VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Supriya S(1BM22CS350)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Supriya S (1BM22CS350),** 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.

Sunayana S Assistant Professor Department of CSE, BMSCE	Dr. Jyothi S Nayak Professor & HOD Department of CSE, BMSCE

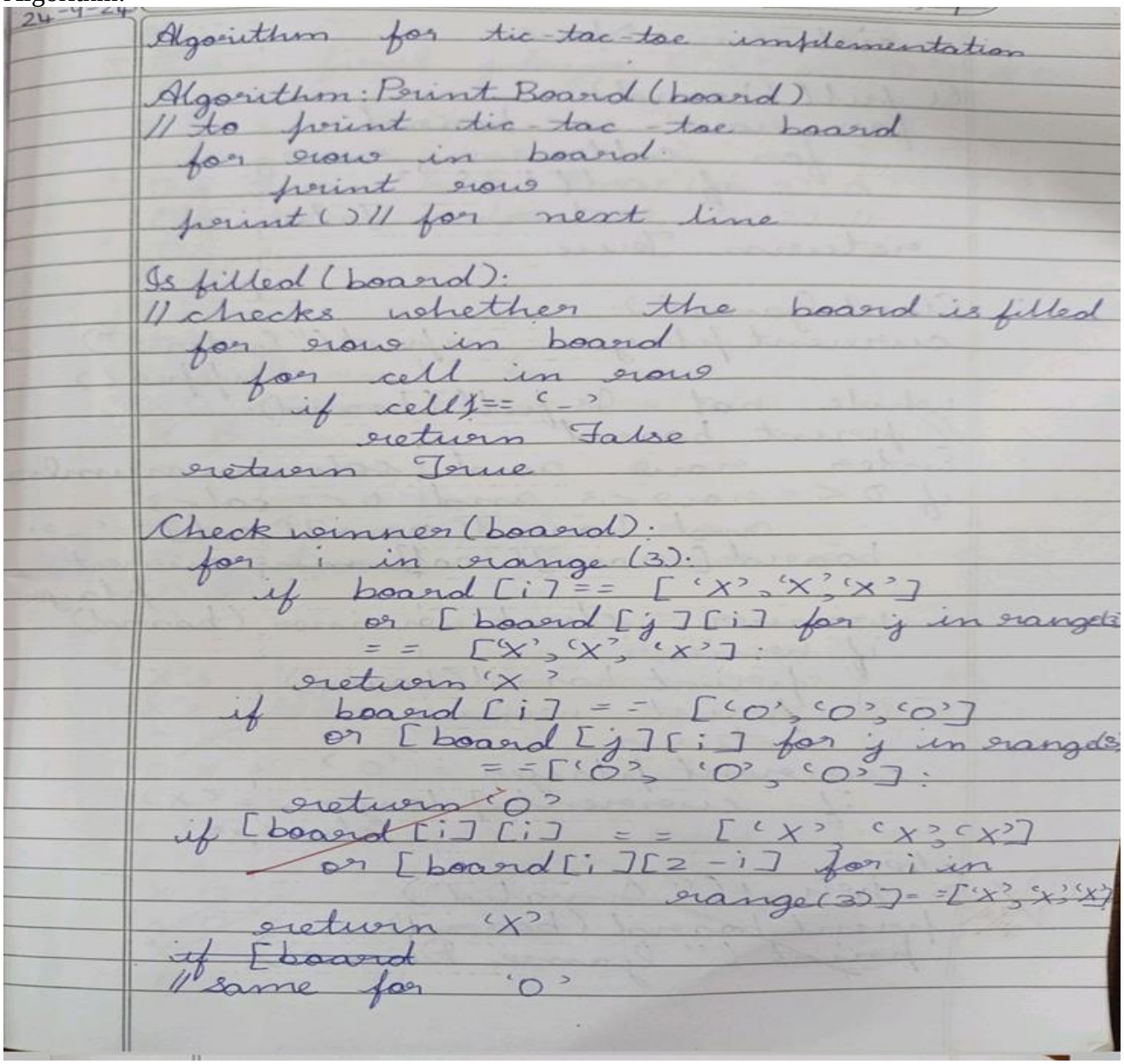
Index

Github Link:
https://github.com/SupriyaS26/AI5thSem.git

Program 1

Implement Tic –Tac –Toe Game

Algorithm:



```
return None

Is_full(board):
    for row in board
        for cell in row
            if cell != '_'
                return False
    return True

Start_game():
current_player = input("Enter : ").
                                upper()
while not Is_full(board):
// print board
Enter row and column numbers
if 0 <= row <3 and 0 <= col <3
        and board[row][col] == '_':
    board[row][col] = current
                                player
    winner = check_winner(board)
    if winner:
        print_board(board)
        print winner
        return
    current_player = 'O'
    if current_player == 'X'
                        else 'X'
else:
print("Invalid")
print_board(board)
print("Game Draw")
```

24/9/2024

Code:

```
# Initialize the board
tic_tac_toe_board = [['-' for _ in range(3)] for _ in range(3)]

def print_board(board):
    for row in board:
        print(" ".join(row))
    print()

def is_filled(board):
    return all(cell != '-' for row in board for cell in row)

def check_winner(board):
    # Check rows
    for row in board:
        if row == ['X', 'X', 'X']:
            print("X is the winner")
            return 'X'
        if row == ['O', 'O', 'O']:
            print("O is the winner")
            return 'O'

    # Check columns
    for col in range(3):
        if all(board[row][col] == 'X' for row in range(3)):
            print("X is the winner")
            return 'X'
        if all(board[row][col] == 'O' for row in range(3)):
            print("O is the winner")
            return 'O'

    # Check diagonals
    if all(board[i][i] == 'X' for i in range(3)):
        print("X is the winner")
        return 'X'
    if all(board[i][2 - i] == 'X' for i in range(3)):
        print("X is the winner")
        return 'X'
    if all(board[i][i] == 'O' for i in range(3)):
        print("O is the winner")
        return 'O'
    if all(board[i][2 - i] == 'O' for i in range(3)):
        print("O is the winner")
        return 'O'
```

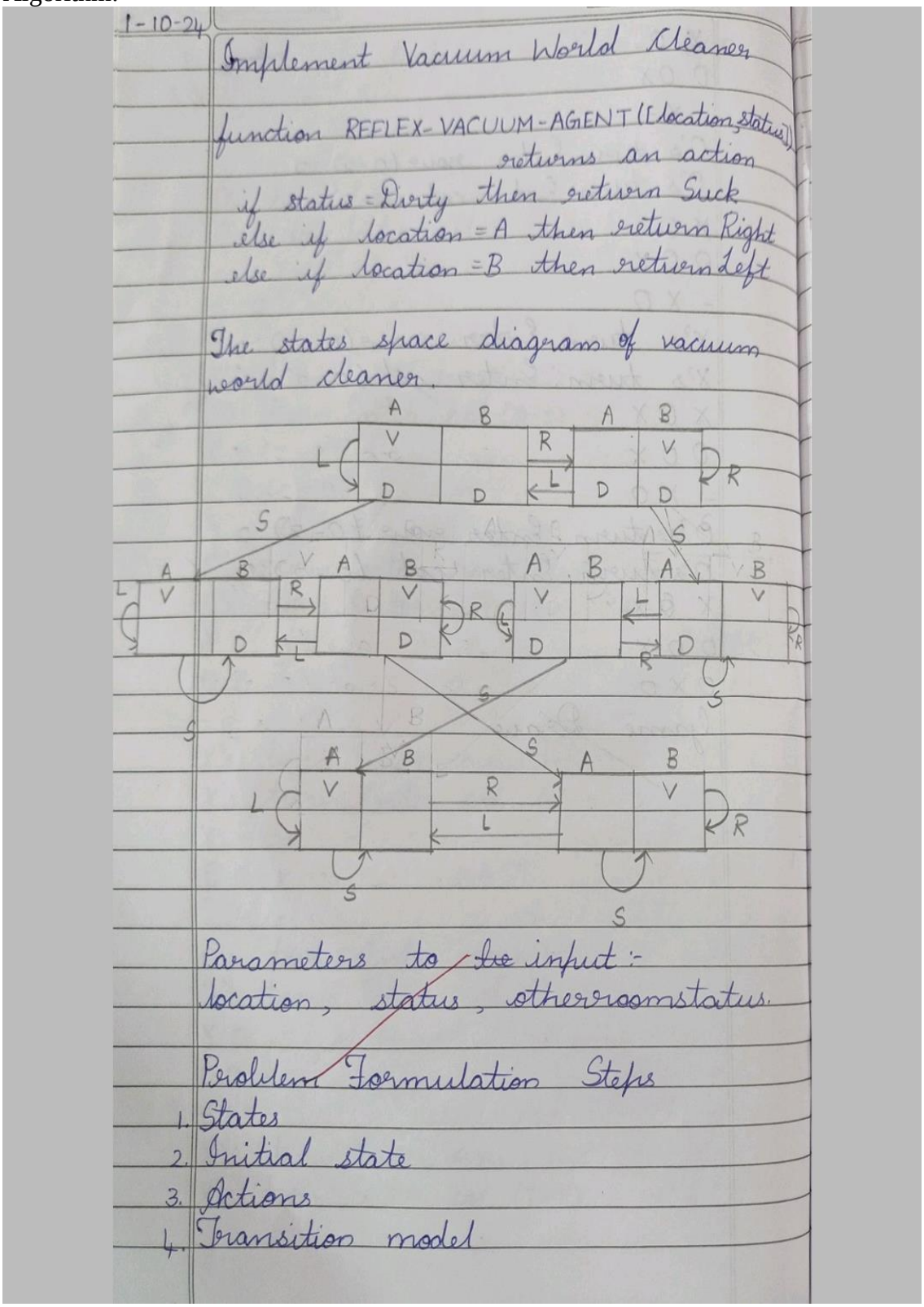
```
    if is_filled(board):
        print("Game Draw")
        return 'Draw'

    return None

def start_game():
    board = [['-' for _ in range(3)] for _ in range(3)]
    current_player = input("Enter first player ('X' or 'O'): ").upper()

    while not is_filled(board):
        print_board(board)
        row = int(input(f"{current_player}'s turn. Enter row (0-2): "))
        col = int(input(f"{current_player}'s turn. Enter col (0-2): "))

        if 0 <= row < 3 and 0 <= col < 3 and board[row][col] == '-':
            board[row][col] = current_player
            winner = check_winner(board)
            if winner:
                print_board(board)
                return
            current_player = 'O' if current_player == 'X' else 'X'
        else:
            print("Invalid move! Cell already filled or out of bounds. Try
again.")

    print_board(board)

start_game()
```

Output:

```
Enter first player ('X' or 'O'): X
- - -
- - -
- - -

X's turn. Enter row (0-2): 0
X's turn. Enter col (0-2): 0
X - -
- - -
- - -

O's turn. Enter row (0-2): 0
O's turn. Enter col (0-2): 1
X O -
- - -
- - -

X's turn. Enter row (0-2): 1
X's turn. Enter col (0-2): 1
X O -
- X -
- - -

O's turn. Enter row (0-2): 2
O's turn. Enter col (0-2): 2
X O -
- X -
- - O

X's turn. Enter row (0-2): 1
X's turn. Enter col (0-2): 0
X O -
X X -
- - O

O's turn. Enter row (0-2): 1
O's turn. Enter col (0-2): 2
X O -
X X O
- - O

X's turn. Enter row (0-2): 2
X's turn. Enter col (0-2): 1
X O -
X X O
- X O

O's turn. Enter row (0-2): 0
O's turn. Enter col (0-2): 2
O is the winner
X O O
X X O
- X O
```

Implement vacuum cleaner agent

Algorithm:



5. Goal Test
6. Path cost

Algorithm for Two Quadrants

1. Declare goal = [ ]
2. Input room location, status of room location (A or B) as 0 for clean and 1 for dirty and other room status
3. cost ← 0
4. if room location == 'A'
   goal [,] ← status
   goal [3] ← other room
   // display initial goal state
   print "Vacuum is placed in location A"
   if status == 0
   print "Location A is already clean"
   if status == 1
   print "Location A is Dirty"
   cost ← cost + 1
   goal [1] ← 0
   print "Location A has been cleaned"
   print "Moving right to location B"
   print "Vacuum is placed in location B"
   if other room == 1
   print "Location B is dirty"
   cost ← cost + 1
   goal [3] ← 0
   print "Location B has been cleaned"
else
   print "Location B is already clean"

```
//display final goal state and cost
  for suck
elif "room location == 'B':
   goal[3] ← status
   goal[1] ← other room
   //display initial goal state
   //display "Vacuum is placed in location B"
   if status == 0
      print "location B is already clean"
   if status == 1
      print "location B is Dirty"
      cost ← cost + 1
      goal[3] ← 0
      print "location B has been cleaned"
   print "Moving left to location A"
   print "Vacuum is placed in location A"
   if other room == 1
      print "location A is Dirty
      cost ← cost + 1
      goal[1] ← 0
      print "location A has been cleaned"
   if other room == 0
      print("location A is already clean")
   //display final goal state and
     cost for suck
else
   //display "Invalid input"
<end of if>
```

5. Add battery level and check if its greater than zero else print "Battery low", decrement it each time

Code:

```
goal = ['A', 1, 'B', 1]
battery_level = int(input("Enter the battery level: "))
room_location = input("Enter the room location A or B: ")
status = int(input("Enter status of the room (0 for clean, 1 for dirty): "))
other_room = int(input("Enter other room status (0 or 1): "))
cost = 0
def clean_room(location, room_status):
    global battery_level, cost
    if room_status == 0:
        print(f"Location {location} is already clean")
    else:
        print(f"Location {location} is dirty")
        cost += 1
        print(f"Location {location} has been cleaned")
    battery_level -= 1
if battery_level > 0:
    if room_location == 'A':
        goal[1], goal[3] = status, other_room
        print(f"Initial Goal state {goal}\nVacuum is placed in Location A")
        if battery_level > 0:
            clean_room('A', status)
            print("Moving to Location B")
            clean_room('B', other_room)
    elif room_location == 'B':
        goal[3], goal[1] = status, other_room
        print(f"Initial Goal state {goal}\nVacuum is placed in Location B")
        if battery_level > 0:
            clean_room('B', status)
            print("Moving to Location A")
            clean_room('A', other_room)
    print(f"Goal state is {goal}\nCost for suck is {cost}")
else:
    print("Invalid input or Battery Low!")
```

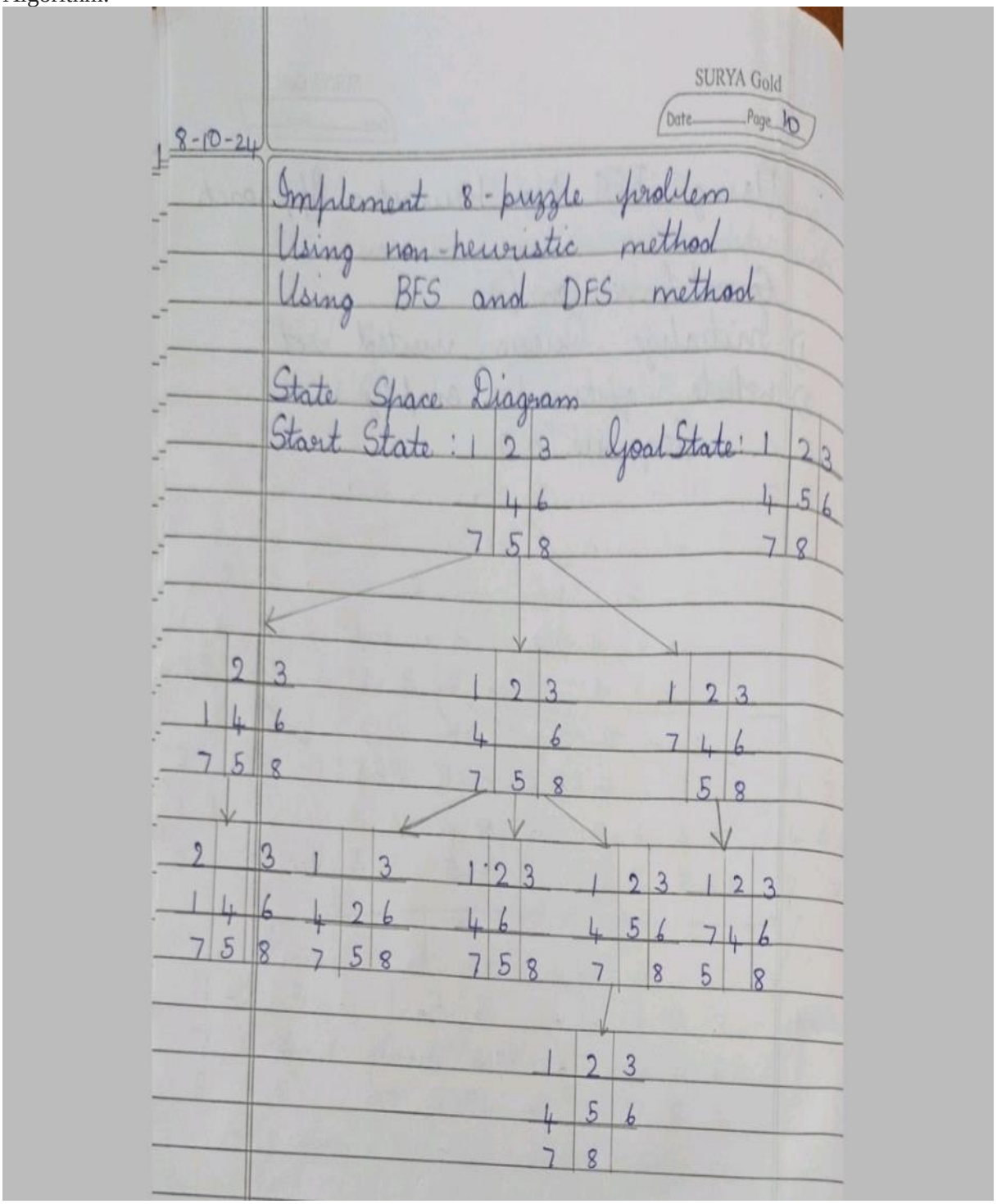
Output:

```
Enter the battery level: 5
Enter the room location A or B: A
Enter status of the room (0 for clean, 1 for dirty): 1
Enter other room status (0 or 1): 0
Initial Goal state ['A', 1, 'B', 0]
Vacuum is placed in Location A
Location A is dirty
Location A has been cleaned
Moving to Location B
Location B is already clean
Goal state is ['A', 1, 'B', 0]
Cost for suck is 1
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)

Algorithm:



Algorithm for using DFS method

```
class BoardNode:
    initialize (board, parent = None, move = None)
    is_goal (goal_state)
    __hash__()
    __eq__(other)
    __repr__()
class DFS:
    initialize (initial_board, goal_state)
    search():
        visited = empty set
        call dfs (initial_node, visited)
        if solutions found:
            print solutions
        print total unique states, total
                permutations
    _dfs(node, visited):
        if node is goal state:
            append solution path to found
                                solutions
            return True
        mark node as visited
        add current board to unique_states
        increment permutation_count
```

```
for each successor in get_successors(node):
    if successor not in visited:
        call dfs(successor, visited)
get_successors(node):
    for each move in [UP, DOWN, LEFT, RIGHT]:
        if valid move:
            create new board
            append new BoardNode
            (new_board, node, move) to successors
find_zero(board):
    return coordinates of zero
get_solution_path(node):
    while node has a parent:
        append move to path
        move to parent node
    return reversed(path)
print_solution(path):
    print solution path
function get_user_input():
    read & return 3x3 board
main():
    get initial & goal states
```

O/P:

Enter initial state: 1 2 3
0 4 6
7 5 8

Enter goal state: 1 2 3
4 5 6
7 8 0

9/10/2024

Total unique states encountered: 852

Code:

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

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

    def __hash__(self):
        return hash(str(self.board))

    def __eq__(self, other):
        return self.board == other.board

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

class DFS:
    def __init__(self, initial_board, goal_state):
        self.initial_node = BoardNode(initial_board)
        self.goal_state = goal_state
        self.found_solutions = []
        self.unique_states = set()  # Set to track unique states
        self.permutation_count = 0   # Counter for permutations

    def search(self):
        visited = set()
        self._dfs(self.initial_node, visited)

        # Print solutions and states
        if self.found_solutions:
            for solution in self.found_solutions:
                self.print_solution(solution)
        else:
            print("No solution found.")

        # Print unique states encountered
        print(f"Total unique states encountered: {len(self.unique_states)}")
        print(f"Total permutations encountered: {self.permutation_count}")

    def _dfs(self, node, visited):
        if node.is_goal(self.goal_state):
```

```
            self.found_solutions.append(self.get_solution_path(node))
            return True

        visited.add(node)  # Add the current node to visited set
        # Add current state as a hashable tuple
        self.unique_states.add(tuple(map(tuple, node.board)))  # Add current
state to unique states
        self.permutation_count += 1  # Increment permutation count

        for neighbor in self.get_successors(node):
            if neighbor not in visited:
                if self._dfs(neighbor, visited):
                    return True

        return False

    def get_successors(self, node):
        board = node.board
        x, y = self.find_zero(board)
        successors = []
        moves = [('UP', -1, 0), ('DOWN', 1, 0), ('LEFT', 0, -1), ('RIGHT', 0,
1)]

        for move_name, dx, dy in moves:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:  # Check bounds
                new_board = [row[:] for row in board]  # Make a copy
                new_board[x][y], new_board[new_x][new_y] =
new_board[new_x][new_y], new_board[x][y]
                successors.append(BoardNode(new_board, node, move_name))

        return successors

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

    def get_solution_path(self, node):
        path = []
        while node:
            if node.move:
                path.append(node.move)
            node = node.parent
        return path[::-1]  # Reverse path to show from start to goal
```

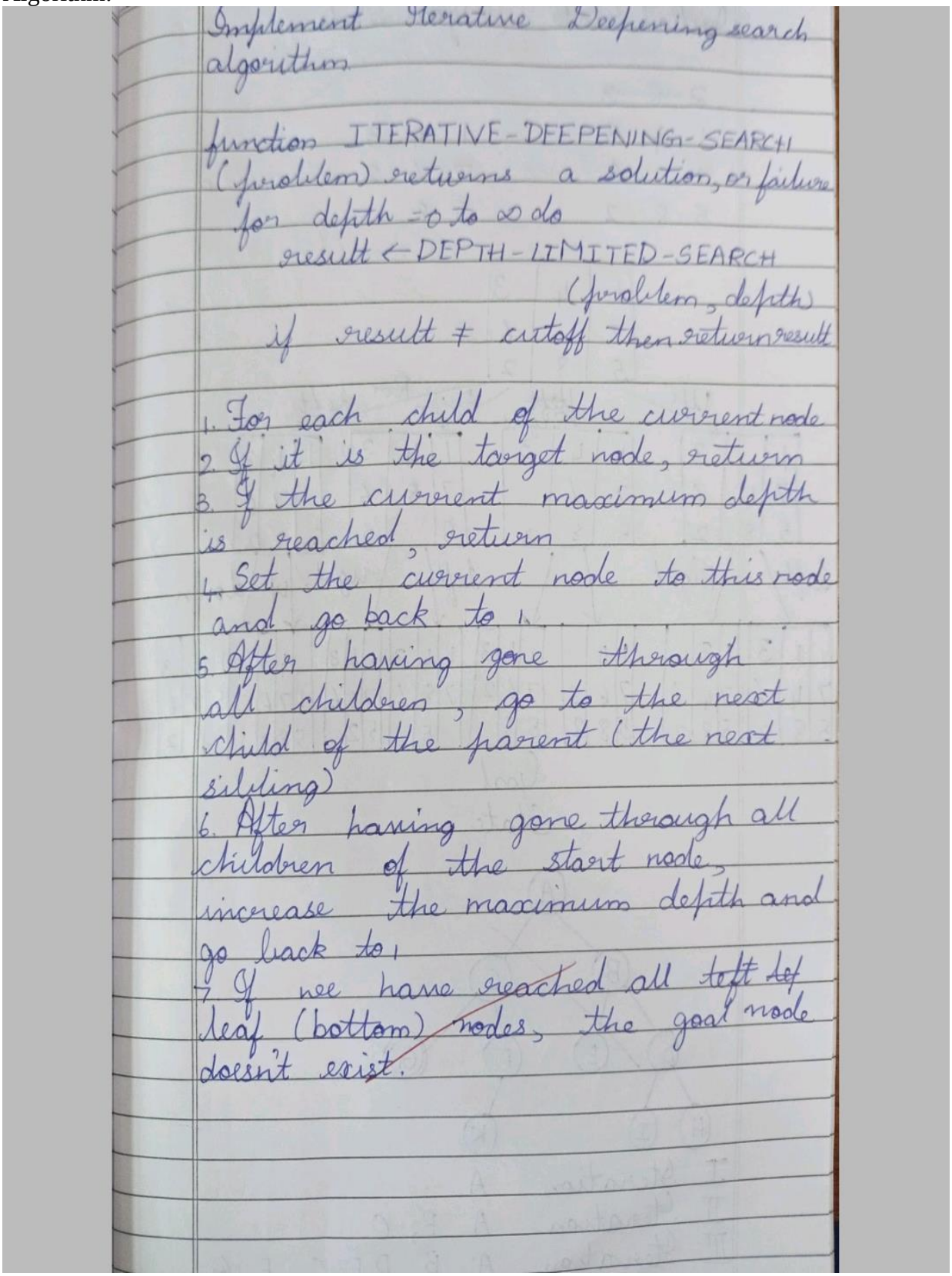
```
    def print_solution(self, path):
        print("Solution moves:", path)

def get_user_input():
    print("Enter the initial state (3 rows of 3 numbers, use 0 for empty
space):")
    board = []
    for _ in range(3):
        row = list(map(int, input().strip().split()))
        board.append(row)
    return board

def main():
    initial_board = get_user_input()

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

    dfs_solver = DFS(initial_board, goal_state)
    dfs_solver.search()

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

Output:

```
Enter the initial state (3 rows of 3 numbers, use 0 for empty space):
1 2 3
0 4 6
7 5 8
Enter the goal state (3 rows of 3 numbers, use 0 for empty space):
1 2 3
4 5 6
7 8 0
Total unique states encountered: 852
Total permutations encountered: 852
```

Implement Iterative deepening search algorithm

Algorithm:



Implement Iterative Deepening search algorithm

function ITERATIVE-DEEPENING-SEARCH (problem) returns a solution, or failure
for depth = 0 to ∞ do
result ← DEPTH-LIMITED-SEARCH (problem, depth)
if result ≠ cutoff then return result

1. For each child of the current node
2. If it is the target node, return
3. If the current maximum depth is reached, return
4. Set the current node to this node and go back to 1.
5. After having gone through all children, go to the next child of the parent (the next sibling)
6. After having gone through all children of the start node, increase the maximum depth and go back to 1.
7. If we have reached all ~~left~~ leaf (bottom) nodes, the goal node doesn't exist.

SURYA Gold

Date Page

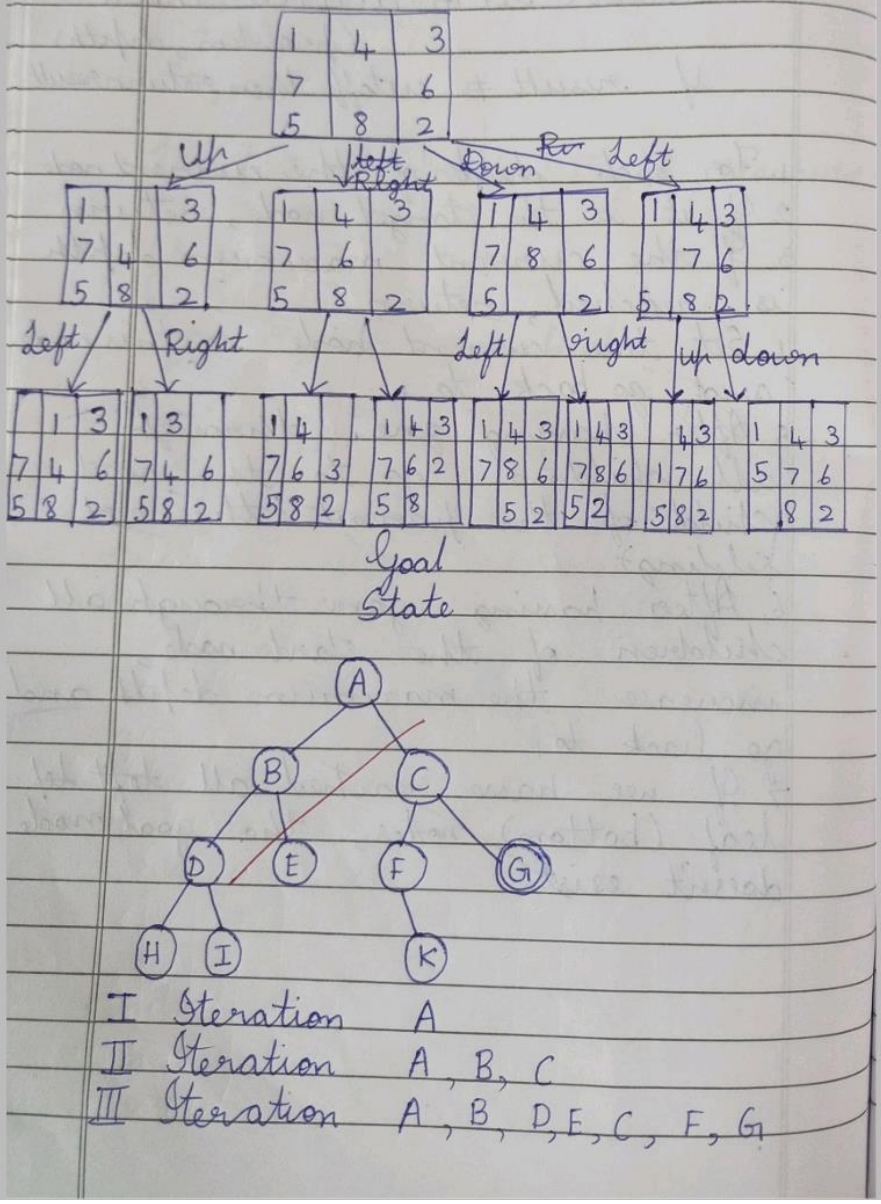
Initial State Goal State

~~2 8 3~~

1 4 3 1 4 3

7 6 7 6 2

5 8 2 5 8



I Iteration A

II Iteration A, B, C

III Iteration A, B, D, E, C, F, G

Code:

```
class PuzzleState:
    def __init__(self, board, empty_tile_pos, moves=0, prev=None,
direction=None):
        self.board = board
        self.empty_tile_pos = empty_tile_pos
        self.moves = moves
        self.prev = prev
        self.direction = direction
        self.size = 3  # 3x3 board

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

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

        for dr, dc, dir_name in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < self.size and 0 <= new_col < self.size:
                new_empty_tile_pos = new_row * self.size + new_col
                moves.append((new_empty_tile_pos, dir_name))

        return moves

    def move(self, new_empty_tile_pos, direction):
        new_board = self.board[:]
        new_board[self.empty_tile_pos], new_board[new_empty_tile_pos] =
new_board[new_empty_tile_pos], new_board[self.empty_tile_pos]
        return PuzzleState(new_board, new_empty_tile_pos, self.moves + 1,
self, direction)

    def __repr__(self):
        return '\n'.join([' '.join([str(self.board[i * self.size + j]) for j
in range(self.size)]) for i in range(self.size)]) + f'\nMoves: {self.moves}'


def depth_limited_search(state, limit):
    if state.is_goal():
        return state
    if state.moves >= limit:
```

```
        return None

    for new_empty_tile_pos, direction in state.get_possible_moves():
        new_state = state.move(new_empty_tile_pos, direction)
        result = depth_limited_search(new_state, limit)
        if result:
            return result

    return None


def iterative_deepening_search(initial_state, limit):
    for depth in range(0, limit + 1, 2):  # Increase depth by 2 for even
limits
        print(f"Searching at depth: {depth}")
        states_at_level = []
        result = depth_limited_search_with_states(initial_state, depth,
states_at_level)
        print_states(states_at_level, depth)
        if result:
            return result
    return None


def depth_limited_search_with_states(state, limit, states_at_level):
    if state.is_goal():
        return state
    if state.moves >= limit:
        return None

    # Store the state and direction
    states_at_level.append((state, state.direction))

    for new_empty_tile_pos, direction in state.get_possible_moves():
        new_state = state.move(new_empty_tile_pos, direction)
        result = depth_limited_search_with_states(new_state, limit,
states_at_level)
        if result:
            return result

    return None


def print_states(states_at_level, depth):
    if not states_at_level:
        print(f"No states found at depth {depth}.")
```

```
        return

    print(f"States at depth {depth}:")
    for state, direction in states_at_level:
        print(f"Direction: {direction}, State:\n{state}\n")


def main():
    # Get user input for the initial state of the puzzle
    print("Enter the initial state of the 8-puzzle (use 0 for the empty
tile):")
    initial_board = list(map(int, input().strip().split()))

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

    empty_tile_pos = initial_board.index(0)
    initial_state = PuzzleState(initial_board, empty_tile_pos)

    # Get user input for limit
    limit = int(input("Enter the maximum depth limit (even number): "))
    if limit % 2 != 0:
        print("Limit must be an even number!")
        return

    # Run IDS
    solution = iterative_deepening_search(initial_state, limit)

    # Print the solution path
    if solution:
        path = []
        while solution:
            path.append(solution)
            solution = solution.prev
        for step in reversed(path):
            print(step)
    else:
        print("No solution found within the specified limit.")


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

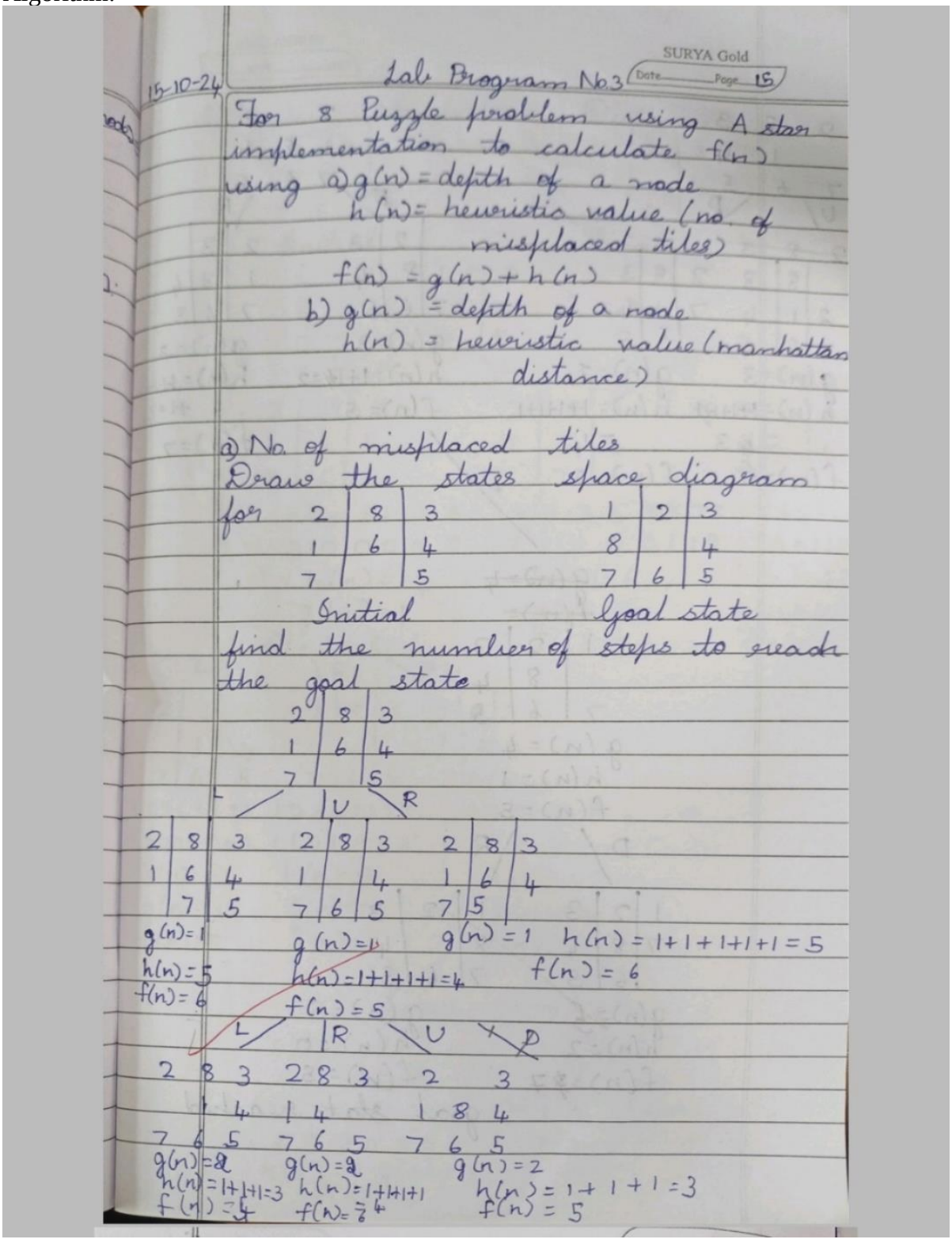
Output:

```
Enter the initial state of the 8-puzzle (use 0 for the empty tile):
1 4 3 7 0 6 5 8 2
Enter the maximum depth limit (even number): 2
Searching at depth: 0
No states found at depth 0.
Searching at depth: 2
States at depth 2:
Direction: None, State:
1 4 3
7 0 6
5 8 2
Moves: 0

Direction: Up, State:
1 0 3
7 4 6
5 8 2
Moves: 1

Direction: Down, State:
1 4 3
7 8 6
5 0 2
Moves: 1

Direction: Left, State:
1 4 3
0 7 6
5 8 2
Moves: 1

Direction: Right, State:
1 4 3
7 6 0
5 8 2
Moves: 1

No solution found within the specified limit.
```

Program 3

Implement A* search Algorithm

Algorithm:



b) Using Manhattan Distance

2	8	3
1	6	4
7		5

Initial

1	2	3
8		4
7	6	5

Final

2	8	3
1	6	4
7		5

U L R

2	8	3
1		4
7	6	5

2	8	3
1	6	4
7		5

2	8	3
1	6	4
7	5	

1 2 3 4 5 6 7 8 — 1 1 0 0 0 0 0 2 — d = h(n) = 4, g(n) = 1, f(n) = 5

1 2 3 4 5 6 7 8 — 1 1 0 0 0 1 1 2 — h(n) = 6, g(n) = 1, f(n) = 7

1 2 3 4 5 6 7 8 — 1 1 0 0 1 1 0 2 — h(n) = 6, g(n) = 1, f(n) = 7

L R U

2	8	3
	1	4
7	6	5

2	8	3
1		4
7	6	5

2	8	3
	1	4
7	6	5

1 2 3 4 5 6 7 8 — 1 2 0 0 0 0 0 2 — h(n) = 5, g(n) = 2, f(n) = 7

1 2 3 4 5 6 7 8 — 1 1 0 1 0 0 0 2 — h(n) = 5, g(n) = 2, f(n) = 7

1 2 3 4 5 6 7 8 — 1 1 0 0 0 0 0 1 — h(n) = 3, g(n) = 2, f(n) = 5

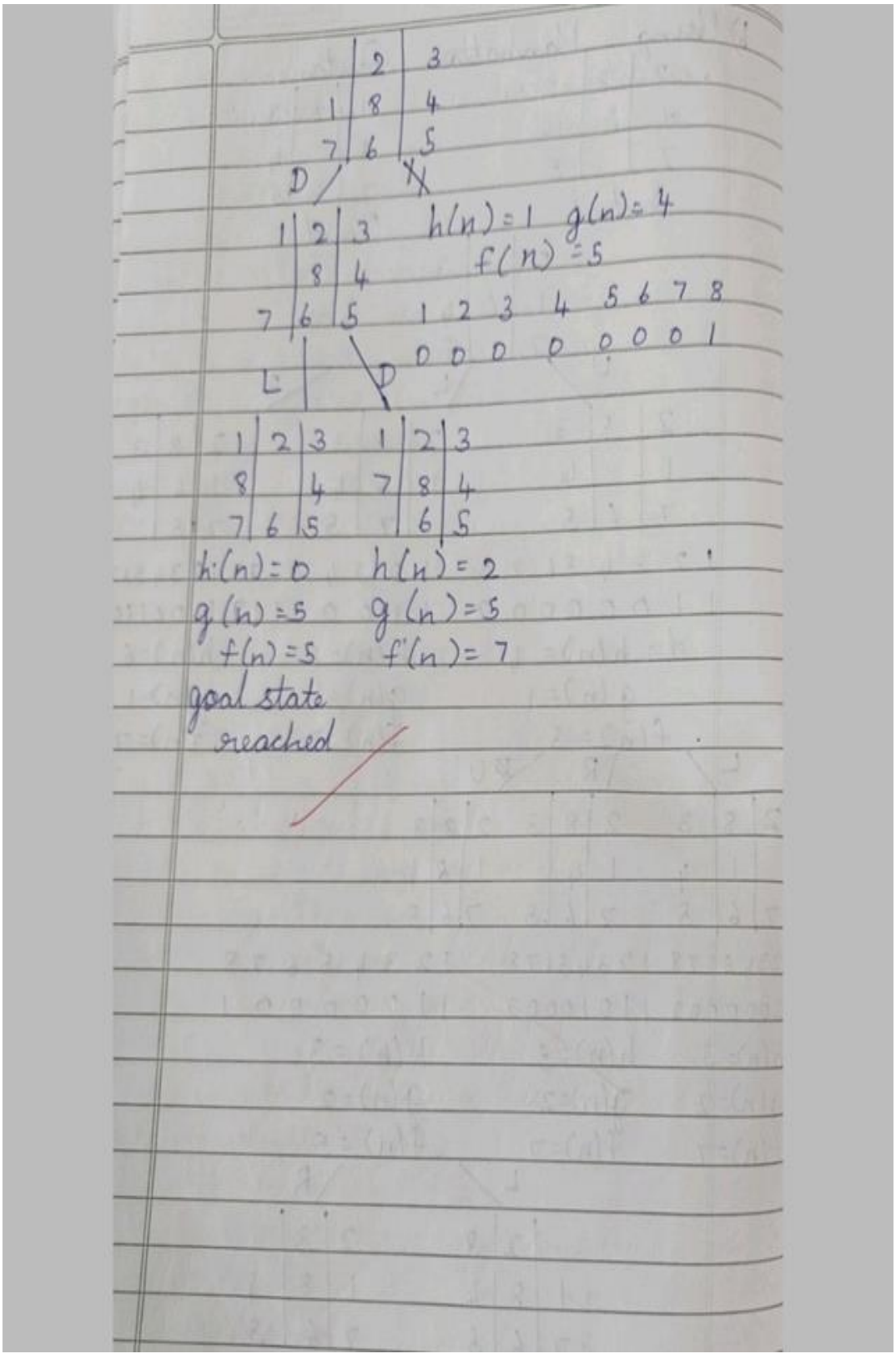
L R

	2	3
1	8	4
7	6	5

2	3	
1	8	4
7	6	5

1 2 3 4 5 6 7 8 — 1 0 0 0 0 0 0 1 — h(n) = 2, g(n) = 3, f(n) = 5

1 2 3 4 5 6 7 8 — 1 1 1 0 0 0 0 1 — h(n) = 4, g(n) = 3, f(n) = 7


2 3
1 8 4
7 6 5
D
X
1 2 3 h(n) = 1 g(n) = 4
8 4 f(n) = 5
7 6 5 1 2 3 4 5 6 7 8
0 0 0 0 0 0 0 1
L
D
1 2 3 1 2 3
8 4 7 8 4
7 6 5 6 5
h(n) = 0 h(n) = 2
g(n) = 5 g(n) = 5
f(n) = 5 f(n) = 7
goal state
reached

a) No. of misplaced tiles

```
Algorithm: Astarmisplacedtiles(start,goal)
// to find the no. of misplaced tiles
h=0
for i in range(3):
    for j in range(3):
        if (start[i][j] != goal[i][j])
            h=h+1
return h-1

Algorithm: main()
//Input start[][] and goal[][]
g=0
visited=[]
f=999
while (start != goal):
    if start not in visited:
        g=g+1
        emptytile = locateemptytile(start)
        for each move, UP, DOWN, RIGHT, LEFT
            and update in start, the empty tile
            h = Astarmisplacedtiles
                (start, goal)
        if ((g+h)<f):
            f=g+h
            visited.append(start[][])
// print f
Algorithm: locateemptytile(start)
    for i in range(3) (len(start)):
        for j in range(3):
            if start[i][j] == 0:
                return i, start[i][j]
```

b) Manhattan distance

Algorithm: Manhattan distance (start, goal)

```
h=0
for i in [1,2,3,4,5,6,7,8]:
    ~~a = indexof~~
    store index of a i in start and store in a
    store index of i in goal and store in b
    h = h+abs(a-b)
    ~~return h~~
return h
```

Algorithm: main()

```
// Input start and goal
g=0
visited = []
f=999
while (start != goal):
  if start not in visited
    g=g+1
    emptytile = locateemptytile(start)
    for each move UP, DOWN, LEFT, RIGHT, update in start
        h = Astarmisplacedtiles(start, goal)
        if ((g+h)<f)
          f=g+h
          visited.append(start[][])
// printf
```

Code:

```
import heapq

class PuzzleState:
    def __init__(self, state, empty_tile_pos, g, h, path, level):
        self.state = state
        self.empty_tile_pos = empty_tile_pos  # (row, col)
        self.g = g  # Cost from start to current state
        self.h = h  # Heuristic cost to goal
        self.f = g + h  # Total cost
        self.path = path  # Path taken to reach this state
        self.level = level  # Depth level in the state space

    def __lt__(self, other):
        return self.f < other.f  # Priority queue comparison based on f value

def astar_misplaced_tiles(start_state, goal_state):
    directions = {
        (-1, 0): 'Up',
        (1, 0): 'Down',
        (0, -1): 'Left',
        (0, 1): 'Right'
    }

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

    def generate_moves(state, empty_tile_pos):
        moves = []
        row, col = empty_tile_pos
        for (dr, dc), direction in directions.items():
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_state = [list(r) for r in state]  # Deep copy
                # Swap the empty tile with the adjacent tile
                new_state[row][col], new_state[new_row][new_col] =
new_state[new_row][new_col], new_state[row][col]
                moves.append((new_state, (new_row, new_col), direction))
        return moves
```

```
    visited = set()
    start_empty_pos = next((i, j) for i in range(3) for j in range(3) if
start_state[i][j] == 0)
    start_h = calculate_heuristic(start_state)
    start_node = PuzzleState(start_state, start_empty_pos, 0, start_h,
[start_state], 0)

    priority_queue = []
    heapq.heappush(priority_queue, start_node)

    while priority_queue:
        current_node = heapq.heappop(priority_queue)

        # Check if we reached the goal
        if current_node.state == goal_state:
            print("Goal state reached!")
            for step in current_node.path:
                for row in step:
                    print(row)
                print()
            return

        visited.add(tuple(map(tuple, current_node.state)))  # Add current
state to visited

        # Generate possible moves
        for new_state, new_empty_pos, direction in
generate_moves(current_node.state, current_node.empty_tile_pos):
            if tuple(map(tuple, new_state)) not in visited:
                g = current_node.g + 1  # Cost from start
                h = calculate_heuristic(new_state)  # Heuristic
                new_path = current_node.path + [new_state]
                new_node = PuzzleState(new_state, new_empty_pos, g, h,
new_path, current_node.level + 1)

                # Print state information
                print(f"Level: {new_node.level}, Direction: {direction},
Heuristic: {new_node.h}")
                for row in new_node.state:
                    print(row)
                print()

                heapq.heappush(priority_queue, new_node)

def main():
```

```
    print("Enter initial state (3x3 grid, use 0 for empty tile):")
    start_state = [list(map(int, input().split())) for _ in range(3)]

    print("Enter goal state (3x3 grid, use 0 for empty tile):")
    goal_state = [list(map(int, input().split())) for _ in range(3)]

    astar_misplaced_tiles(start_state, goal_state)

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

Output:

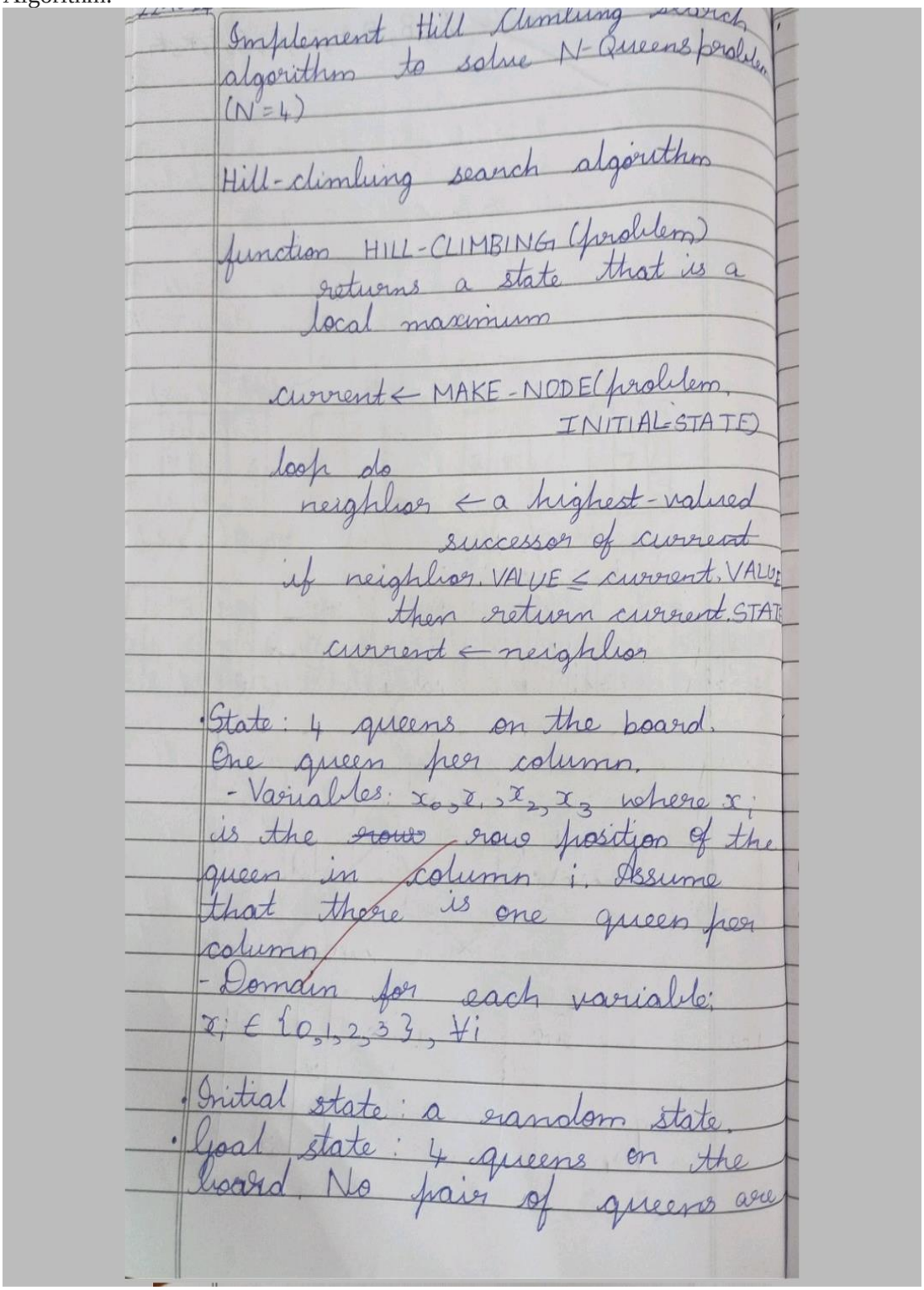
```
Enter initial state (3x3 grid, use 0 for empty tile):
2 8 3
1 6 4
7 0 5
Enter goal state (3x3 grid, use 0 for empty tile):
1 2 3
8 0 4
7 6 5
Level: 1, Direction: Up, Heuristic: 3
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

Level: 1, Direction: Left, Heuristic: 5
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Level: 1, Direction: Right, Heuristic: 5
[2, 8, 3]
[1, 6, 4]
[7, 5, 0]

Level: 2, Direction: Up, Heuristic: 3
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

Level: 2, Direction: Left, Heuristic: 3
[2, 8, 3]
[0, 1, 4]
[7, 6, 5]

Level: 2, Direction: Right, Heuristic: 4
[2, 8, 3]
[1, 4, 0]
[7, 6, 5]

Level: 3, Direction: Left, Heuristic: 2
```

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

Level: 3, Direction: Right, Heuristic: 4
[2, 3, 0]
[1, 8, 4]
[7, 6, 5]

Level: 3, Direction: Up, Heuristic: 3
[0, 8, 3]
[2, 1, 4]
[7, 6, 5]

Level: 3, Direction: Down, Heuristic: 4
[2, 8, 3]
[7, 1, 4]
[0, 6, 5]

Level: 4, Direction: Down, Heuristic: 1
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

Level: 5, Direction: Down, Heuristic: 2
[1, 2, 3]
[7, 8, 4]
[0, 6, 5]

Level: 5, Direction: Right, Heuristic: 0
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

Goal state 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]
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



Implement Hill Climbing search algorithm to solve N-Queens problem (N=4)

Hill-climbing search algorithm

function HILL-CLIMBING (problem)
returns a state that is a local maximum

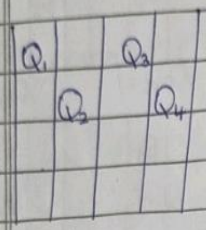
current ← MAKE-NODE(problem, INITIAL-STATE)
loop do
neighbor ← a highest-valued successor of current
if neighbor.VALUE ≤ current.VALUE then return current.STATE
current ← neighbor

- State: 4 queens on the board. One queen per column.
  - Variables: x_0, x_1, x_2, x_3 where x_i is the ~~row~~ row position of the queen in column i. Assume that there is one queen per column
  - Domain for each variable: $x_i \in \{0,1,2,3\}, \forall i$
- Initial state: a random state.
- Goal state: 4 queens on the board. No pair of queens are

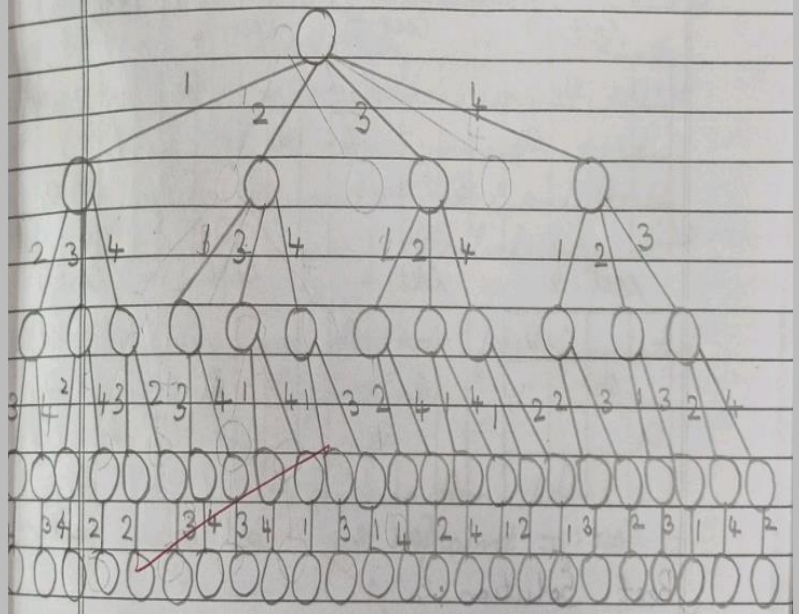
attacking each other

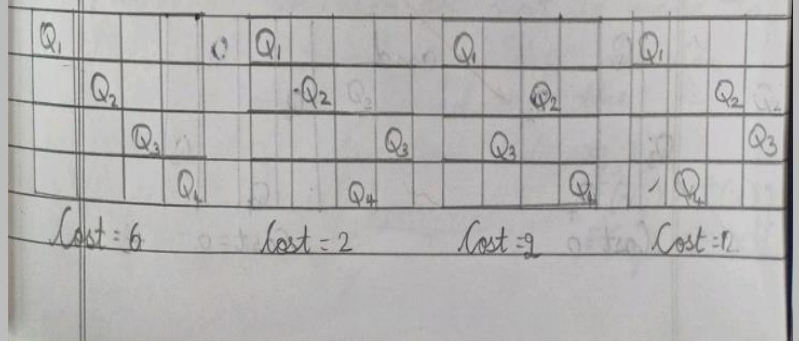
· Neighbour relation:
Swap the row positions of two queens.

· Cost function: The number of pairs of queens attacking each other, directly or indirectly.

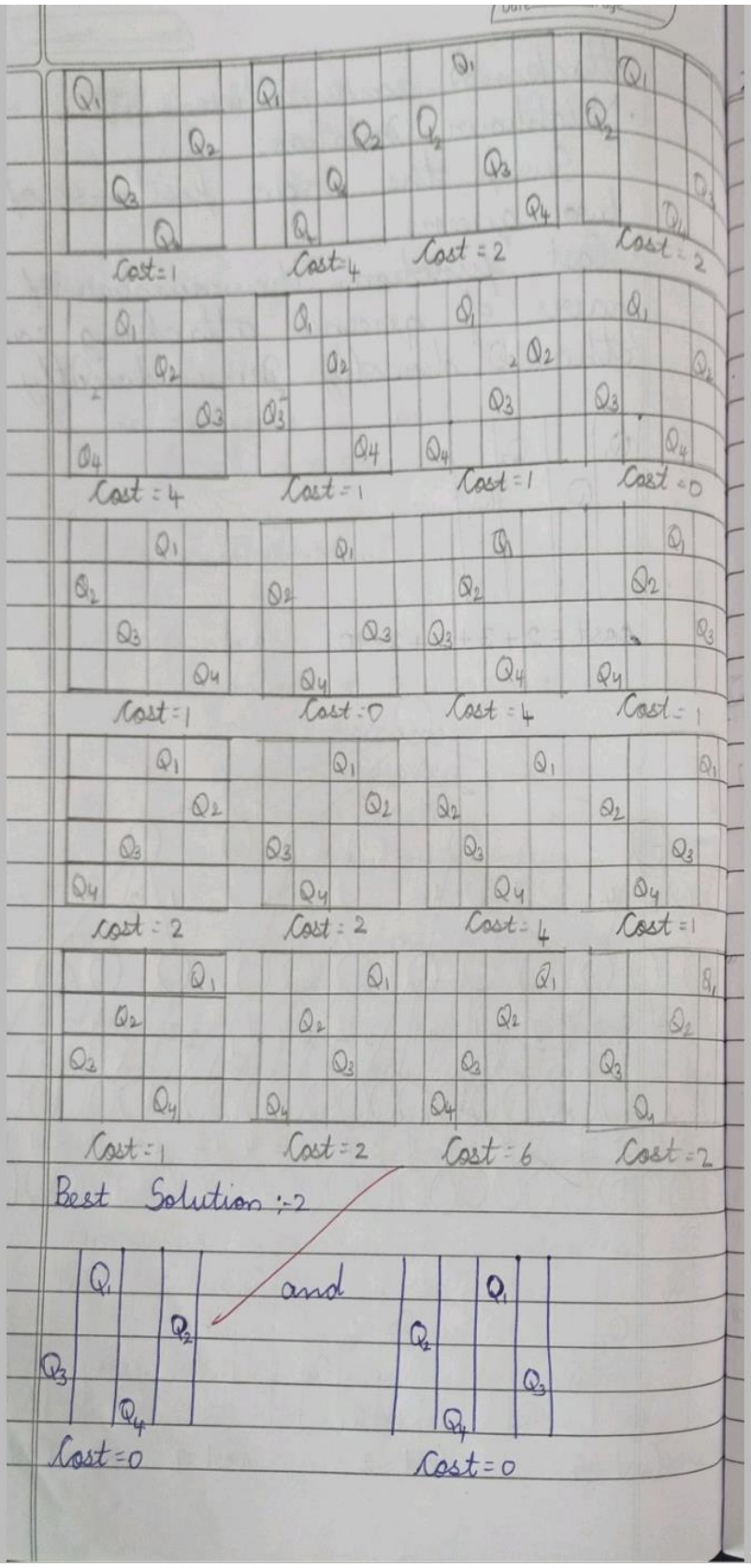


cost = 2 + 3 + 3 + 2 = 10





Cost = 6 Cost = 2 Cost = 2 Cost = 1


Cost=1
Cost=4
Cost=2
Cost=2
Cost=4
Cost=1
Cost=1
Cost=0
Cost=1
Cost=0
Cost=4
Cost=1
Cost=2
Cost=2
Cost=4
Cost=1
Cost=1
Cost=2
Cost=6
Cost=2
Best Solution :-2
and
Cost=0
Cost=0

Code:

```
import random

class NQueens:
    def __init__(self, n):
        self.n = n
        self.solutions = set()  # To store unique solutions

    def random_state(self):
        """Generate a random state (initial placement of queens)."""
        return [random.randint(0, self.n - 1) for _ in range(self.n)]

    def fitness(self, state):
        """Calculate the number of pairs of queens that are attacking each
other."""
        attacks = 0
        for i in range(self.n):
            for j in range(i + 1, self.n):
                if state[i] == state[j] or abs(state[i] - state[j]) == abs(i
- j):
                    attacks += 1
        return attacks

    def get_neighbors(self, state):
        """Generate all neighboring states by moving one queen to a different
row in its column."""
        neighbors = []
        for col in range(self.n):
            for row in range(self.n):
                if row != state[col]:
                    new_state = state[:]
                    new_state[col] = row
                    neighbors.append(new_state)
        return neighbors

    def hill_climbing(self):
        """Perform the hill climbing algorithm to solve the N-Queens
problem."""
        attempts = 0
        while attempts < 1000:  # Limit the number of attempts to avoid
infinite loops
            current = self.random_state()  # Start with a random state
            current_fitness = self.fitness(current)
```

```
            while True:
                # If the current state is a solution
                if current_fitness == 0:
                    self.solutions.add(tuple(current))  # Store unique
solution
                    break

                neighbors = self.get_neighbors(current)
                next_state = None
                next_fitness = float('inf')

                # Evaluate neighbors
                for neighbor in neighbors:
                    neighbor_fitness = self.fitness(neighbor)
                    if neighbor_fitness < next_fitness:
                        next_fitness = neighbor_fitness
                        next_state = neighbor

                # If no better neighbor found, break to start over
                if next_fitness >= current_fitness:
                    break

                current = next_state
                current_fitness = next_fitness

            attempts += 1  # Increment the number of attempts

def print_board(state):
    """Print the board state in a readable format."""
    for row in range(len(state)):
        line = ['Q' if col == state[row] else '.' for col in
range(len(state))]
        print(' '.join(line))
    print("\n")

def main():
    n = int(input("Enter the number of queens (N): "))
    nqueens = NQueens(n)
    nqueens.hill_climbing()

    if nqueens.solutions:
        print(f"Found {len(nqueens.solutions)} unique solutions:")
        for idx, solution in enumerate(nqueens.solutions):
            print(f"Solution {idx + 1}:")
            print_board(solution)
    else:
```

```
        print("No solution found.")

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

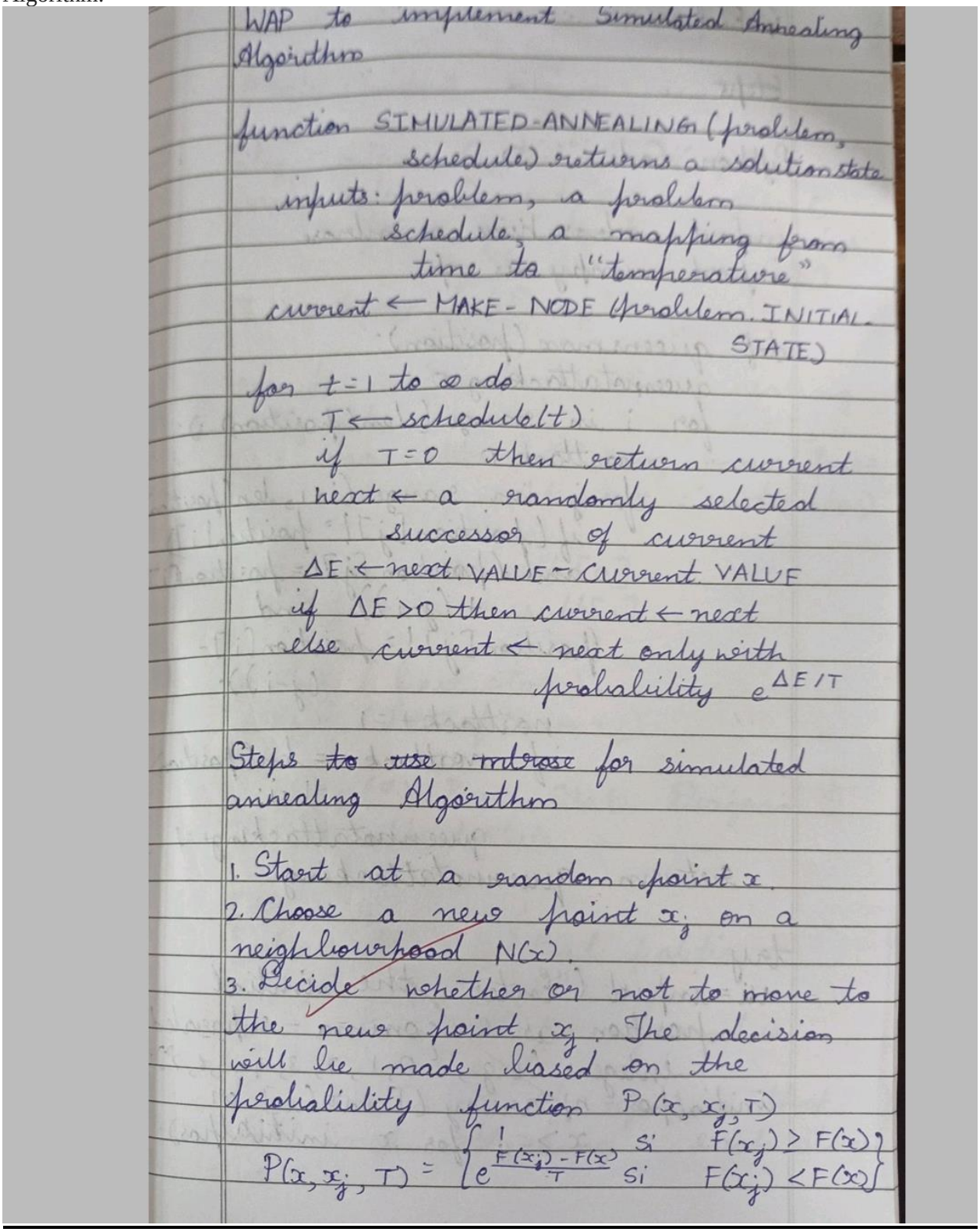
Output:

```
Enter the number of queens (N): 4
Found 2 unique solutions:
Solution 1:
. . Q .
Q . . .
. . . Q
. Q . .

Solution 2:
. Q . .
. . . Q
Q . . .
. . Q .
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



WAP to implement Simulated Annealing 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 $e^{\Delta E/T}$

Steps ~~to use~~ ~~increase~~ for simulated annealing Algorithm

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

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

Code:

```
import mlrose_hiive as mlrose
import numpy as np

def queens_max(position):
    queennotattacking = 0
    for i in range(len(position) - 1):
        noattack = 0
        for j in range(i + 1, len(position)):
            if (position[j] != position[i]) and (position[j] != position[i] +
(j - i)) and (position[j] != position[i] - (j - i)):
                noattack += 1
                if noattack == len(position) - 1 - i:
                    queennotattacking += 1
    return queennotattacking

# Take user input for the initial position
try:
    user_input = input("Enter the initial position as 8 comma-separated
integers (e.g., '0,1,2,3'): ")
    initialpos = np.array([int(x) for x in user_input.split(',')])
    if len(initialpos) != 4 or any(x < 0 or x >= 4 for x in initialpos):
        raise ValueError("Please enter exactly 8 integers between 0 and 7.")
except ValueError as e:
    print(e)
    exit()

# Define the problem and schedule
objective = mlrose.CustomFitness(queens_max)
problem = mlrose.DiscreteOpt(length=4, fitness_fn=objective, maximize=True,
max_val=4)
T = mlrose.ExpDecay()

# Run the simulated annealing algorithm
result = mlrose.simulated_annealing(problem=problem, schedule=T,
max_attempts=500, max_iters=5000, init_state=initialpos)

# Access the best state and best fitness from the result
best_state = result[0]  # Best state
best_fitness = result[1]  # Best fitness

print('The best position found is:', best_state)
```

```
print('The number of queens that are not attacking each other is:',
best_fitness+1)

# Print the diagram of the best state
print("\nBest State Diagram:")
board = [['.' for _ in range(4)] for _ in range(4)]
for row, col in enumerate(best_state):
    board[col][row] = 'Q'  # Place queen

# Print the board
for row in board:
    print(' '.join(row))
```

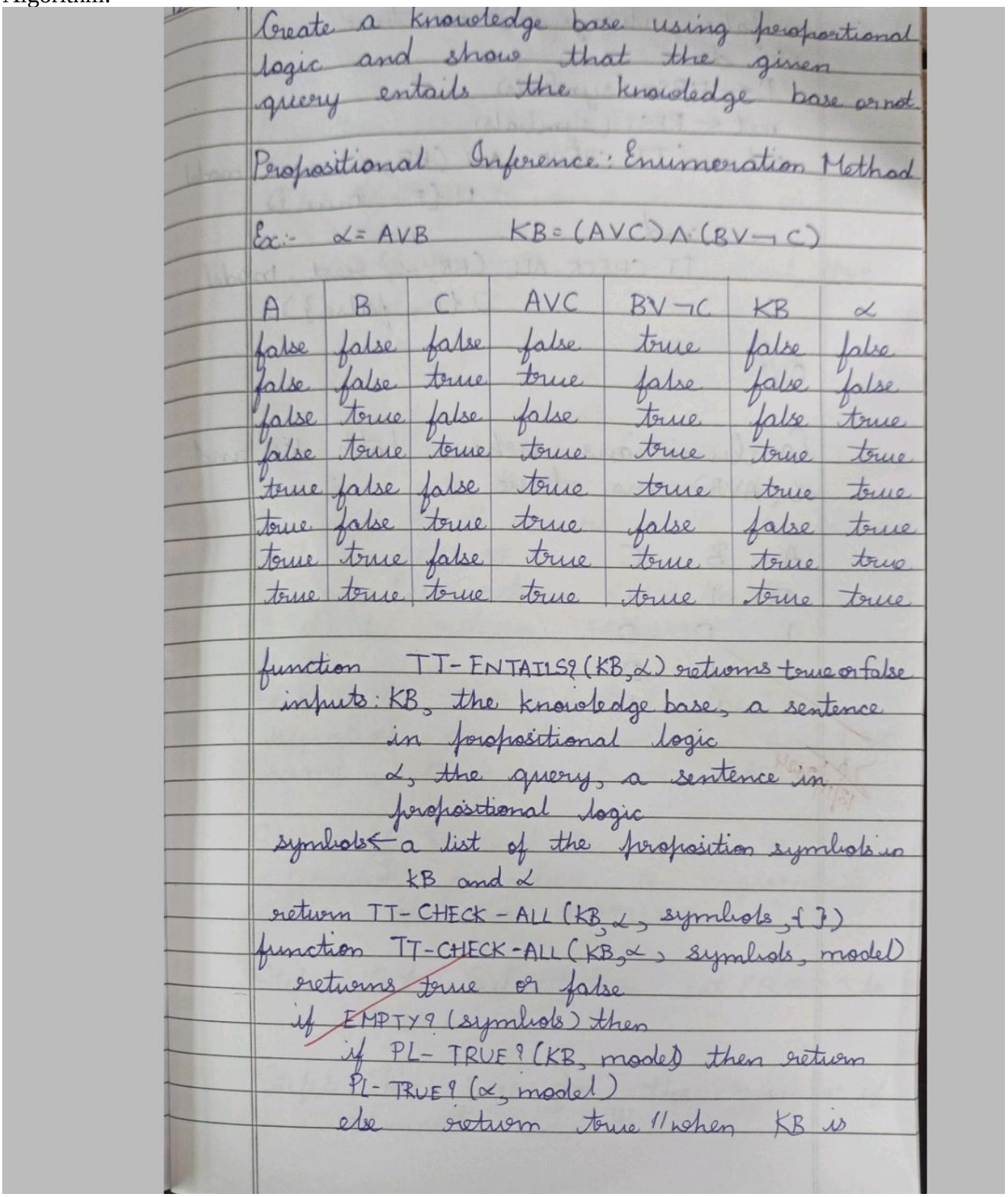
Output:

```
Enter the initial position as 8 comma-separated integers (e.g., '0,1,2,3'):
0,1,2,3
The best position found is: [1 3 0 2]
The number of queens that are not attacking each other is: 4.0

Best State Diagram:
. . Q .
Q . . .
. . . Q
. Q . .
```

Program 6

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

Algorithm:



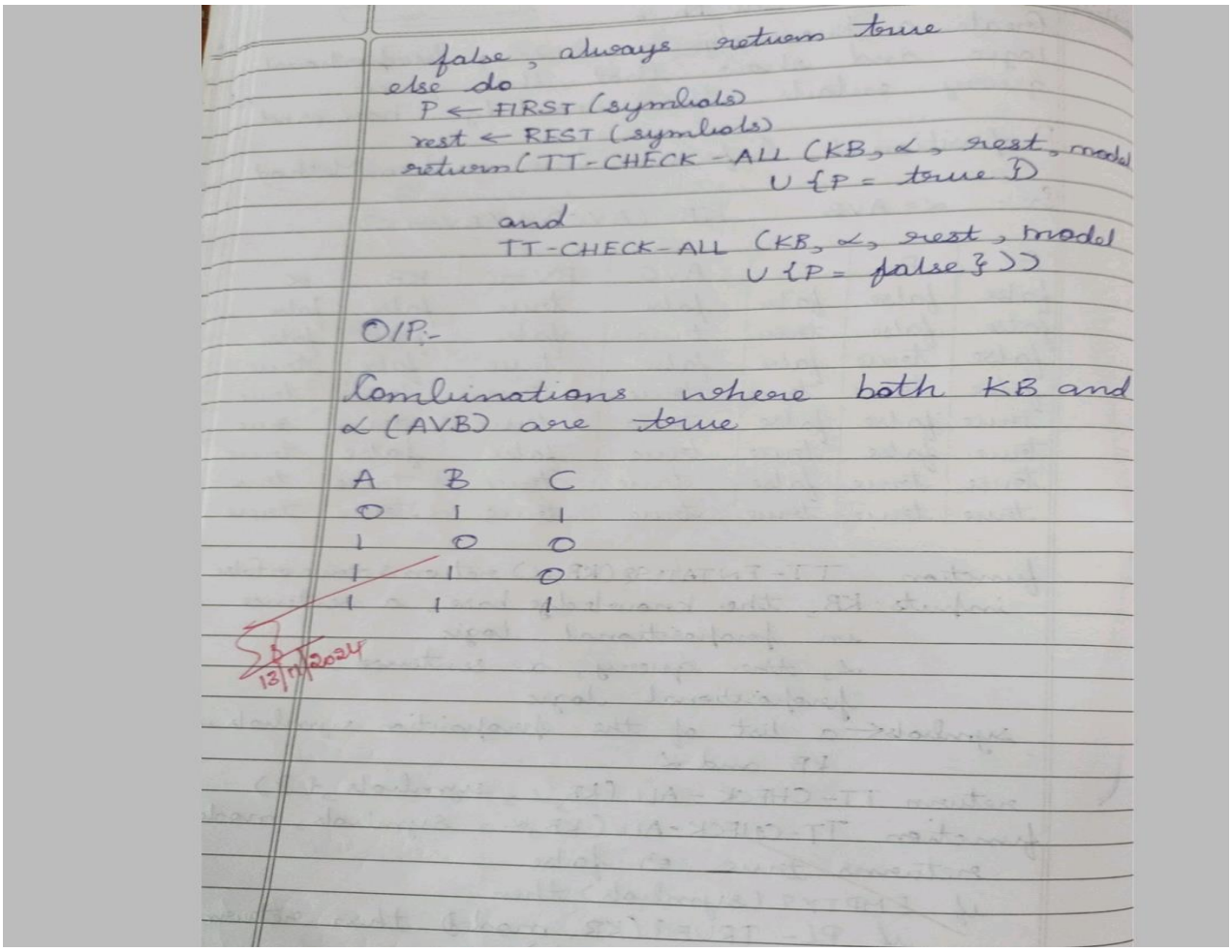
Create a knowledge base using proportional logic and show that the given query entails the knowledge base or not.

Propositional Inference: Enumeration Method

Ex:- $\alpha = A \vee B$ $KB = (A \vee C) \wedge (B \vee \neg C)$

A	B	C	AVC	BV¬C	KB	α
false	false	false	false	true	false	false
false	false	true	true	false	false	false
false	true	false	false	true	false	true
false	true	true	true	true	true	true
true	false	false	true	true	true	true
true	false	true	true	false	false	true
true	true	false	true	true	true	true
true	true	true	true	true	true	true

function TT-ENTAILS?(KB, α) returns true or false
inputs: KB, the knowledge base, a sentence in propositional logic
α, the query, a sentence in propositional logic
symbols ← a list of the proposition 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?(symbols) then
if PL-TRUE?(KB, model) then return PL-TRUE?(α, model)
else return true // when KB is



Code:

```
def pl_true(sentence, model):
    """Returns True if the sentence holds within the given model, False
otherwise."""
    if isinstance(sentence, bool):
        return sentence
    elif sentence in model:
        return model[sentence]
    elif isinstance(sentence, tuple):
        operator, *args = sentence
        if operator == 'NOT':
            return not pl_true(args[0], model)
        elif operator == 'AND':
            return all(pl_true(arg, model) for arg in args)
        elif operator == 'OR':
            return any(pl_true(arg, model) for arg in args)
        elif operator == 'IMPLIES':
            antecedent, consequent = args
            return not pl_true(antecedent, model) or pl_true(consequent,
model)
```

```
        elif operator == 'IFF':
            left, right = args
            return pl_true(left, model) == pl_true(right, model)
    return False  # if sentence can't be evaluated, default to False

def tt_entails(kb, alpha):
    """Checks if knowledge base kb entails alpha."""
    symbols = get_symbols(kb, alpha)
    return tt_check_all(kb, alpha, symbols, {})

def tt_check_all(kb, alpha, symbols, model):
    """Recursive check of all truth assignments of symbols in kb."""
    if not symbols:
        if pl_true(kb, model):
            return pl_true(alpha, model)
        else:
            return True  # If KB is false in model, we don't care about alpha
    else:
        p, rest = symbols[0], symbols[1:]
        model_true = model.copy()
        model_true[p] = True
        model_false = model.copy()
        model_false[p] = False
        return tt_check_all(kb, alpha, rest, model_true) and tt_check_all(kb,
alpha, rest, model_false)

def get_symbols(*sentences):
    """Returns a list of unique symbols in the given sentences."""
    symbols = set()
    for sentence in sentences:
        collect_symbols(sentence, symbols)
    return list(symbols)

def collect_symbols(sentence, symbols):
    """Recursively collects all symbols in a sentence."""
    if isinstance(sentence, str) and sentence.isalpha():
        symbols.add(sentence)
    elif isinstance(sentence, tuple):
        for arg in sentence[1:]:
            collect_symbols(arg, symbols)

# User input for knowledge base and query
print("Enter the knowledge base (KB) as a tuple expression (e.g., ('AND',
('IMPLIES', 'P', 'Q'), 'P')):")
kb = eval(input("KB: "))
```

```
print("Enter the query (alpha) as a tuple expression or a symbol (e.g.,
'Q'):")
alpha = eval(input("alpha: "))

# Check if KB entails alpha
result = tt_entails(kb, alpha)
print("Does KB entail alpha?", result)
```

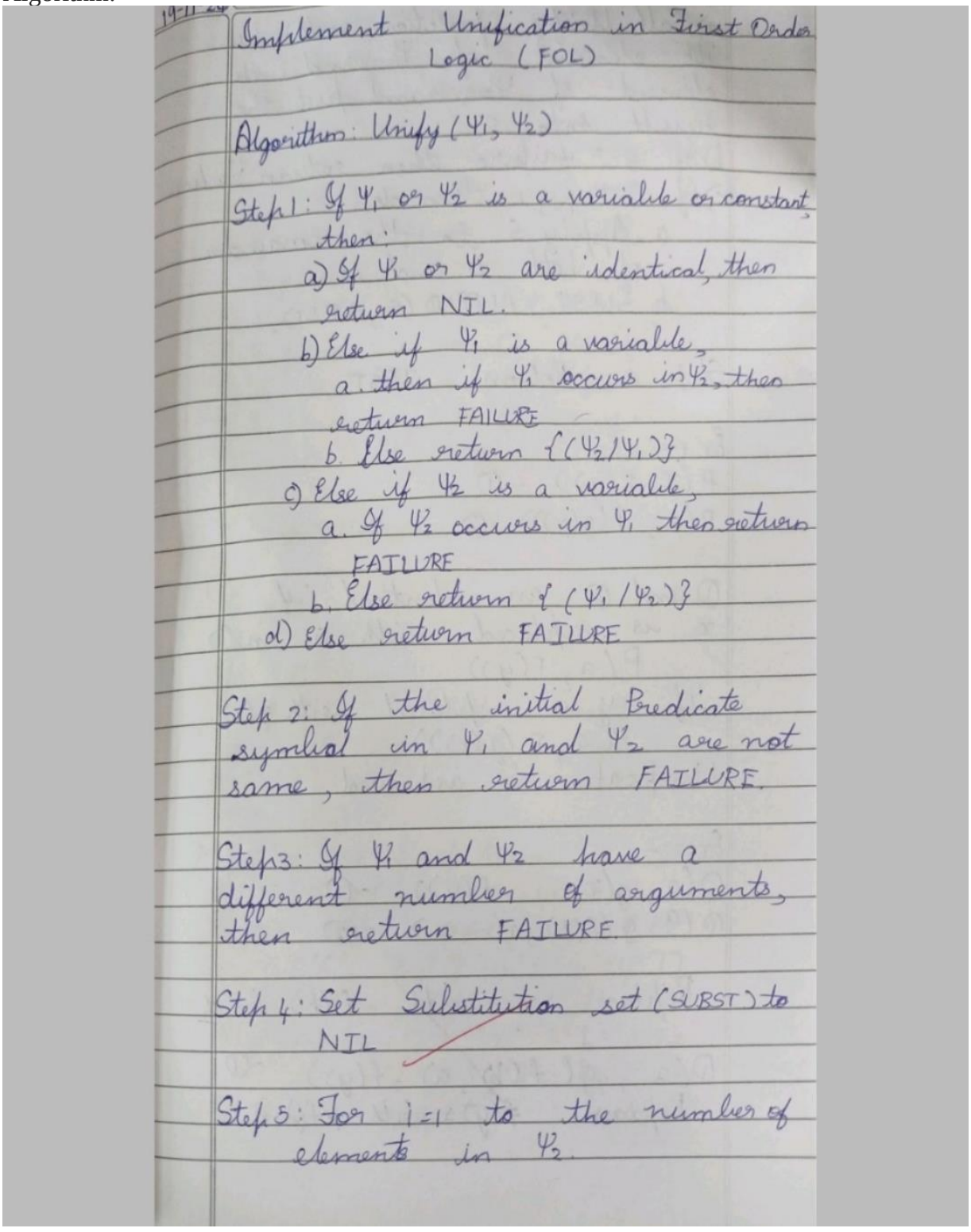
Output:

```
Enter the knowledge base (KB) as a tuple expression (e.g., ('AND',
('IMPLIES', 'P', 'Q'), 'P')):
KB: (('A','OR','C'),'AND',('B','OR','NOT','C'))
Enter the query (alpha) as a tuple expression or a symbol (e.g., 'Q'):
alpha: ('A','OR','B')
Does KB entail alpha? True
```

Program 7

Implement unification in first order logic

Algorithm:



Implement Unification in First Order Logic (FOL)

Algorithm: Unify (Ψ_1, Ψ_2)

Step 1: If Ψ_1 or Ψ_2 is a variable or constant, then:
a) If Ψ_1 or Ψ_2 are identical, then return NIL.
b) Else if Ψ_1 is a variable,
a. then if Ψ_1 occurs in Ψ_2, then return FAILURE
b. Else return {(Ψ_2/Ψ_1)}
c) Else if Ψ_2 is a variable,
a. If Ψ_2 occurs in Ψ_1 then return FAILURE
b. Else return {(Ψ_1/Ψ_2)}
d) Else return FAILURE

Step 2: If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE.

Step 3: If Ψ_1 and Ψ_2 have a different number of arguments, then return FAILURE.

Step 4: Set Substitution set (SUBST) to NIL

Step 5: For i=1 to the number of elements in Ψ_2.

a) Call Unify function with the ith element of Ψ_1 and ith element of Ψ_2, and put the result into S.
b) If S = failure then return failure
c) If S ≠ NIL then do,
   a. Apply S to the remainder of both L1 and L2
   b. SUBST = APPEND (S, SUBST).

Step 6 :- Return SUBST.

Ex 1:-
P(x, F(y)) - ①
P(a, F(g(x))) - ②

① and ② an identical if
x is replaced with a in ①
P(a, F(y))
if y is replaced with g(x)
P(a, F(g(x)))
Unification is achieved

Ex 2:-
Q(a, g(x, a), f(y)) - ①
Q(a, g(f(b), a), x) - ②

Replace x with f(b) in ①

Q(a, g(f(b), a), f(y)) - ①
~~Replace f(y) x with xf(y) in x~~

SURYA Gold

Date Page

~~Q(a, g(f(b), a), x)~~

Q(a, g(f(b), a), f(y)) - (2)

They are not unified

Ex 3:-

$\psi_1 = P(b, x, f(g(z)))$ - (1)

$\psi_2 = P(z, f(y), f(y))$ - (2)

Replace z in (2) as b

$\psi_2 = P(b, f(y), f(y))$

Replace x in (1) as f(y)

$\psi_1 = P(b, f(y), f(g(z)))$

Replace y in (2) as g(z)

$\psi_2 = P(b, f(y), f(g(z)))$

Unification is achieved.

Ex 4:-

$\psi_1 = P(f(a), g(y))$

$\psi_2 = P(x, x)$

Failure as x cannot be replaced by f(a), g(y)

O/P:-

i) Enter the first term: ['P', 'b', '?x', ['f', ['g', '?z']]]

Enter the second term: ['P', '?z', ['f', '?y'], ['f', '?y']]

Unification successful!

Substitution: {'?z': 'b', '?x': ['f', '?y'], '?y': ['g', '?z']}

Code:

```
class UnificationError(Exception):
    pass

def occurs_check(var, term, subst):
    if var == term:
        return True
    elif isinstance(term, (list, tuple)):
        return any(occurs_check(var, t, subst) for t in term)
    elif isinstance(term, str) and term in subst:
        return occurs_check(var, subst[term], subst)
    return False

def is_variable(term):
    return isinstance(term, str) and term.startswith('?')

def unify(psi1, psi2, subst=None):
    if subst is None:
        subst = {}

    if psi1 == psi2:
        return subst


    elif is_variable(psi1):
        if psi1 in subst:
            return unify(subst[psi1], psi2, subst)
        elif occurs_check(psi1, psi2, subst):
            raise UnificationError(f"Occurs check failed: {psi1} in {psi2}")
        else:
            subst[psi1] = psi2
            return subst

    elif is_variable(psi2):
        if psi2 in subst:
            return unify(psi1, subst[psi2], subst)
        elif occurs_check(psi2, psi1, subst):
            raise UnificationError(f"Occurs check failed: {psi2} in {psi1}")
        else:
            subst[psi2] = psi1
            return subst


    elif isinstance(psi1, list) and isinstance(psi2, list):
        if psi1[0] != psi2[0]:
```

```
            raise UnificationError(f"Predicate symbols don't match: {psi1[0]}
!= {psi2[0]}")


        if len(psi1) != len(psi2):
            raise UnificationError(f"Argument lengths don't match:
{len(psi1)} != {len(psi2)}")


        for arg1, arg2 in zip(psi1[1:], psi2[1:]):  # Skip the predicate
symbol (first element)
            subst = unify(arg1, arg2, subst)


        return subst


    else:
        raise UnificationError(f"Cannot unify {psi1} with {psi2}")


def get_input():
    try:
        term1 = eval(input("Enter the first term (e.g., ['P', 'b', 'x', ['f',
['g', 'z']]]): "))
        term2 = eval(input("Enter the second term (e.g., ['P', 'z', ['f',
'y'], ['f', 'y']]): "))
        substitution = unify(term1, term2)
        print("Unification successful!")
        print("Substitution:", substitution)
    except UnificationError as e:
        print("Unification failed:", e)
    except Exception as e:
        print("Invalid input or error:", e)

get_input()
```

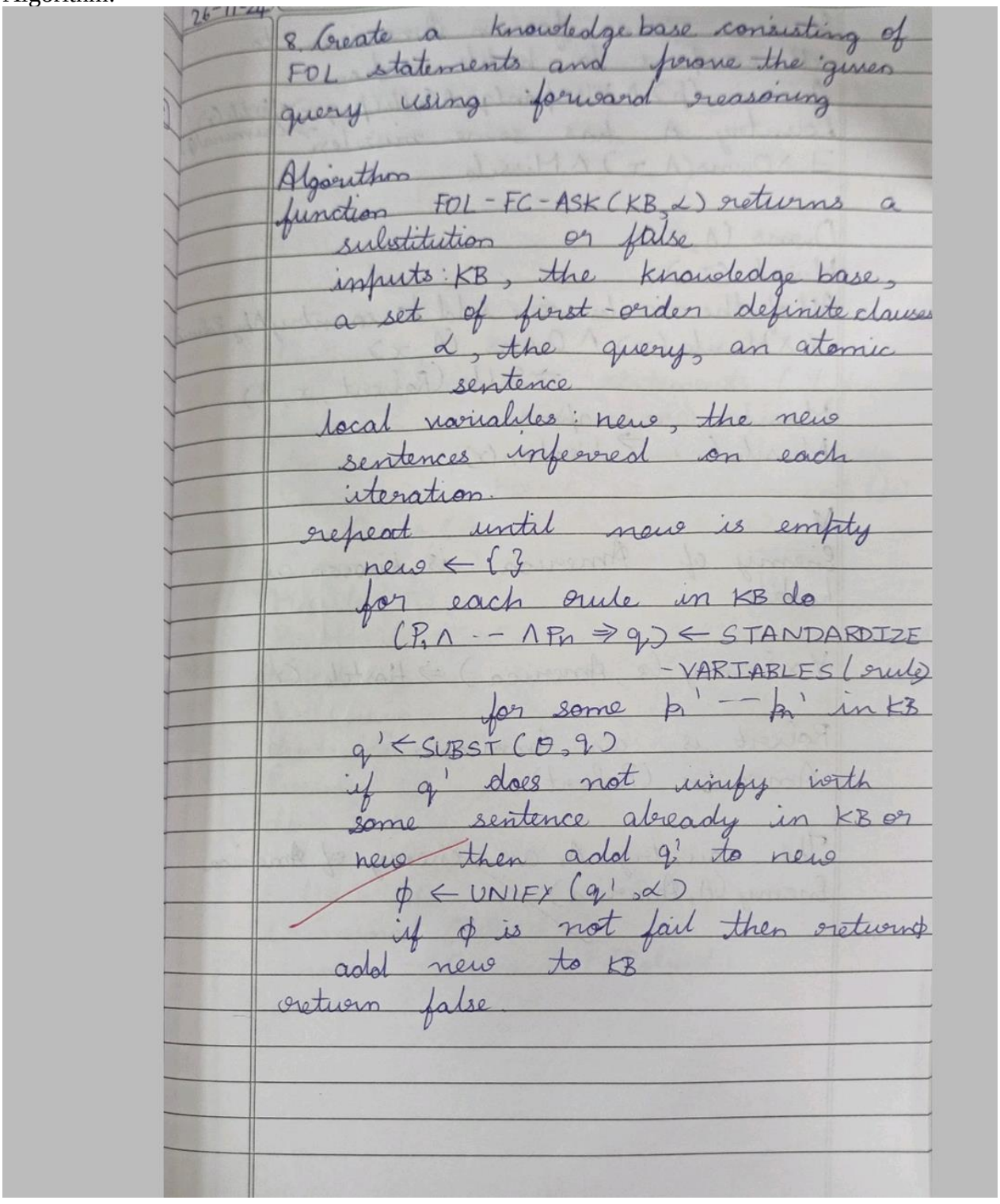
Output:

```
Enter the first term (e.g., ['P', 'b', 'x', ['f', ['g', 'z']]]):
['P','?x',['f','?y']]
Enter the second term (e.g., ['P', 'z', ['f', 'y'], ['f', 'y']]):
['P','a',['f',['g','?x']]]
Unification successful!
Substitution: {'?x': 'a', '?y': ['g', '?x']}
```

Program 8

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

Algorithm:



8. Create a knowledge base consisting of FOL statements and prove the given query using forward reasoning

Algorithm

function FOL-FC-ASK(KB, α) returns a substitution or false

inputs: KB, the knowledge base, a set of first-order definite clauses

α, the query, an atomic sentence

local variables: new, the new sentences inferred on each iteration.

repeat until new is empty

new ← { }

for each rule in KB do

$(p_1 \wedge \dots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES(rule)

for some $p_1' \dots p_n'$ in KB

$q' \leftarrow$ SUBST(θ, q)

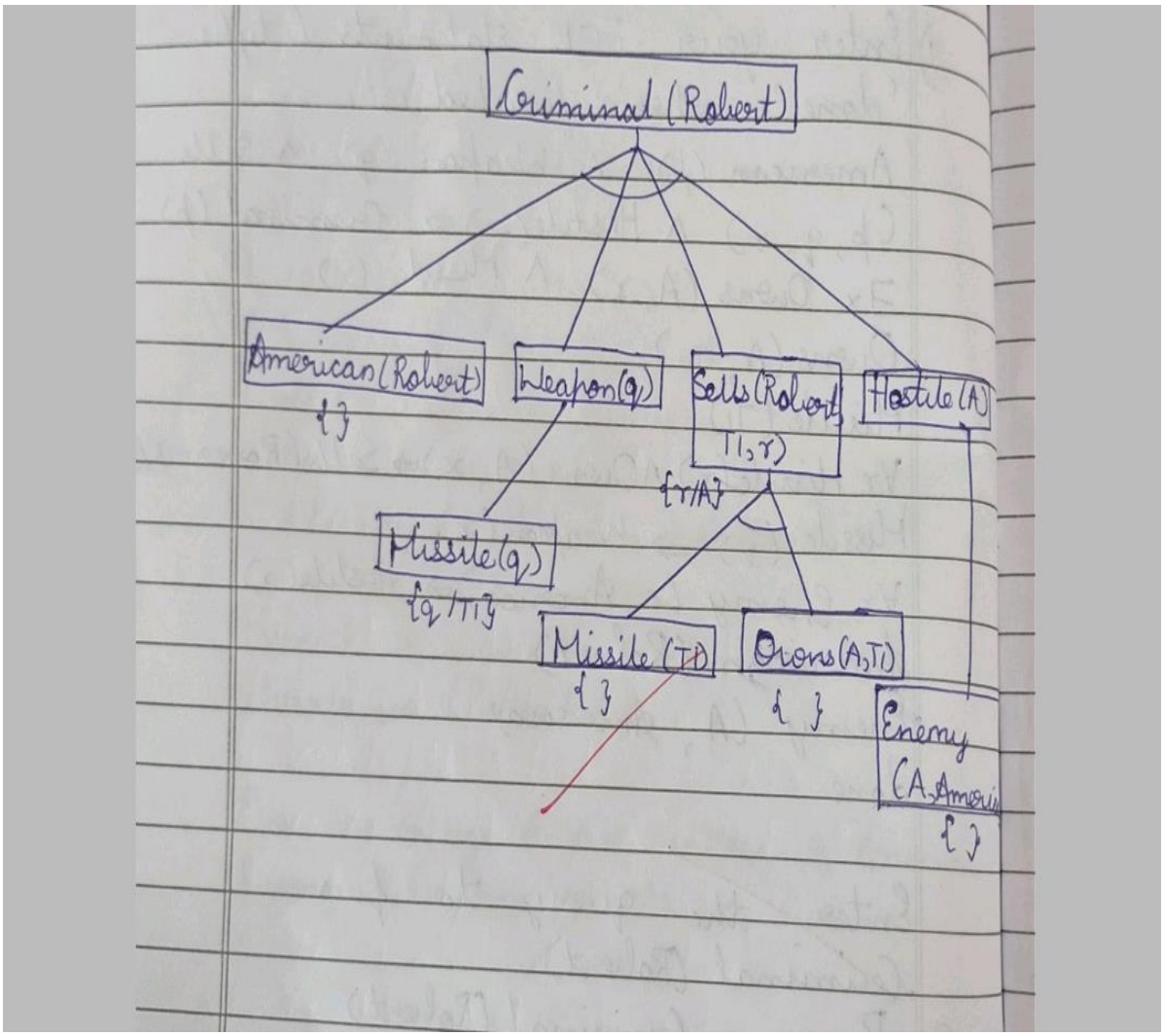
if q' does not unify with some sentence already in KB or new then add q' to new

$\phi \leftarrow$ UNIFY(q', α)

if ϕ is not fail then return ϕ

add new to KB

return false.



Code:

```
import re

class ForwardReasoning:
    def __init__(self, rules, facts):
        self.rules = rules  # List of rules (condition -> result)
        self.facts = set(facts)  # Known facts

    def match_condition(self, condition):
        """
        Check if the condition can be matched against the known facts.
        Handles universal and existential quantifiers, treating variables as
placeholders.
        """
        variable_map = {}  # Map to store variable-value pairs

        for cond in condition:
```

```
            if "∀" in cond:  # Universal quantifier handling
                var = cond[2:-1].strip()  # Extract variable from ∀x
                for fact in self.facts:
                    if var in fact:
                        variable_map[var] = fact
                        break
                else:
                    return False, variable_map
            elif "∃" in cond:  # Existential quantifier handling
                var = cond[2:-1].strip()  # Extract variable from ∃x
                for fact in self.facts:
                    if var in fact:
                        variable_map[var] = fact
                        return True, variable_map
                return False, variable_map
            else:  # Simple fact match
                fact_match = False
                for fact in self.facts:
                    if self.match_fact(cond, fact, variable_map):
                        fact_match = True
                        break
                if not fact_match:
                    return False, variable_map

        return True, variable_map

    def match_fact(self, cond, fact, variable_map):
        """
        Match a single fact to a condition by replacing variables in the
condition with the actual fact.
        """
        var_pattern = re.compile(r'\b[a-zA-Z]+\b')  # Pattern to detect
variable names

        # Check if the condition is a variable-based fact
        condition_parts = re.findall(var_pattern, cond)
        fact_parts = re.findall(var_pattern, fact)

        if len(condition_parts) == len(fact_parts):
            for var, fact_part in zip(condition_parts, fact_parts):
                if var not in variable_map:
                    variable_map[var] = fact_part
                elif variable_map[var] != fact_part:
                    return False
            return True
        return cond == fact  # If not variable-based, check exact match
```

```
    def infer(self, query):
        """
        Forward chaining algorithm to infer if the query can be derived from
rules and facts.
        """
        applied_rules = True

        while applied_rules:
            applied_rules = False
            for condition, result in self.rules:
                matched, variable_map = self.match_condition(condition)
                if matched and result not in self.facts:
                    self.facts.add(result)  # Add the result to known facts
                    applied_rules = True
                    print(f"Applied rule: {condition} -> {result}")

                    # If the query is inferred, return True immediately
                    if self.match_fact(query, result, variable_map):
                        return True

        # Return True if the query is in facts after the reasoning process,
else False
        return self.match_fact(query, result, variable_map)

def get_input_rules():
    rules = []
    while True:
        rule = input("Enter rule (or 'done' to finish): ").strip()
        if rule.lower() == "done":
            break

        # Parse the rule properly
        if "=>" in rule:
            # Check for complex expressions with quantifiers and split the
rule
            condition_str, result = rule.split("=>")

            # Remove extra spaces and deal with complex conditions
            condition_str = condition_str.strip()
            result = result.strip()

            # Handle potential multiple conditions (ANDs)
            conditions = set(re.split(r'\s*AND\s*', condition_str))

            # Add the rule to the list
```

```
            rules.append((conditions, result))

    return rules

def get_input_facts():
    facts = set()
    while True:
        fact = input("Enter fact (or 'done' to finish): ").strip()
        if fact.lower() == "done":
            break
        facts.add(fact)
    return facts

def get_input_query():
    query = input("Enter the query: ").strip()
    return query


# Main program to run the forward reasoning
def main():
    print("Enter the rules:")
    rules = get_input_rules()

    print("\nEnter the facts:")
    facts = get_input_facts()

    print("\nEnter the query:")
    query = get_input_query()

    # Initialize and run forward reasoning
    reasoner = ForwardReasoning(rules, facts)
    result = reasoner.infer(query)

    # Debugging Output
    print("\nFinal facts:")
    print(reasoner.facts)
    print(f"\nQuery '{query}' inferred: {result}")

# Call the main function to start
main()
```

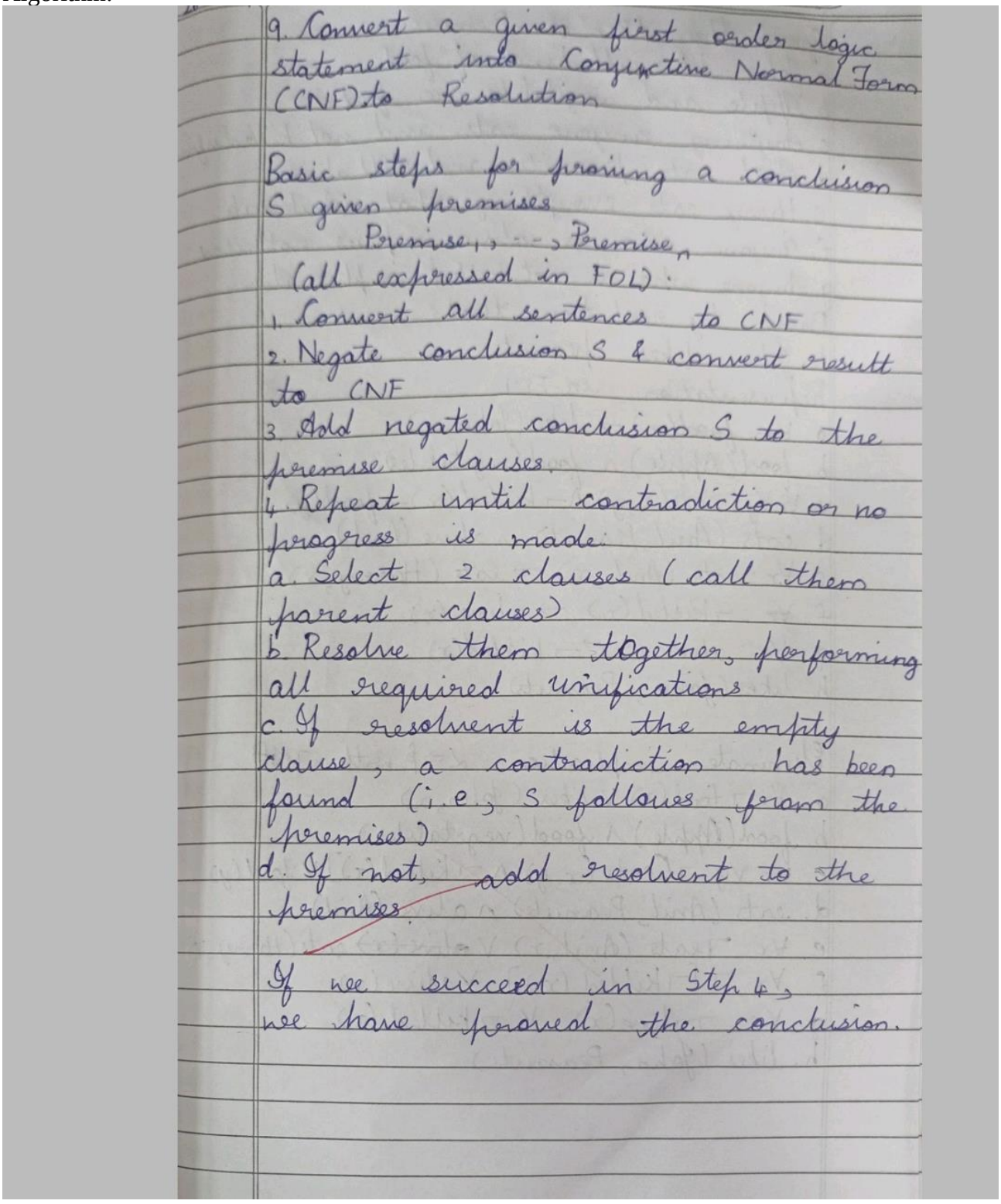
Output:

```
Enter the rules:
Enter rule (or 'done' to finish): American(p) AND Weapon(q) AND Sells(p, q,
r) AND Hostile(r) => Criminal(p)
Enter rule (or 'done' to finish): ∃x (Owns(A, x) AND Missile(x)) =>
Missile(x) AND Weapon(x)
Enter rule (or 'done' to finish): ∀x(Missile(x) AND Owns(A, x)) =>
Sells(Robert, x, A)
Enter rule (or 'done' to finish): Missile(x) => Weapon(x)
Enter rule (or 'done' to finish): ∀x (Enemy(x, America)) => Hostile(x)
Enter rule (or 'done' to finish): done

Enter the facts:
Enter fact (or 'done' to finish): American(Robert)
Enter fact (or 'done' to finish): Enemy(A, America)
Enter fact (or 'done' to finish): Owns(A, T1)
Enter fact (or 'done' to finish): Missile(T1)
Enter fact (or 'done' to finish): done

Enter the query:
Enter the query: Criminal(Robert)
Applied rule: {'Missile(x)'} -> Weapon(x)

Final facts:
{'American(Robert)', 'Owns(A, T1)', 'Missile(T1)', 'Enemy(A, America)',
'Weapon(x)'}

Query 'Criminal(Robert)' inferred: True
```

Program 9

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

Algorithm:



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

Basic steps for proving a conclusion S given premises

Premise$_1$, ..., Premise$_n$

(all expressed in FOL):

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.

Given the KB or Premises:
a. John likes all kind of food.
b. Apple and vegetables are food.
c. Anything anyone eats and not killed is food.
d. Anil eats peanuts and still alive
e. Harry eats everything that Anil eats.
f. Anyone who is alive implies not killed
g. Anyone who is not killed implies alive
Prove by resolution that:
h. John likes peanuts.

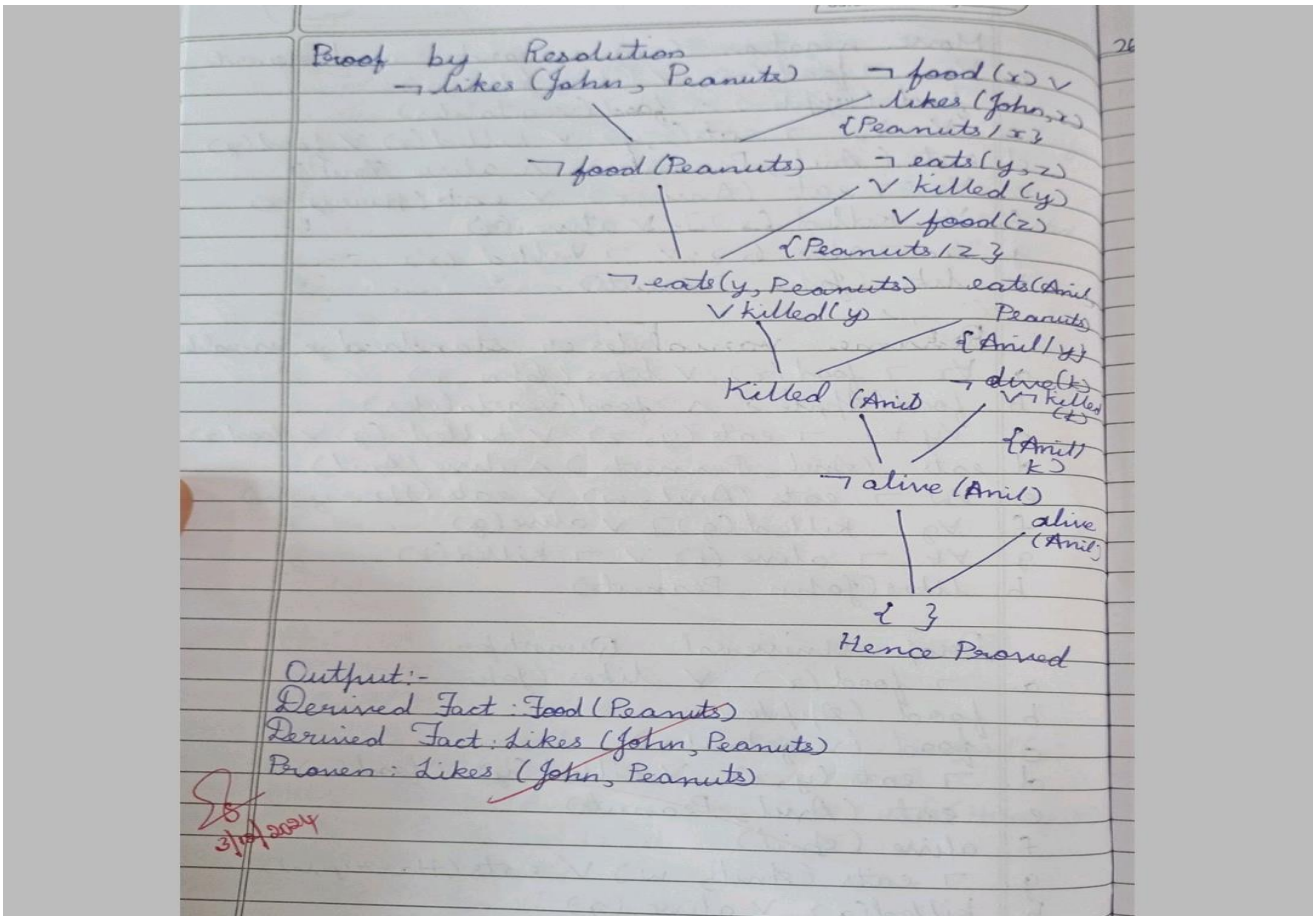
Representation in FOL
a. $\forall x: food(x) \rightarrow likes(John, x)$
b. $food(Apple) \wedge food(vegetables)$
c. $\forall x \forall y: eats(x,y) \wedge \neg killed(x) \rightarrow food(y)$
d. $eats(Anil, Peanuts) \wedge alive(Anil)$
e. $\forall x: eats(Anil, x) \rightarrow eats(Harry, x)$
f. $\forall x: \neg killed(x) \rightarrow alive(x)$
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. $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$
f. $\forall x \neg[\neg killed(x)] \vee alive(x)$
g. $\forall x \neg alive(x) \vee \neg killed(x)$
h. $likes(John, Peanuts)$

Move negation ($\neg$) inwards and rewrite
a. $\forall x \neg food(x) \vee likes(John, x)$
b. $food(Apple) \wedge food(vegetables)$
c. $\forall x \forall y \neg eats(x,y) \vee killed(x) \vee food(y)$
d. $eats(Anil, Peanuts) \wedge alive(Anil)$
e. $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$
f. $\forall x killed(x)] \vee alive(x)$
g. $\forall x \neg alive(x) \vee \neg killed(x)$
h. $likes(John, Peanuts)$.

Rename variables or standardize vari
a. $\forall x \neg food(x) \vee likes(John, x)$
b. $food(Apple) \wedge food(vegetables)$
c. $\forall y \forall z \neg eats(y, z) \vee killed(y) \vee food$
d. $eats(Anil, Peanuts) \wedge alive(Anil)$
e. $\forall w \neg eats(Anil, w) \vee eats(Harry, w)$
f. $\forall g killed(g)] \vee alive(g)$
g. $\forall k \neg alive(k) \vee \neg killed(k)$
h. $likes(John, Peanuts)$

Drop Universal Quantifier
a. $\neg food(x) \vee likes(John, x)$
b. $food(Apple)$
c. $food(vegetables)$
d. $\neg eats(y, z) \vee killed(y) \vee food(z)$
e. $eats(Anil, Peanuts)$
f. $alive(Anil)$
g. $\neg eats(Anil, w) \vee eats(Harry, w)$
h. $killed(g) \vee alive(g)$
i. $\neg alive(k) \vee \neg killed(k)$
j. $likes(John, Peanuts)$.



Code:

```
# Knowledge Base (KB)
facts = {
    "Eats(Anil, Peanuts)": True,
    "not Killed(Anil)": True,
    "Food(Apple)": True,
    "Food(Vegetables)": True,
}

rules = [
    # Rule: Food(X) :- Eats(Y, X) and not Killed(Y)
    {"conditions": ["Eats(Y, X)", "not Killed(Y)"], "conclusion": "Food(X)"},
    # Rule: Likes(John, X) :- Food(X)
    {"conditions": ["Food(X)"], "conclusion": "Likes(John, X)"},
]

# Query
query = "Likes(John, Peanuts)"

# Helper function to substitute variables in a rule
def substitute(rule_part, substitutions):
    for var, value in substitutions.items():
```

```
        rule_part = rule_part.replace(var, value)
    return rule_part

# Function to resolve the query
def resolve_query(facts, rules, query):
    working_facts = facts.copy()
    while True:
        new_facts_added = False
        for rule in rules:
            conditions = rule["conditions"]
            conclusion = rule["conclusion"]

            # Try all substitutions for variables (X, Y) in the rules
            for entity in ["Apple", "Vegetables", "Peanuts", "Anil", "John"]:
                substitutions = {"X": "Peanuts", "Y": "Anil"}  # Fixed for
this problem
                resolved_conditions = [substitute(cond, substitutions) for
cond in conditions]
                resolved_conclusion = substitute(conclusion, substitutions)

                # Check if all conditions are true
                if all(working_facts.get(cond, False) for cond in
resolved_conditions):
                    if resolved_conclusion not in working_facts:
                        working_facts[resolved_conclusion] = True
                        new_facts_added = True
                        print(f"Derived Fact: {resolved_conclusion}")

        if not new_facts_added:
            break

    # Check if the query is resolved
    return working_facts.get(query, False)

# Run the resolution process
if resolve_query(facts, rules, query):
    print(f"Proven: {query}")
else:
    print(f"Not Proven: {query}")
```

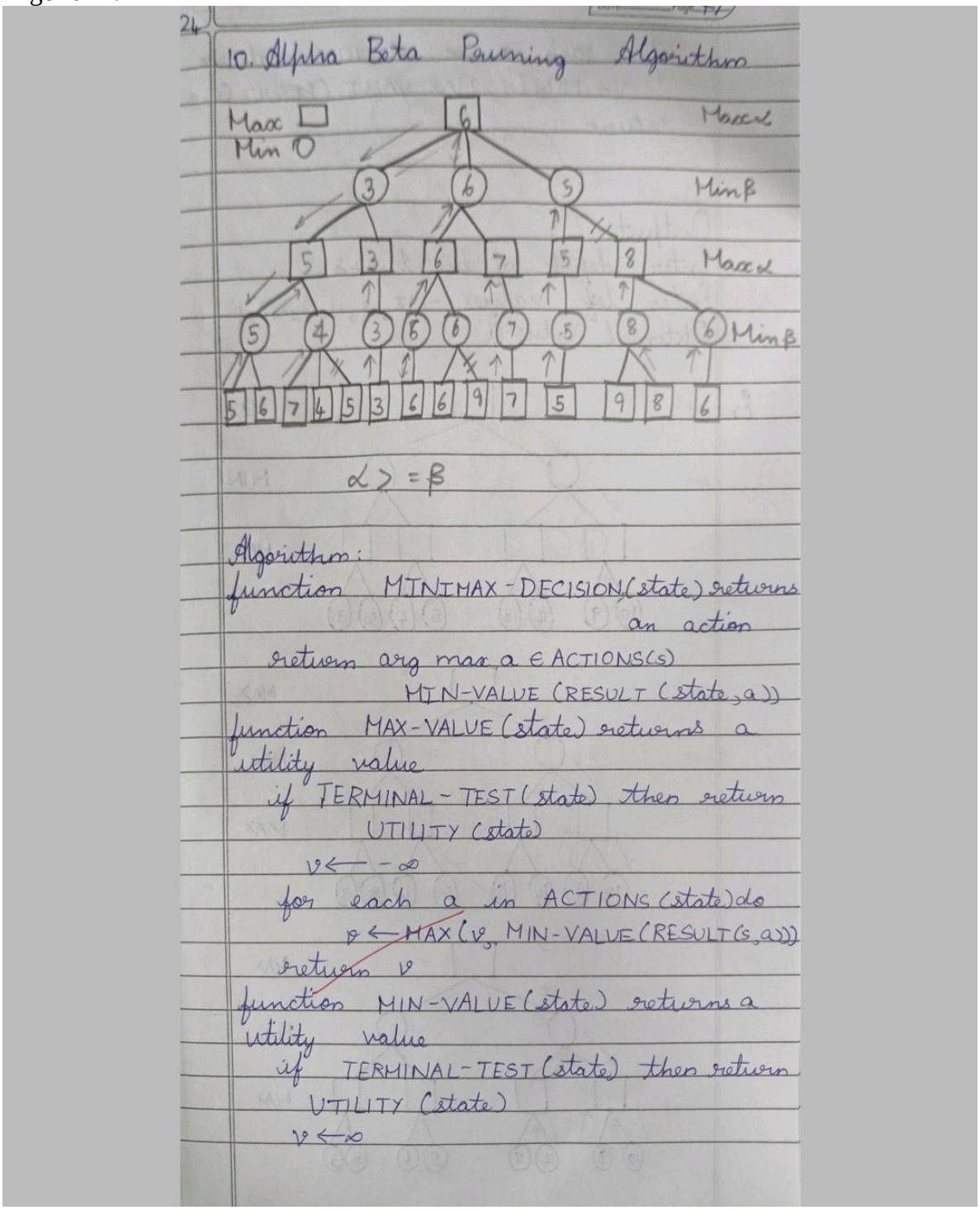
Output:

```
Derived Fact: Food(Peanuts)
Derived Fact: Likes(John, Peanuts)
Proven: Likes(John, Peanuts)
```

Program 10

Implement Alpha-Beta Pruning

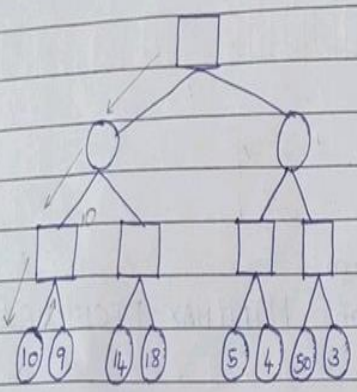
Algorithm:

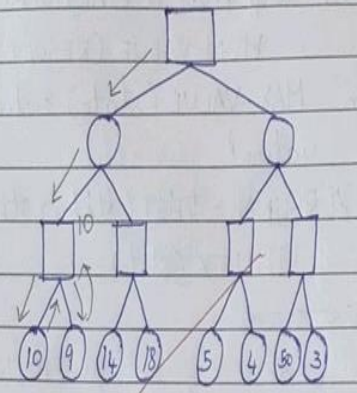


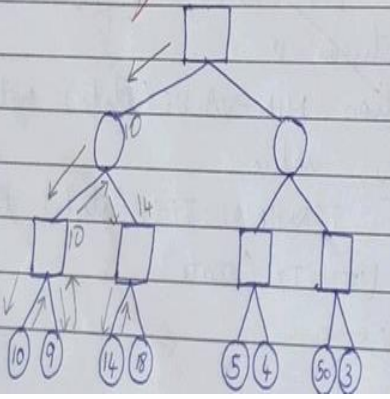
for each a in ACTIONS(state) do
v ← MIN(v, MAX-VALUE(RESULT(s,a)))
return v

Output:-
Enter depth of the tree: 3
Enter leaf values: -1, 8, -3, -1, 2, 1, 3, 4
Optimal Value: 8

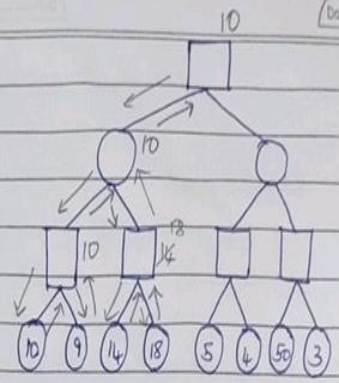
Ex:

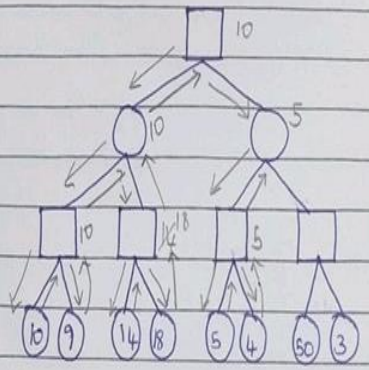


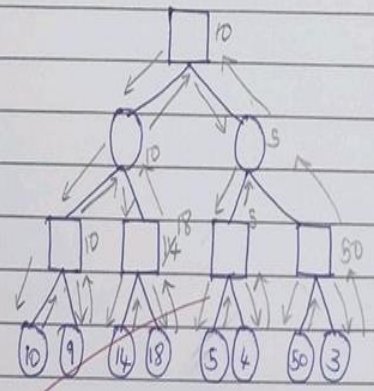




SURYA Gold
Date Page







8
3/12/2024

Code:

```
class MinMaxPruning:
    def __init__(self):
        self.tree = []
        self.pruned_nodes = []

    def take_input(self):
        """
        Takes user input for the tree structure and its values.
        Example Input:
        Enter depth of the tree: 3
        Enter leaf values at depth 3 (comma-separated): 3, 5, 6, 9, 1, 2, 0,
-1
        """
        self.depth = int(input("Enter depth of the tree: "))
        print(f"Tree is assumed to be a full binary tree of depth
{self.depth}.")

        # Input leaf node values
        leaf_values = list(map(int, input(f"Enter leaf values at depth
{self.depth} (comma-separated): ").split(",")))
        self.tree = leaf_values
        print("\nInput Tree (Leaf Nodes):", self.tree)

    def min_max(self, depth, node_index, maximizing_player, alpha, beta):
        """
        Performs Min-Max pruning and calculates the optimal value.
        :param depth: Current depth in the tree
        :param node_index: Index of the current node
        :param maximizing_player: Boolean indicating whether it's Max's turn
        :param alpha: Alpha value for pruning
        :param beta: Beta value for pruning
        :return: Optimal value for the current subtree
        """
        if depth == self.depth:
            return self.tree[node_index]

        if maximizing_player:
            best = float("-inf")
            for i in range(2):
                value = self.min_max(depth + 1, 2 * node_index + i, False,
alpha, beta)
                best = max(best, value)
                alpha = max(alpha, best)
                if beta <= alpha:
                    self.pruned_nodes.append((depth, node_index, "MAX"))
```

```
                    break  # Beta cutoff
            return best
        else:
            best = float("inf")
            for i in range(2):
                value = self.min_max(depth + 1, 2 * node_index + i, True,
alpha, beta)
                best = min(best, value)
                beta = min(beta, best)
                if beta <= alpha:
                    self.pruned_nodes.append((depth, node_index, "MIN"))
                    break  # Alpha cutoff
            return best

    def display_tree(self):
        """
        Display the pruned tree structure and the pruned nodes.
        """
        print("\nPruned Nodes (Depth, Node Index, Type):", self.pruned_nodes)
        print("Pruned nodes represent subtrees that were skipped due to
pruning.")

    def run(self):
        """
        Driver function to run the Min-Max pruning.
        """
        self.take_input()
        print("\nRunning Min-Max Pruning...")
        optimal_value = self.min_max(0, 0, True, float("-inf"), float("inf"))
        print("\nOptimal Value (Root Node):", optimal_value)
        self.display_tree()


if __name__ == "__main__":
    pruning = MinMaxPruning()
    pruning.run()
```

Output:

```
Enter depth of the tree: 3
Tree is assumed to be a full binary tree of depth 3.
Enter leaf values at depth 3 (comma-separated): -1,8,-3,-1,2,1,3,4

Input Tree (Leaf Nodes): [-1, 8, -3, -1, 2, 1, 3, 4]

Running Min-Max Pruning...

Optimal Value (Root Node): 2
```