)
print("Result:", "Entailed (True)" if result else "Not Entailed (False)")
```

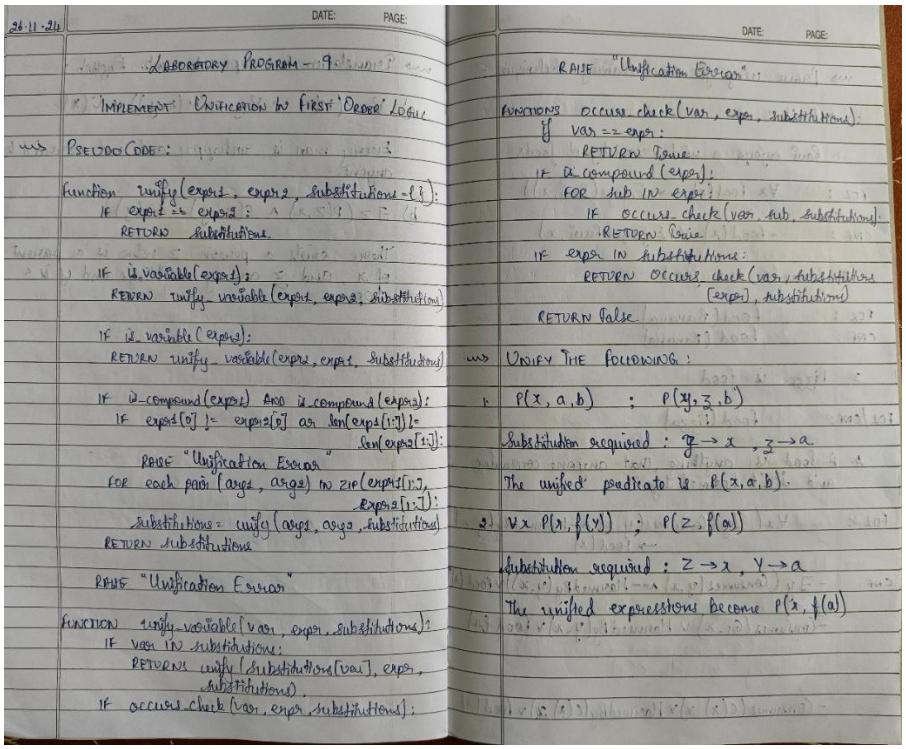
OUTPUT

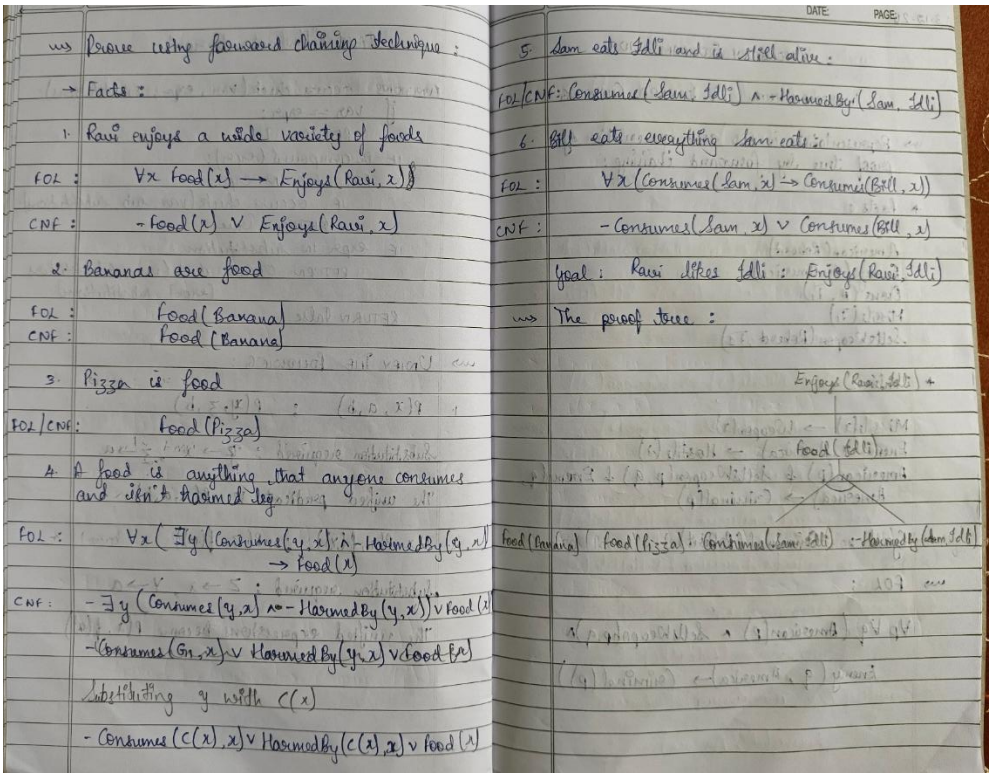
```
Enter knowledge base (clauses) as sets of literals (comma-separated).
Example: Likes(John, Food), Food(Apple). Enter 'done' when finished.
Clause: Likes(John, Food), Food(Apple)
Clause: ~Likes(John, Apple)
Clause: done
Enter query as a set of literals (comma-separated).
Query: Likes(John, Apple)
Result: Entailed (True)
```

LABORATORY PROGRAM – 9

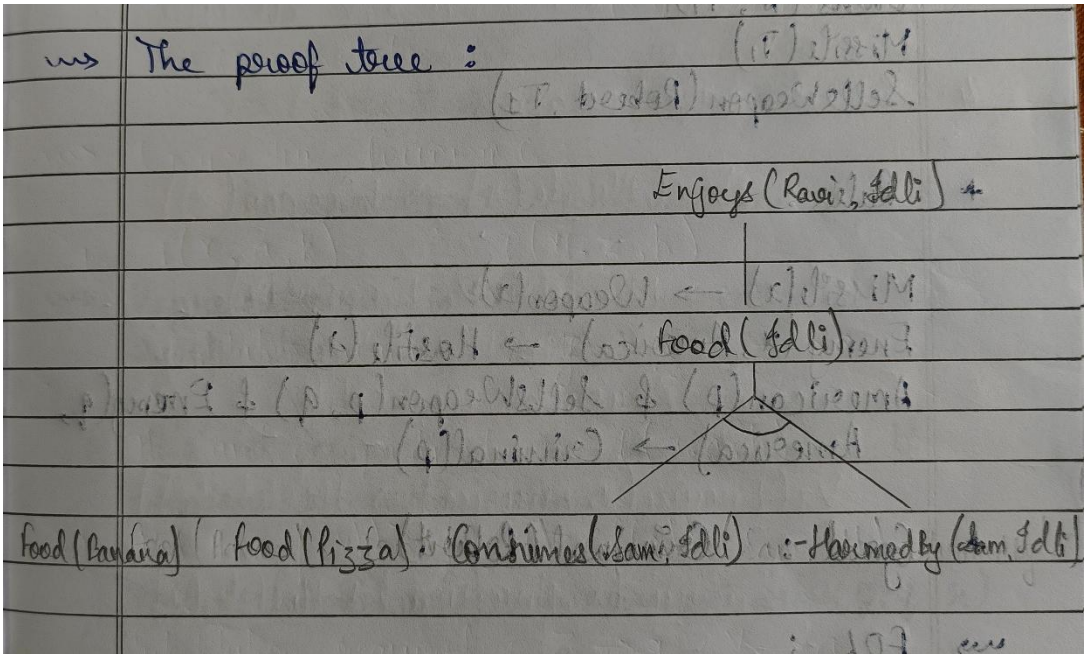
Implement unification in first order logic.

PSEUDOCODE OR ALGORITHM





STATE SPACE TREE



CODE

```
#Laboratory - 9
#Implement unification in first order logic.

class UnificationError(Exception):
    pass

def unify(expr1, expr2, substitutions=None):
    if substitutions is None:
        substitutions = {}

    # If both expressions are identical, return current substitutions
    if expr1 == expr2:
        return substitutions

    # If the first expression is a variable
    if is_variable(expr1):
        return unify_variable(expr1, expr2, substitutions)

    # If the second expression is a variable
    if is_variable(expr2):
        return unify_variable(expr2, expr1, substitutions)

    # If both expressions are compound expressions
    if is_compound(expr1) and is_compound(expr2):
        if expr1[0] != expr2[0] or len(expr1[1:]) != len(expr2[1:]):
            raise UnificationError("Expressions do not match.")
        return unify_lists(expr1[1:], expr2[1:], unify(expr1[0], expr2[0], substitutions))

    # If expressions are not compatible
    raise UnificationError(f"Cannot unify {expr1} and {expr2}.")

def unify_variable(var, expr, substitutions):
    if var in substitutions:
        return unify(substitutions[var], expr, substitutions)
    elif occurs_check(var, expr, substitutions):
        raise UnificationError(f"Occurs check failed: {var} in {expr}.")
    else:
        substitutions[var] = expr
        return substitutions

def unify_lists(list1, list2, substitutions):
    for expr1, expr2 in zip(list1, list2):
        substitutions = unify(expr1, expr2, substitutions)
    return substitutions

def is_variable(term):
    return isinstance(term, str) and term[0].islower()
```

```
def is_compound(term):
    return isinstance(term, (list, tuple)) and len(term) > 0

def occurs_check(var, expr, substitutions):
    if var == expr:
        return True
    elif is_compound(expr):
        return any(occurs_check(var, sub, substitutions) for sub in expr)
    elif expr in substitutions:
        return occurs_check(var, substitutions[expr], substitutions)
    return False

# Function to parse input into a usable expression format
def parse_expression(expr_str):
    # Try to evaluate the expression as a tuple or list
    try:
        expr = eval(expr_str)
        if isinstance(expr, (tuple, list)) and len(expr) > 0:
            return expr
        else:
            raise ValueError("Expression must be a non-empty tuple or list.")
    except Exception as e:
        raise ValueError(f"Invalid expression format: {e}")

# Example usage: allow user input for the expressions
try:
    expr1_str = input("Enter the first expression (e.g., ('f', 'x', ('g', 'y'))): ")
    expr2_str = input("Enter the second expression (e.g., ('f', 'a', ('g', 'b'))): ")

    # Parse the user input expressions
    expr1 = parse_expression(expr1_str)
    expr2 = parse_expression(expr2_str)

    # Perform unification
    result = unify(expr1, expr2)
    print("Unified substitutions:", result)
except UnificationError as e:
    print("Unification failed:", e)
except ValueError as e:
    print("Input error:", e)
```

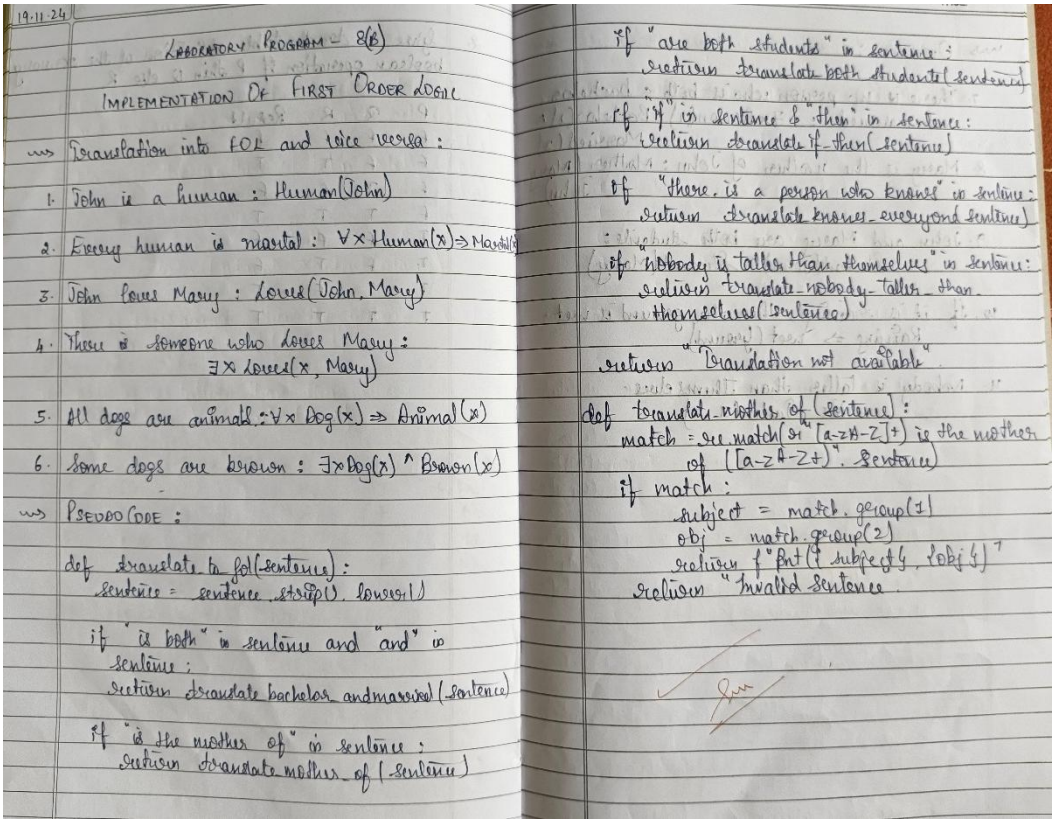
OUTPUT

```
Enter the first expression (e.g., ('f', 'x', ('g', 'y'))): ('P', 'x', 'a', 'b')
Enter the second expression (e.g., ('f', 'a', ('g', 'b'))): ('P', 'y', 'z', 'b')
Unified substitutions: {'x': 'y', 'a': 'z'}
```

LABORATORY PROGRAM – 10

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

PSEUDOCODE OR ALGORITHM


19.11.24

LABORATORY PROGRAM - 8(b)

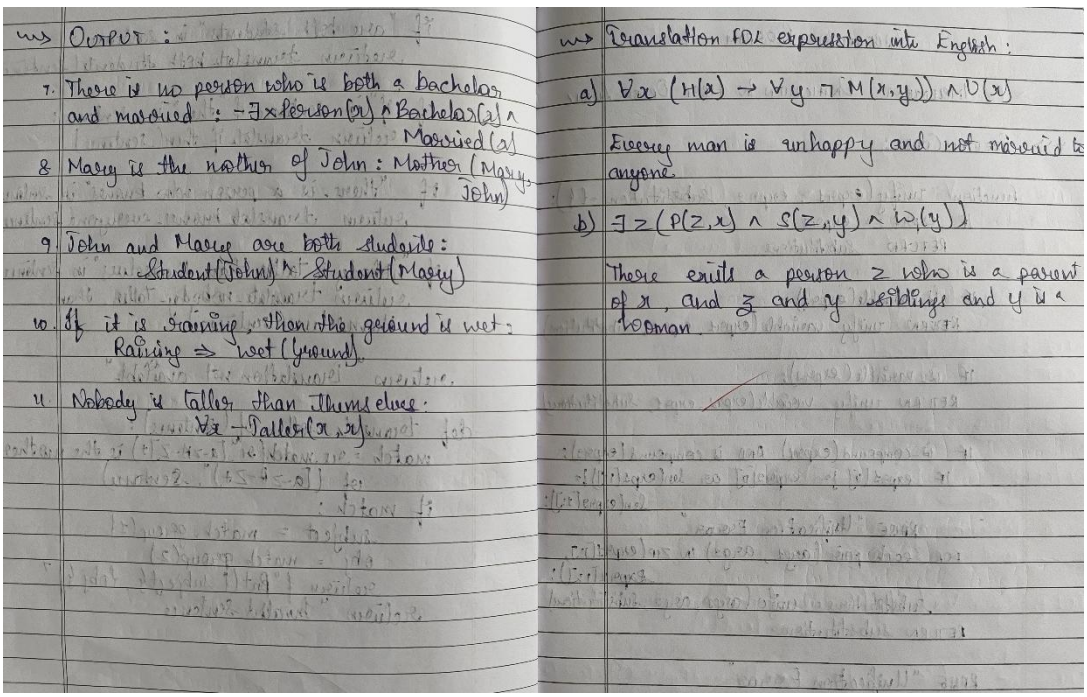
IMPLEMENTATION OF FIRST ORDER LOGIC

=> Translation into FOL and vice versa :

1. John is a human : Human(John)
2. Every human is mortal : ∀x Human(x) ⇒ Mortal(x)
3. John loves Mary : Loves(John, Mary)
4. There is someone who loves Mary : ∃x Loves(x, Mary)
5. All dogs are animals : ∀x Dog(x) ⇒ Animal(x)
6. Some dogs are brown : ∃x Dog(x) ∧ Brown(x)

=> PSEUDO CODE :

```
def translate_to_fol(sentence):
    sentence = sentence.strip().lower()

    if "is both" in sentence and "and" in
    sentence:
    return translate_bachelor_andmarried(sentence)

    if "is the mother of" in sentence:
    return translate_mother_of(sentence)

    if "are both students" in sentence:
    return translate_both_students(sentence)

    if "if" in sentence & "then" in sentence:
    return translate_if_then(sentence)

    if "there is a person who knows" in sentence:
    return translate_knows_everyone(sentence)

    if "nobody is taller than themselves" in sentence:
    return translate_nobody_taller_than_
    themselves(sentence)

    return "Translation not available"

def translate_mother_of(sentence):
    match = re.match(r"([a-zA-Z]+) is the mother
    of ([a-zA-Z]+)", sentence)
    if match:
        subject = match.group(1)
        obj = match.group(2)
        return f"Mother({subject}, {obj})"
    return "Invalid Sentence"
```


=> OUTPUT :

7. There is no person who is both a bachelor and married : ¬∃x Person(x) ∧ Bachelor(x) ∧ Married(x)
8. Mary is the mother of John : Mother(Mary, John)
9. John and Mary are both students : Student(John) ∧ Student(Mary)
10. If it is raining, then the ground is wet : Raining ⇒ Wet(Ground)
11. Nobody is taller than themselves : ∀x ¬Taller(x, x)

=> Translation FOL expression into English :

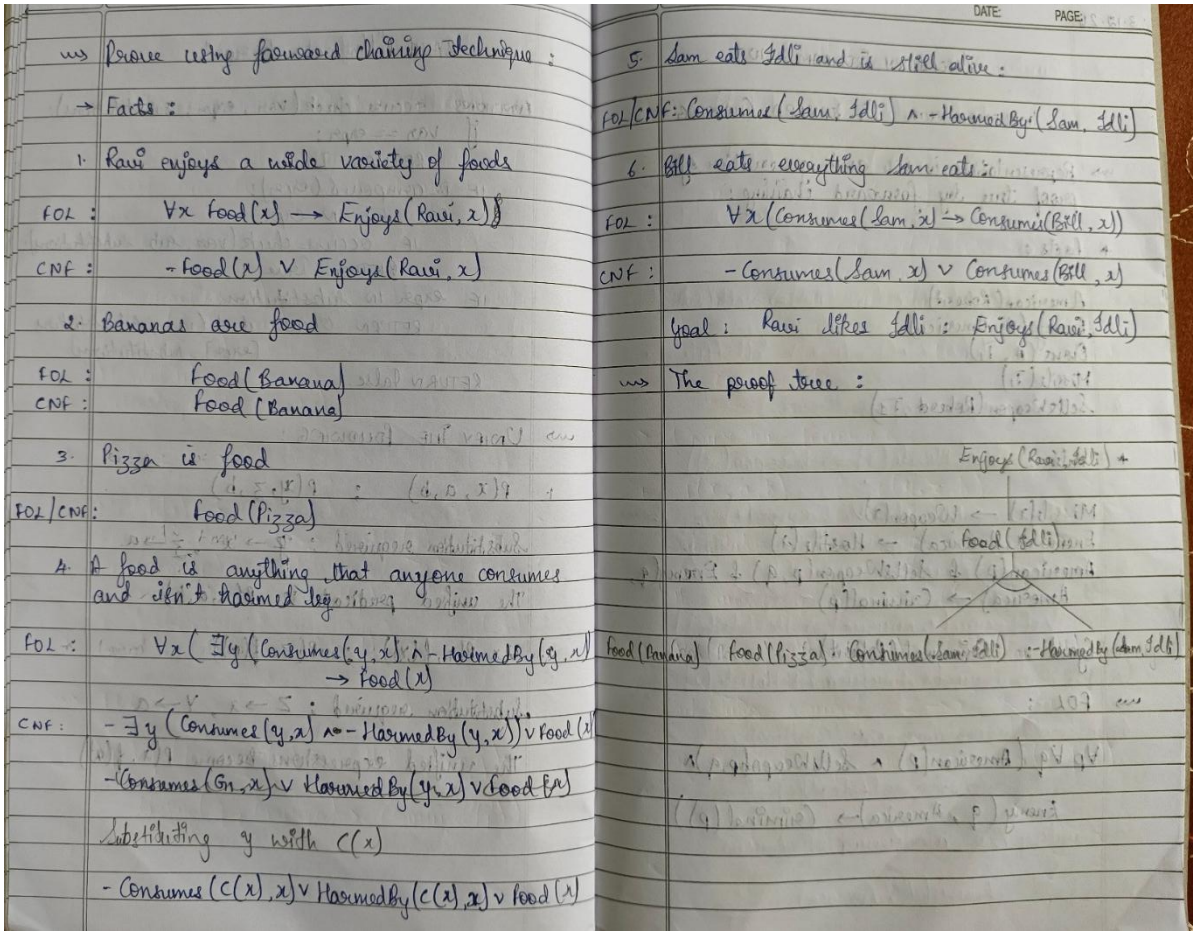
a) ∀x (H(x) → ∀y ¬M(x,y)) ∧ U(x)

Every man is unhappy and not married to anyone.

b) ∃z (P(z,x) ∧ S(z,y) ∧ W(y))

There exists a person z who is a parent of x, and z and y siblings and y is a woman.

STATE SPACE TREE



Prove using forward chaining technique :

→ Facts :

1. Ravi enjoys a wide variety of foods

FOL : $\forall x \; Food(x) \rightarrow Enjoys(Ravi, x)$

CNF : $\neg Food(x) \vee Enjoys(Ravi, x)$

2. Bananas are food

FOL : $Food(Banana)$

CNF : $Food(Banana)$

3. Pizza is food

FOL/CNF : $Food(Pizza)$

4. A food is anything that anyone consumes and isn't harmed by

FOL : $\forall x \left(\exists y \left(Consumes(y, x) \wedge \neg HarmedBy(y, x) \right) \rightarrow Food(x) \right)$

CNF : $\neg \exists y \left(Consumes(y, x) \wedge \neg HarmedBy(y, x) \right) \vee Food(x)$

$\neg Consumes(Gn, x) \vee HarmedBy(y, x) \vee Food(x)$

Substituting y with $C(x)$

$\neg Consumes(C(x), x) \vee HarmedBy(C(x), x) \vee Food(x)$

5. Sam eats Idli and is still alive :

FOL/CNF : $Consumes(Sam, Idli) \wedge \neg HarmedBy(Sam, Idli)$

6. Bill eats everything Sam eats :

FOL : $\forall x \left(Consumes(Sam, x) \rightarrow Consumes(Bill, x) \right)$

CNF : $\neg Consumes(Sam, x) \vee Consumes(Bill, x)$

Goal : Ravi likes Idli : $Enjoys(Ravi, Idli)$

The proof tree :

CODE

```
import re


# Helper functions to apply the transformations step by step.

# 1. Eliminate biconditionals and implications
def eliminate_biconditionals_implications(expr):
    # Eliminate biconditionals (⇔)
    expr = re.sub(r'([A-Za-z0-9()]+) ⇔ ([A-Za-z0-9()]+)', r'(\1 ⇒ \2) ∧ (\2 ⇒ \1)', expr)

    # Eliminate implications (⇒)
    expr = re.sub(r'([A-Za-z0-9()]+) ⇒ ([A-Za-z0-9()]+)', r'¬\1 ∨ \2', expr)

    return expr

# 2. Move negations inward
def move_negations_inward(expr):
    expr = re.sub(r'¬\((∀x [A-Za-z0-9()]+)\)', r'∃x ¬\1', expr)  # Move negation inside ∀
    expr = re.sub(r'¬\((∃x [A-Za-z0-9()]+)\)', r'∀x ¬\1', expr)  # Move negation inside ∃
    expr = re.sub(r'¬\(([^()]+) ∨ ([^()]+)\)', r'¬\1 ∧ ¬\2', expr)  # De Morgan's law: ¬(A ∨ B)
    expr = re.sub(r'¬\(([^()]+) ∧ ([^()]+)\)', r'¬\1 ∨ ¬\2', expr)  # De Morgan's law: ¬(A ∧ B)
    expr = re.sub(r'¬¬([A-Za-z0-9()]+)', r'\1', expr)  # Double negation elimination
    return expr

# 3. Skolemization: Replace existential quantifiers with Skolem constants/functions
def skolemize(expr):
    expr = re.sub(r'∃([A-Za-z0-9]+)', r'G1', expr)  # Replace ∃x with Skolem constant (G1)
    expr = re.sub(r'∃([A-Za-z0-9]+)\(([A-Za-z0-9, ()]+)\)', r'F1(\2)', expr)  # Existential
quantifier -> Skolem function
    return expr

# 4. Drop universal quantifiers
def drop_universal_quantifiers(expr):
    expr = re.sub(r'∀([A-Za-z0-9]+)', '', expr)  # Drop ∀ quantifiers
    return expr

# 5. Distribute AND over OR to get CNF
def distribute_and_over_or(expr):
    expr = re.sub(r'(\([A-Za-z0-9()]+ ∧ [A-Za-z0-9()]+\)) ∨ ([A-Za-z0-9()]+)', r'(\1 ∨ \2)', expr)
    expr = re.sub(r'([A-Za-z0-9()]+ ∧ [A-Za-z0-9()]+) ∨ ([A-Za-z0-9()]+)', r'(\1 ∨ \2)', expr)
    return expr

# Convert to CNF using the above steps
def convert_to_cnf(expr):
    # Step 1: Eliminate biconditionals and implications
    expr = eliminate_biconditionals_implications(expr)

    # Step 2: Move negations inward
```

```
    expr = move_negations_inward(expr)

    # Step 3: Skolemize the expression
    expr = skolemize(expr)

    # Step 4: Drop universal quantifiers
    expr = drop_universal_quantifiers(expr)

    # Step 5: Distribute AND over OR to get CNF
    expr = distribute_and_over_or(expr)

    return expr

# Example FOL expressions (from the problem statement)
fol_expressions = [
    "Mary is the mother of John: Mother(Mary, John)",
    "John and Mary are both students: Student(John) ∧ Student(Mary)",
    "If it is raining, then the ground is wet: Raining ⇒ Wet(Ground)",
    "There is a person who knows every other person: ∃x ∀y (x ≠ y ⇒ Knows(x, y))",
    "Nobody is taller than themselves: ∀x ¬Taller(x, x)",
    "All students in the class passed the exam: ∀x (Student(x) ⇒ Passed(x, Exam))",
    "Mary has a pet dog: ∃x (Pet(x) ∧ Dog(x) ∧ Has(Mary, x))",
    "If Alice is a teacher, then Alice teaches mathematics: Teacher(Alice) ⇒ Teaches(Alice,
Mathematics)",
    "Everyone loves someone: ∀x ∃y Loves(x, y)",
    "No one is both a teacher and a student: ∀x ¬(Teacher(x) ∧ Student(x))",
    "Every man respects his parent: ∀x (Man(x) ⇒ Respects(x, Parent(x)))",
    "Not all students like both Mathematics and Science: ¬∀x (Student(x) ⇒ (Likes(x,
Mathematics) ∧ Likes(x, Science)))"
]

# Convert and print CNF for each expression
for expr in fol_expressions:
    print(f"Original FOL Expression: {expr}")
    expression = expr.split(":")[1].strip()  # Remove the description part and keep the formula
    cnf = convert_to_cnf(expression)
    print(f"CNF: {cnf}\n")
```

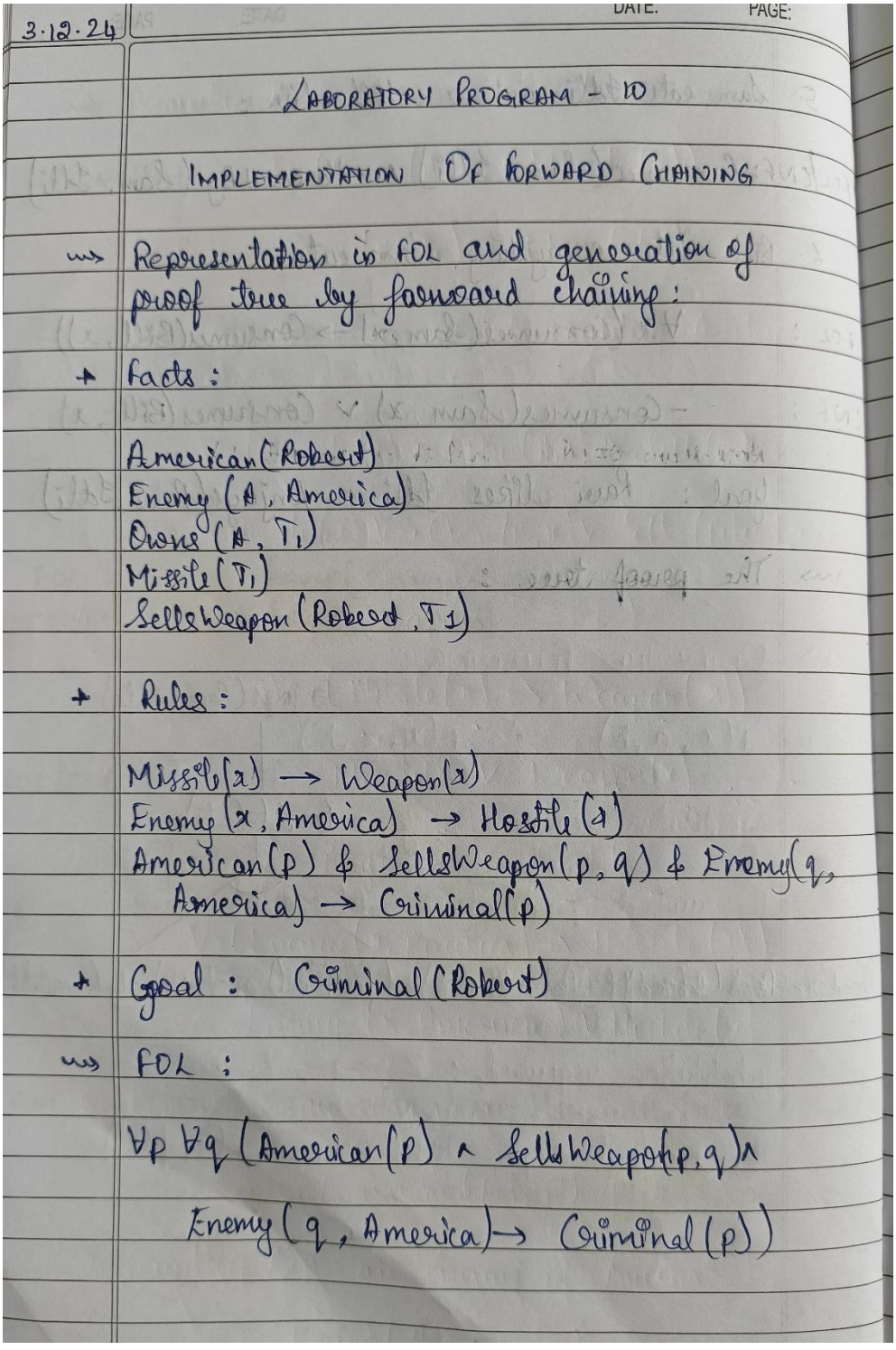
OUTPUT

```
Original FOL Expression: Mary is the mother of John: Mother(Mary, John)
CNF: Mother(Mary, John)

Original FOL Expression: John and Mary are both students: Student(John) ∧ Student(Mary)
CNF: Student(John) ∧ Student(Mary)

Original FOL Expression: If it is raining, then the ground is wet: Raining ⇒ Wet(Ground)
CNF: ¬Raining ∨ Wet(Ground)

Original FOL Expression: There is a person who knows every other person: ∃x ∀y (x ≠ y ⇒ Knows(x, y))
CNF: G1  (x ≠ ¬y ∨ Knows(x, y))

Original FOL Expression: Nobody is taller than themselves: ∀x ¬Taller(x, x)
CNF:  ¬Taller(x, x)

Original FOL Expression: All students in the class passed the exam: ∀x (Student(x) ⇒ Passed(x, Exam))
CNF:  ¬(Student(x) ∨ Passed(x, Exam))

Original FOL Expression: Mary has a pet dog: ∃x (Pet(x) ∧ Dog(x) ∧ Has(Mary, x))
CNF: G1 (Pet(x) ∧ Dog(x) ∧ Has(Mary, x))

Original FOL Expression: If Alice is a teacher, then Alice teaches mathematics: Teacher(Alice) ⇒ Teaches(Alice, Mathematics)
CNF: ¬Teacher(Alice) ∨ Teaches(Alice, Mathematics)

Original FOL Expression: Everyone loves someone: ∀x ∃y Loves(x, y)
CNF:  G1 Loves(x, y)

Original FOL Expression: No one is both a teacher and a student: ∀x ¬(Teacher(x) ∧ Student(x))
CNF:  ¬(Teacher(x) ∧ Student(x))

Original FOL Expression: Every man respects his parent: ∀x (Man(x) ⇒ Respects(x, Parent(x)))
CNF:  ¬(Man(x) ∨ Respects(x, Parent(x)))

Original FOL Expression: Not all students like both Mathematics and Science: ¬∀x (Student(x) ⇒ (Likes(x, Mathematics) ∧ Likes(x, Science)))
CNF: ¬ ¬(Student(x) ∨ (Likes(x, Mathematics) ∧ Likes(x, Science)))
```

LABORATORY PROGRAM – 11

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

PSEUDOCODE OR ALGORITHM



3·12·24

DATE: PAGE:

LABORATORY PROGRAM - 10

IMPLEMENTATION OF FORWARD CHAINING

⇝ Representation in FOL and generation of proof tree by forward chaining:

+ Facts:

American(Robert)
Enemy(A, America)
Owns(A, T1)
Missile(T1)
SellsWeapon(Robert, T1)

+ Rules:

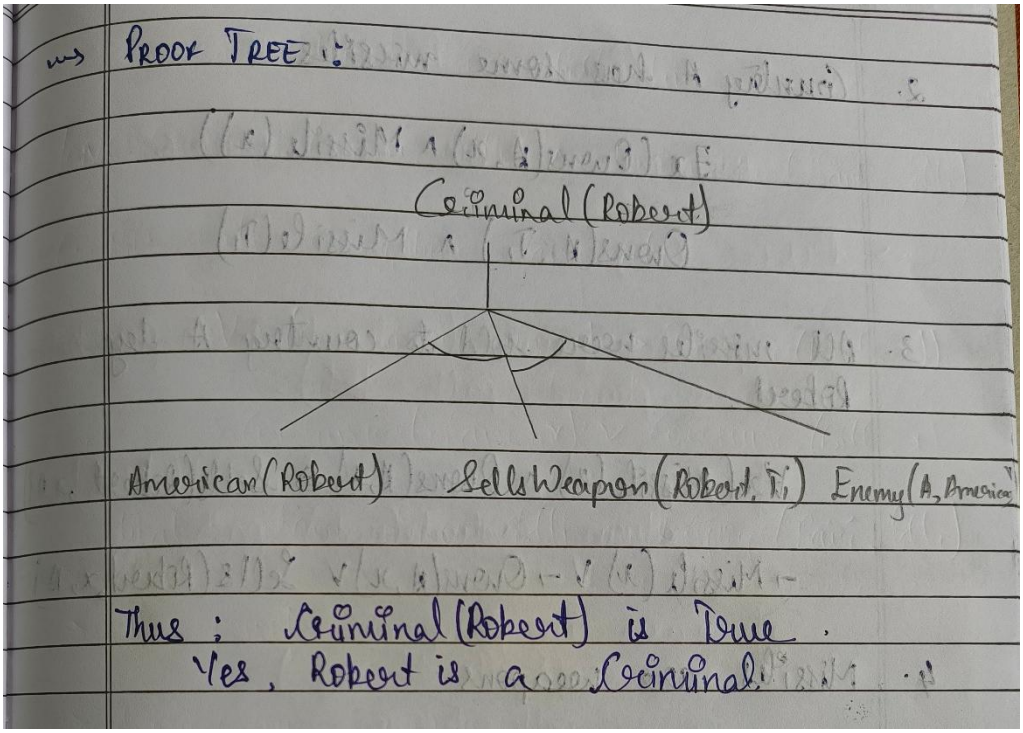
Missile(x) → Weapon(x)
Enemy(x, America) → Hostile(x)
American(p) & SellsWeapon(p, q) & Enemy(q, America) → Criminal(p)

+ Goal: Criminal(Robert)

⇝ FOL:

∀p ∀q (American(p) ∧ SellsWeapon(p, q) ∧ Enemy(q, America) → Criminal(p))

STATE SPACE TREE

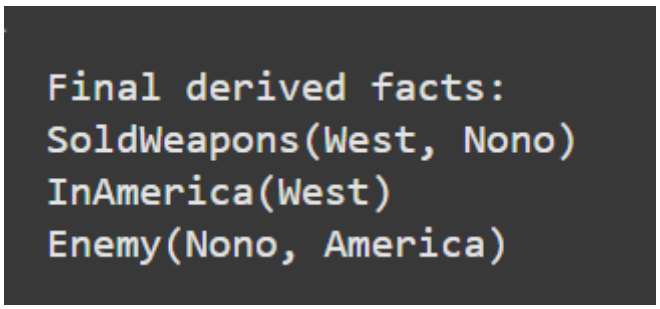


CODE

```
# Define initial facts and rules
facts = {"InAmerica(West)", "SoldWeapons(West, Nono)", "Enemy(Nono, America)"}
rules = [
    {
        "conditions": ["InAmerica(x)", "SoldWeapons(x, y)", "Enemy(y, America)"],
        "conclusion": "Criminal(x)",
    },
    {
        "conditions": ["Enemy(y, America)"],
        "conclusion": "Dangerous(y)",
    },
]

# Forward chaining function
def forward_chaining(facts, rules):
    derived_facts = set(facts)  # Initialize derived facts
    while True:
        new_fact_found = False

        for rule in rules:
            # Substitute variables and check if conditions are met
            for fact in derived_facts:
                if "x" in rule["conditions"][0]:
                    # Substitute variables (x, y) with specific instances
                    for condition in rule["conditions"]:
                        if "x" in condition or "y" in condition:
                            x = "West"  # Hardcoded substitution for simplicity
                            y = "Nono"
                            conditions = [
                                cond.replace("x", x).replace("y", y)
                                for cond in rule["conditions"]
                            ]
                            conclusion = (
                                rule["conclusion"].replace("x", x).replace("y", y)
                            )

                            # Check if all conditions are satisfied
                            if all(cond in derived_facts for cond in conditions) and conclusion not in
derived_facts:
                                derived_facts.add(conclusion)
                                print(f"New fact derived: {conclusion}")
                                new_fact_found = True

        # Exit loop if no new fact is found
        if not new_fact_found:
            break

    return derived_facts
```

```
# Run forward chaining
final_facts = forward_chaining(facts, rules)
print("\nFinal derived facts:")
for fact in final_facts:
    print(fact)
```

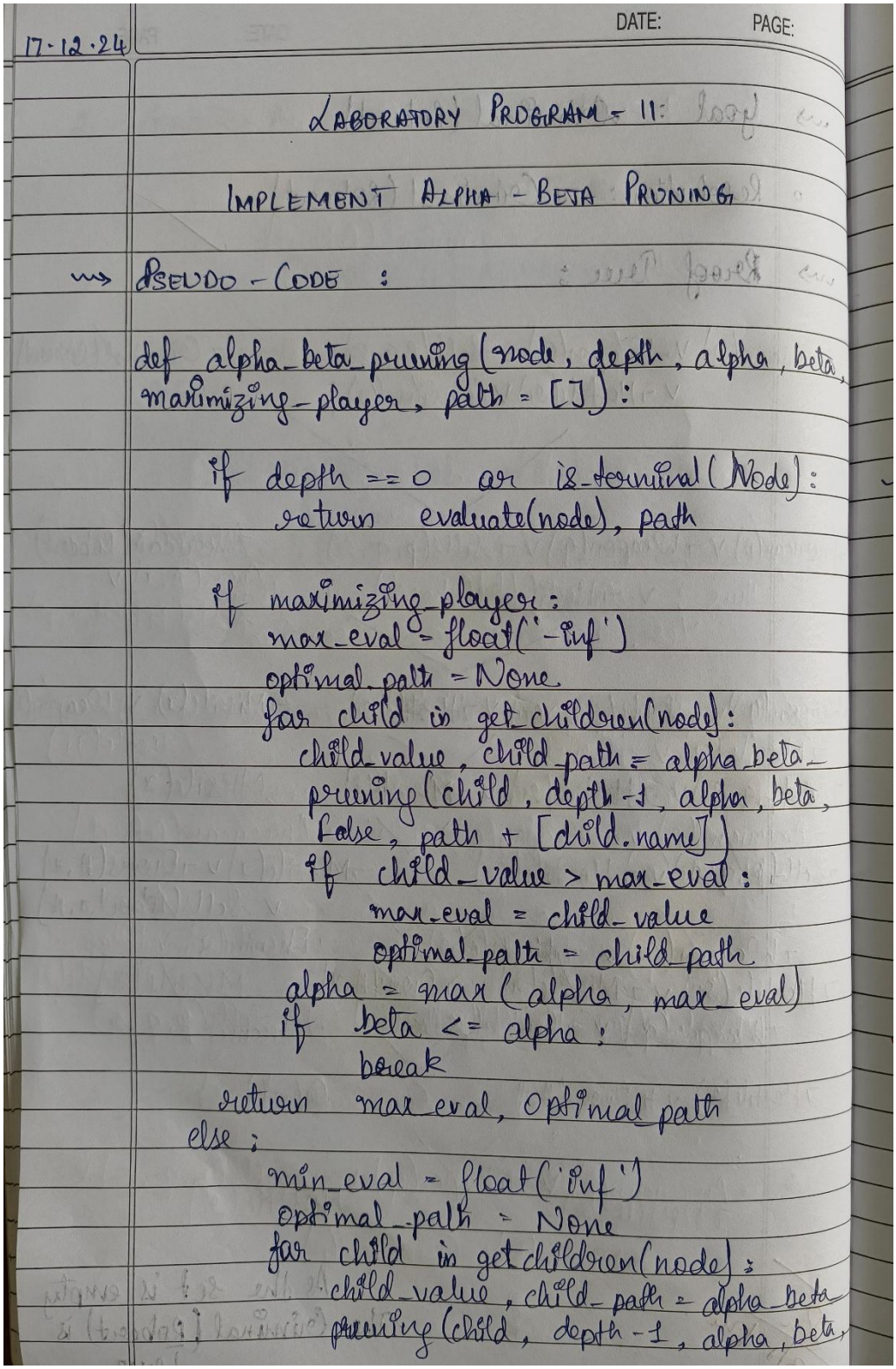
OUTPUT



LABORATORY PROGRAM – 12

Implement Alpha-Beta Pruning

PSEUDOCODE OR ALGORITHM



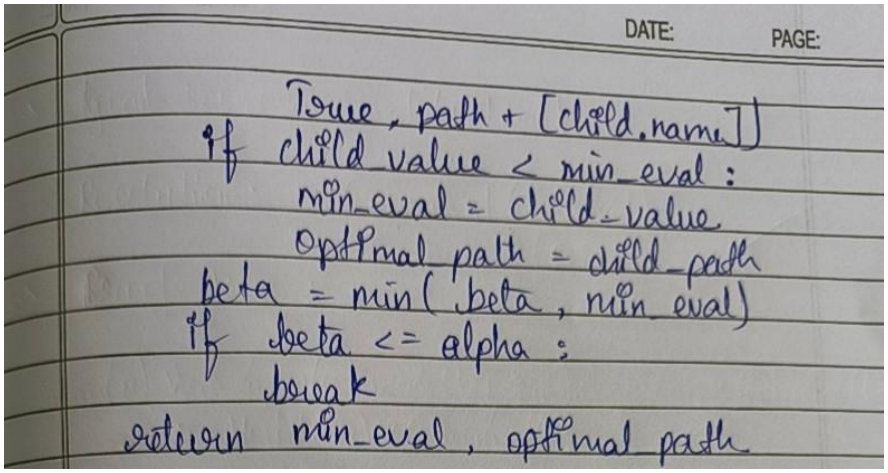
17-12-24 DATE: PAGE:

LABORATORY PROGRAM - 11:

IMPLEMENT ALPHA - BETA PRUNING

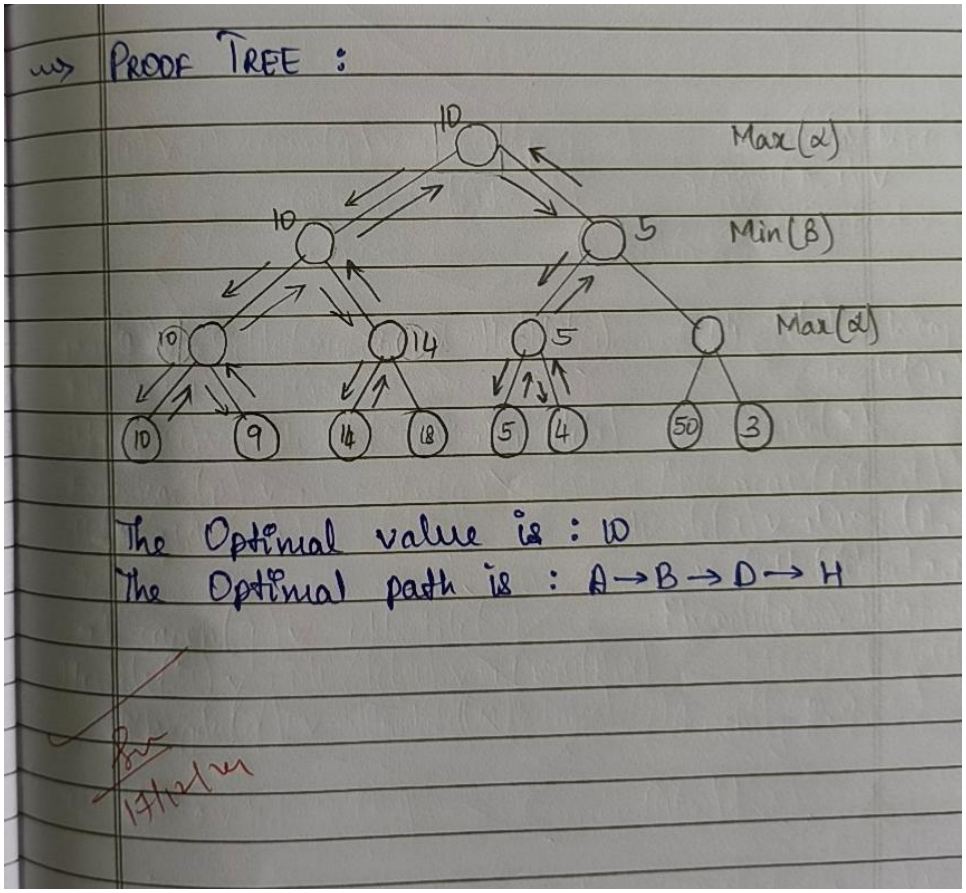
⇝ PSEUDO - CODE :

```
def alpha_beta_pruning(node, depth, alpha, beta,
maximizing_player, path = []):

    if depth == 0 or is_terminal(Node):
        return evaluate(node), path

    if maximizing_player:
        max_eval = float('-inf')
        optimal_path = None
        for child in get_children(node):
            child_value, child_path = alpha_beta_
            pruning(child, depth-1, alpha, beta,
            False, path + [child.name])
            if child_value > max_eval:
                max_eval = child_value
                optimal_path = child_path
            alpha = max(alpha, max_eval)
            if beta <= alpha:
                break
        return max_eval, optimal_path
    else:
        min_eval = float('inf')
        optimal_path = None
        for child in get_children(node):
            child_value, child_path = alpha_beta_
            pruning(child, depth-1, alpha, beta,
```



```
                                    DATE:        PAGE:
            True, path + [child.name])
        if child_value < min_eval:
            min_eval = child_value
            optimal_path = child_path
        beta = min(beta, min_eval)
        if beta <= alpha:
            break
    return min_eval, optimal_path
```

STATE SPACE TREE



CODE

```
class Node:
    def __init__(self, name, value=None, children=None):
        self.name = name
        self.value = value  # The value of the node (used for terminal nodes)
        self.children = children or []  # List of child nodes

def alpha_beta_search(state):
    def max_value(node, alpha, beta, path):
        print(f"MAX: Visiting node {node.name}, alpha={alpha}, beta={beta}")
        if terminal_test(node):
            print(f"MAX: Terminal node {node.name} has utility {utility(node)}")
            return utility(node), path
        v = float('-inf')
        best_path = []
        for child in node.children:
            value, new_path = min_value(child, alpha, beta, path + [child.name])
            print(f"MAX: From node {node.name}, child {child.name} → value={value}")
            if value > v:
                v = value
                best_path = new_path
            if v >= beta:
                print(f"MAX: Pruning at node {node.name} with value={v} ≥ beta={beta}")
                return v, best_path
            alpha = max(alpha, v)
        print(f"MAX: Returning value={v} for node {node.name}")
        return v, best_path

    def min_value(node, alpha, beta, path):
        print(f"MIN: Visiting node {node.name}, alpha={alpha}, beta={beta}")
        if terminal_test(node):
            print(f"MIN: Terminal node {node.name} has utility {utility(node)}")
            return utility(node), path
        v = float('inf')
        best_path = []
        for child in node.children:
            value, new_path = max_value(child, alpha, beta, path + [child.name])
            print(f"MIN: From node {node.name}, child {child.name} → value={value}")
            if value < v:
                v = value
                best_path = new_path
            if v <= alpha:
                print(f"MIN: Pruning at node {node.name} with value={v} ≤ alpha={alpha}")
                return v, best_path
            beta = min(beta, v)
        print(f"MIN: Returning value={v} for node {node.name}")
        return v, best_path

    print("Starting Alpha-Beta Search...\n")
```

```
    final_value, final_path = max_value(state, float('-inf'), float('inf'), [state.name])
    return final_value, final_path

# Helper Functions

# Terminal test function (checks if a node is terminal)
def terminal_test(node):
    return node.value is not None  # If the node has a value, it's a terminal node

# Utility function (returns the utility of a terminal node)
def utility(node):
    return node.value  # Return the value of the terminal node

# Example usage:

# Create the terminal nodes (leaf nodes)
H = Node('H', value=10)
I = Node('I', value=9)
J = Node('J', value=14)
K = Node('K', value=18)
L = Node('L', value=5)
M = Node('M', value=4)
N = Node('N', value=50)
O = Node('O', value=3)

# Create the non-terminal nodes with children
D = Node('D', children=[H, I])
E = Node('E', children=[J, K])
F = Node('F', children=[L, M])
G = Node('G', children=[N, O])

# Create the parent nodes
B = Node('B', children=[D, E])
C = Node('C', children=[F, G])

# Create the root node
A = Node('A', children=[B, C])

# Perform Alpha-Beta Search starting from the root node 'A'
final_value, final_path = alpha_beta_search(A)
print(f"Best path: {final_path}, with final value: {final_value}")
```

OUTPUT

```
Starting Alpha-Beta Search...

MAX: Visiting node A, alpha=-inf, beta=inf
MIN: Visiting node B, alpha=-inf, beta=inf
MAX: Visiting node D, alpha=-inf, beta=inf
MIN: Visiting node H, alpha=-inf, beta=inf
MIN: Terminal node H has utility 10
MAX: From node D, child H → value=10
MIN: Visiting node I, alpha=10, beta=inf
MIN: Terminal node I has utility 9
MAX: From node D, child I → value=9
MAX: Returning value=10 for node D
MIN: From node B, child D → value=10
MAX: Visiting node E, alpha=-inf, beta=10
MIN: Visiting node J, alpha=-inf, beta=10
MIN: Terminal node J has utility 14
MAX: From node E, child J → value=14
MAX: Pruning at node E with value=14 ≥ beta=10
MIN: From node B, child E → value=14
MIN: Returning value=10 for node B
MAX: From node A, child B → value=10
MIN: Visiting node C, alpha=10, beta=inf
MAX: Visiting node F, alpha=10, beta=inf
MIN: Visiting node L, alpha=10, beta=inf
MIN: Terminal node L has utility 5
MAX: From node F, child L → value=5
MIN: Visiting node M, alpha=10, beta=inf
MIN: Terminal node M has utility 4
MAX: From node F, child M → value=4
MAX: Returning value=5 for node F
MIN: From node C, child F → value=5
MIN: Pruning at node C with value=5 ≤ alpha=10
MAX: From node A, child C → value=5
MAX: Returning value=10 for node A
Best path: ['A', 'B', 'D', 'H'], with final value: 10
```