VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

SUJNYAN ULHAS KINI(1BM22CS340)

in partial fulfilment 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 SUJNYAN ULHAS KINI **(1BM22CS340),** 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 Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

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

Index

Sl. No.	Date	Experiment Title	Page No.
1	24-9-2024	Implement Tic –Tac –Toe Game	1-7
2	1-10-2024	Implement vacuum cleaner agent	8-13
3	8-10-2024	Implement 8 puzzle problems using Depth First Search (DFS)	14-20
4	15-10-2024	Implement A* search algorithm Implement Iterative deepening search algorithm	21-31
5	22-10-2024	Simulated Annealing to Solve 8-Queens problem	32-35
6	29-10-2024	Implement A* search algorithm for N queens Implement Hill Climbing search algorithm to solve N-Queens problem	36-40
7	12-11-2024	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	41-43
8	19-11-2024	Implement unification in first order logic	44-46
9	3-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	47-50
10	3-12-2024	Implement Min-Max Algorithm for Tic Tac Toe Implement Alpha-Beta Pruning for 8 queens	51-59

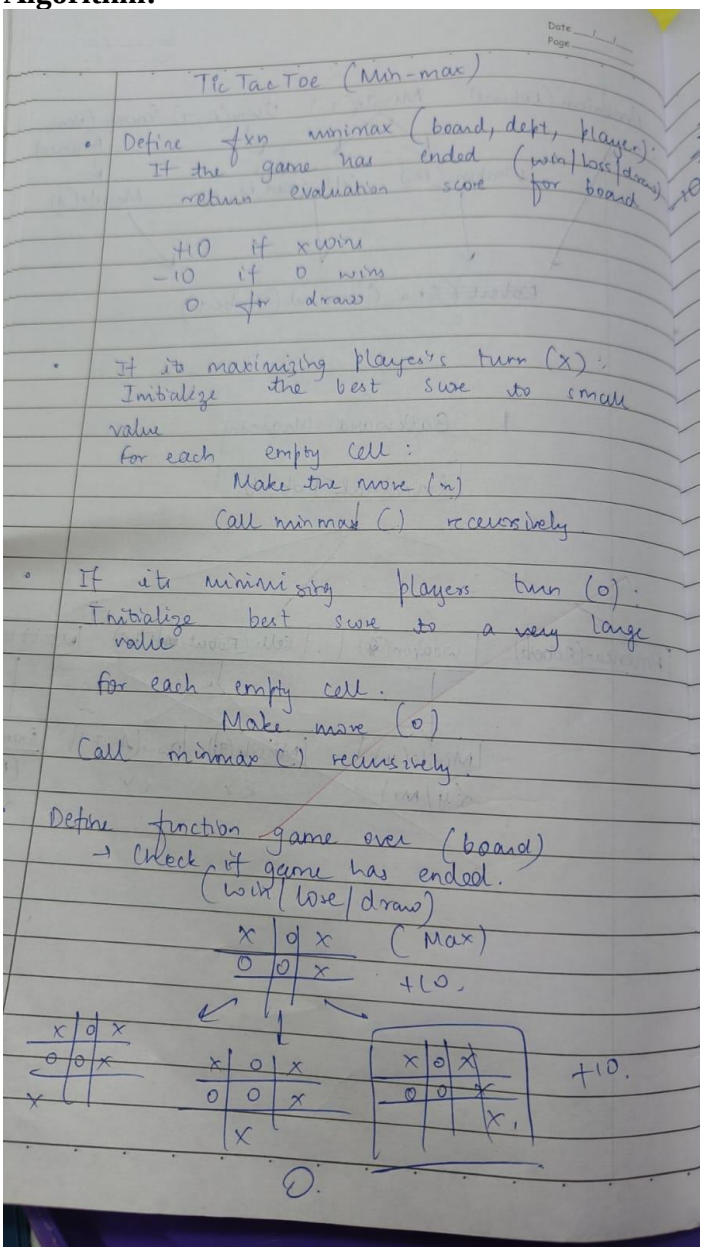
Github Link:

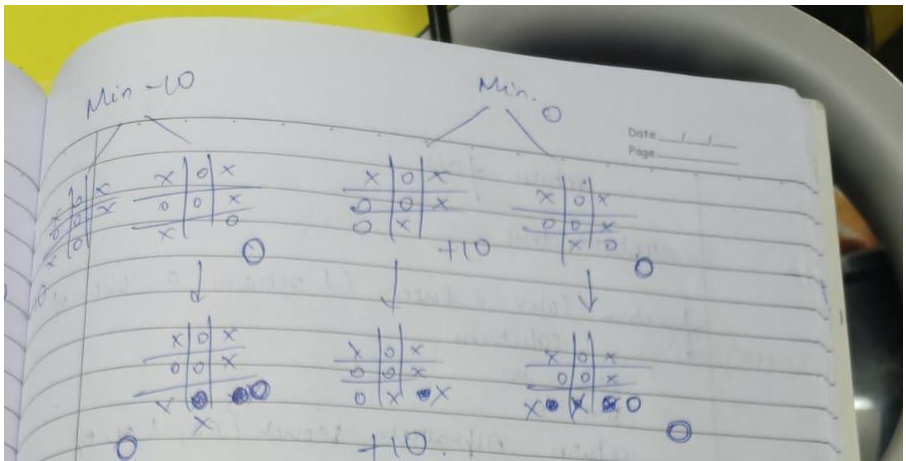
https://github.com/sujkini/AI_Python

Program 1

Implement Tic –Tac –Toe Game

Algorithm:

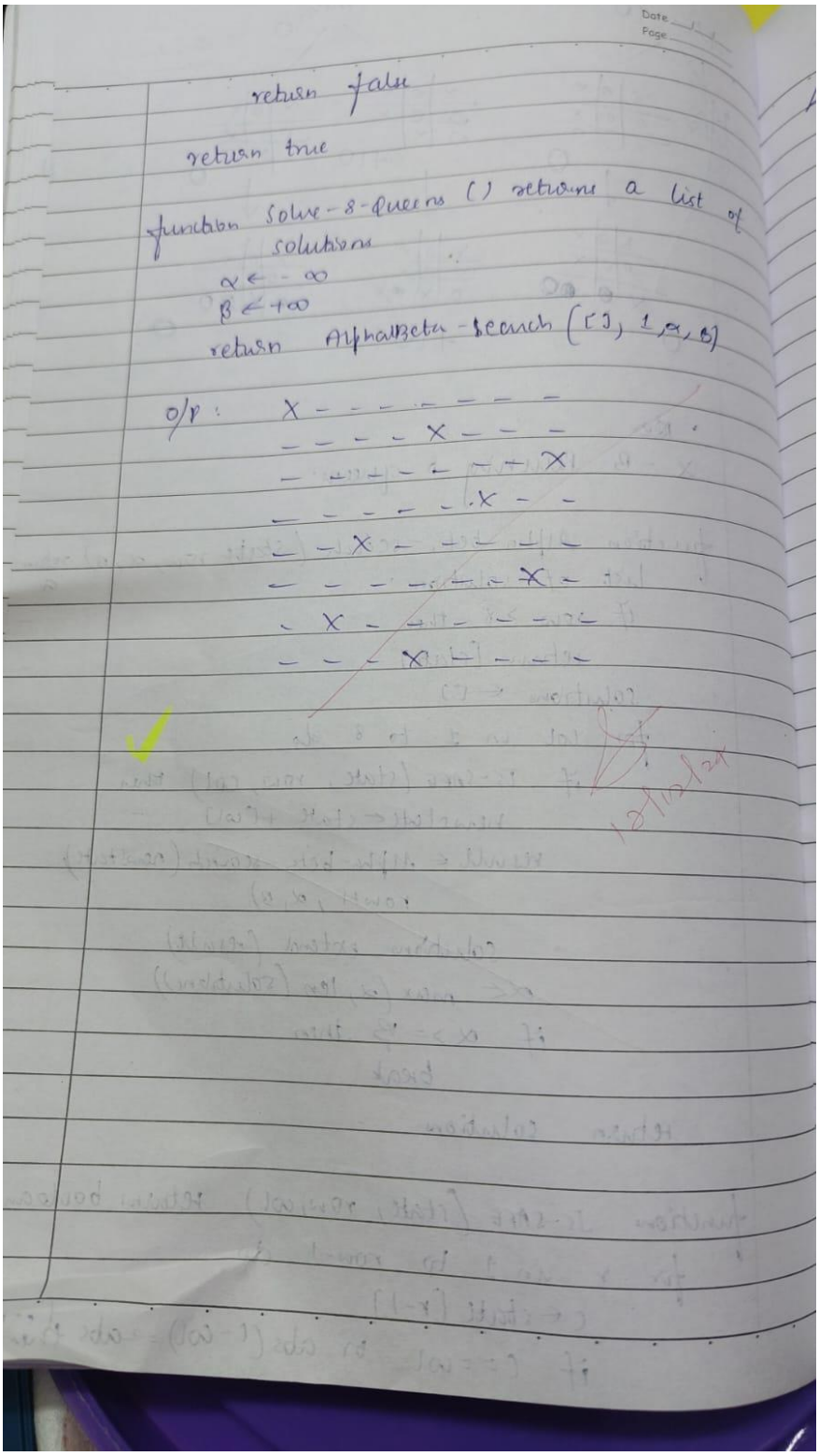




∴ Ex

α - β Pruning 8 queens:

```
function Alpha-beta-search (state, row, α, β) return a
    list of solutions.
    if row > 8 then
        return [state]
    solutions ← []
    for col in 1 to 8 do
        if IS-SAFE (state, row, col) then
            newstate ← state + [col]
            result ← Alpha-beta-search (newstate)
                row+1, α, β)
            solutions.extend (result)
            α ← max (α, len (solutions))
            if α >= β then
                break
    return solutions

function IS-SAFE (state, row, col) returns boolean
    for r in 1 to row-1 do
        c ← state [r-1]
        if c == col or abs (c - col) == abs (r - row) then
```



return false

return true

function Solve-8-Queens () returns a list of solutions

$\alpha \leftarrow -\infty$

$\beta \leftarrow +\infty$

return Alphabeta-Search ([], 1, α, β)

o/p :

X - - - - - - -

- - - - X - - -

- - - - - - - X

- - - - - X - -

- - X - - - - -

- - - - - - X -

- X - - - - - -

- - - X - - - -

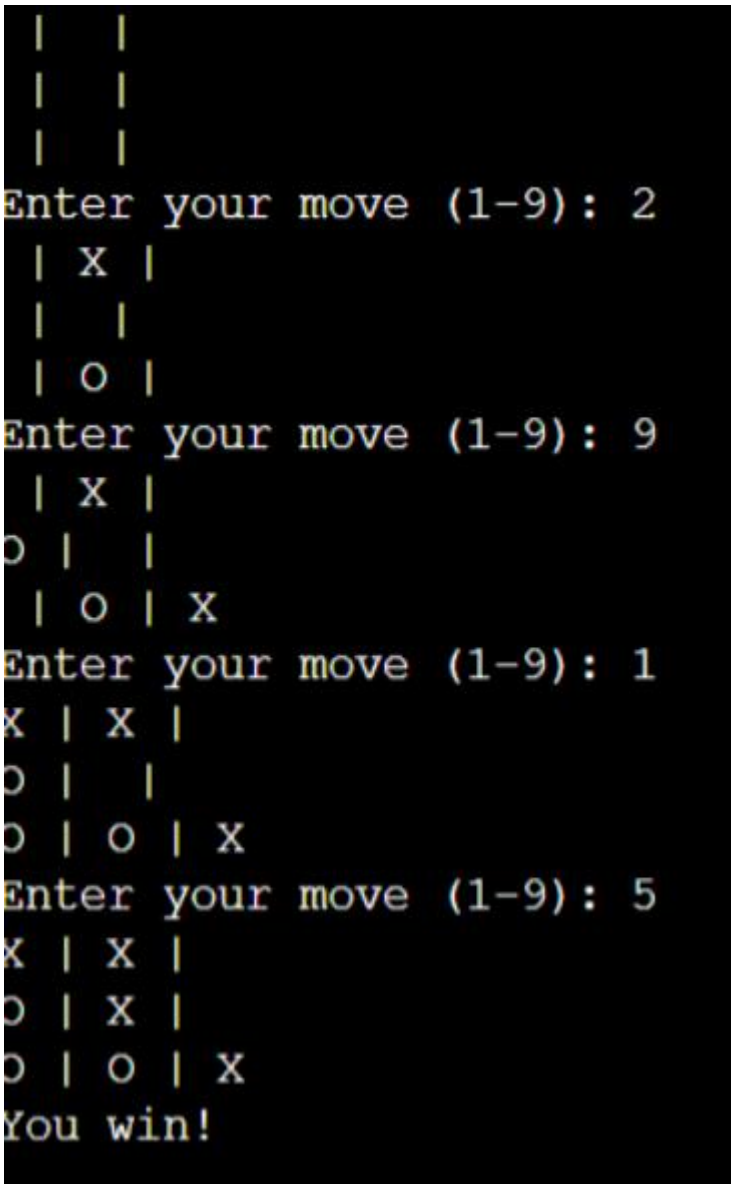
Code:

```
import random

def win(board):
    for row in board:
        if row[0] == row[1] == row[2] != "":
            return True
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] != "":
            return True
    if board[0][0] == board[1][1] == board[2][2] != "":
        return True
    if board[0][2] == board[1][1] == board[2][0] != "":
        return True
    return False

def printBoard(board):
    print("\n".join([" | ".join(row) for row in board]))

def draw(board):
    return all(cell != "" for row in board for cell in row)

def user_move(board):
    while True:
        try:
            move = int(input("Enter your move (1-9): ")) - 1
            row, col = divmod(move, 3)
            if board[row][col] == "":
                board[row][col] = "X"
                break
            else:
```

```
            print("That space is already taken. Try again.")
        except (ValueError, IndexError):
            print("Invalid input. Please enter a number from 1 to 9.")

def computer_move(board):
    while True:
        move = random.randint(0, 8)
        row, col = divmod(move, 3)
        if board[row][col] == "":
            board[row][col] = "O"
            break

def _main():
    board = [["" for _ in range(3)] for _ in range(3)]

    while True:
        printBoard(board)
        user_move(board)
        if win(board):
            printBoard(board)
            print("You win!")
            break
        if draw(board):
            printBoard(board)
            print("It's a draw!")
            break
        computer_move(board)
        if win(board):
            printBoard(board)
            print("Computer wins!")
            break
        if draw(board):
```

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

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

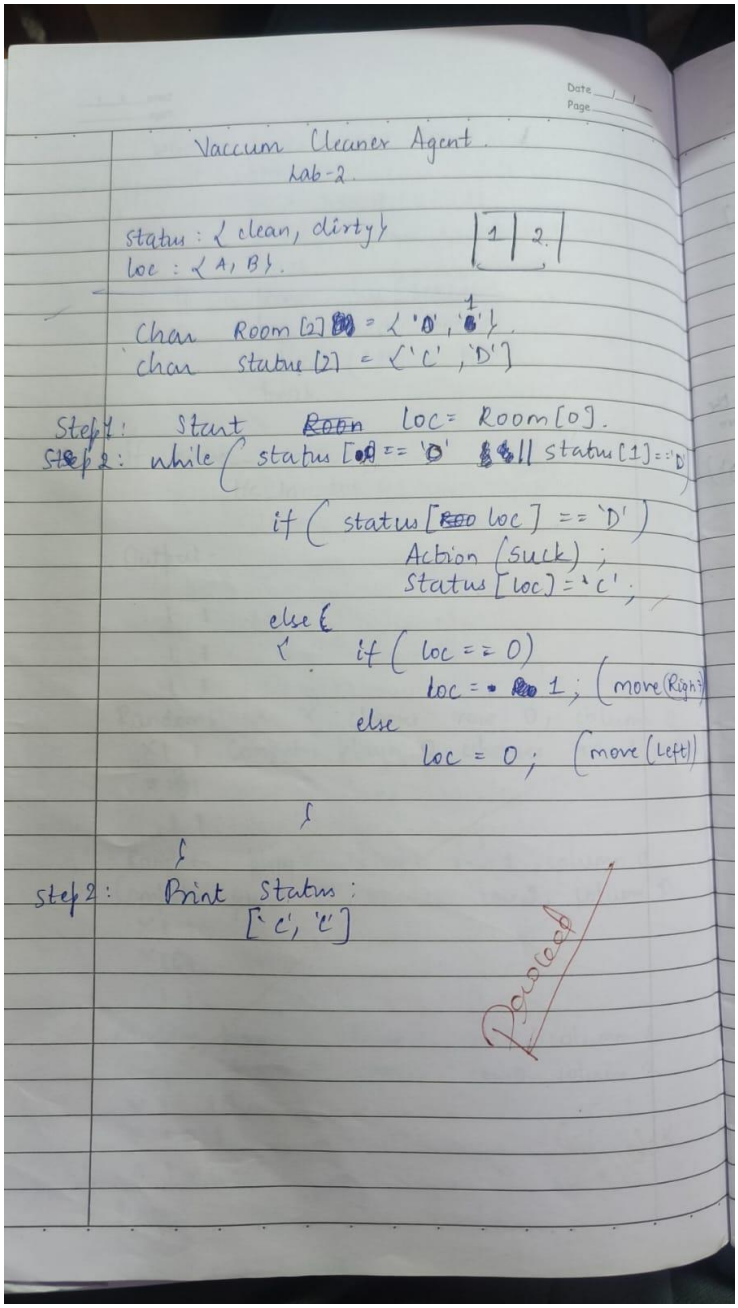
Output:



Program 2

Implement vacuum cleaner agent

Algorithm:



Code:

```
def printArr(arr):
    for row in arr:
        print(row)
    print()
def clean(arr, x, y):
    if arr[x][y] == 1:
        arr[x][y] = 0
def check(arr):
    for row in arr:
        if 1 in row:
            return True
    return False

# Directions: right (0,1), down (1,0), left (0,-1), up (-1,0)
directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
direction_index = 0  # Start moving right

# Get room status
print("Enter the status of the rooms (0 for clean; 1 for dirty):")
arr1 = []
for i in range(2):
    row = []
    for j in range(2):
        a = int(input(f"Status of room ({i}, {j}): "))
        row.append(a)
    arr1.append(row)
x, y = 0, 0  #Start cleaning from the first room
while True:
    printArr(arr1)
    if not check(arr1):
```

```
        break
    clean(arr1, x, y)
    #Move to the next room in the current direction
    dx, dy = directions[direction_index]
    new_x, new_y = x + dx, y + dy

    #Check bounds
    if 0 <= new_x < 2 and 0 <= new_y < 2:
        x, y = new_x, new_y
    else:
        #Change direction(turn right)
        direction_index = (direction_index + 1) % 4
        dx, dy = directions[direction_index]
        x, y = x + dx, y + dy  #Move in the new direction
print("All rooms are cleaned!")
```

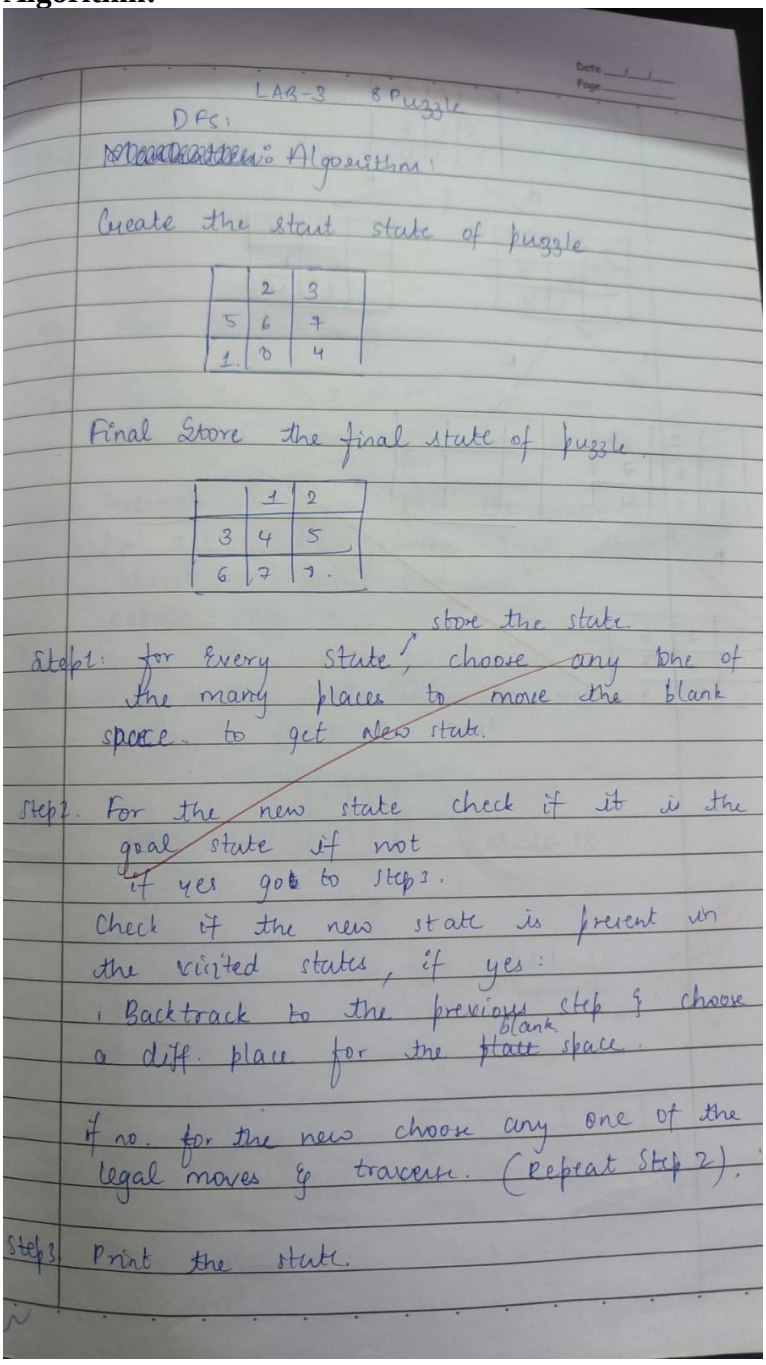
Output:

```
Enter the status of the rooms (0 for clean; 1 for dirty):
Status of room (0, 0): 1
Status of room (0, 1): 0
Status of room (1, 0): 1
Status of room (1, 1): 0
[1, 0]
[1, 0]

[0, 0]
[1, 0]

[0, 0]
[1, 0]

[0, 0]
[1, 0]

[0, 0]
[0, 0]

All rooms are cleaned!
```

Program 3

Implement 8 puzzle problems using Depth First Search (DFS)

Algorithm:



LAB-3 8 Puzzle

DFS:

Algorithm:

Create the start state of puzzle

	2	3
5	6	7
1	0	4

Final Store the final state of puzzle.

	1	2
3	4	5
6	7	8

Step1: for Every state, store the state. choose any one of the many places to move the blank space. to get new state.

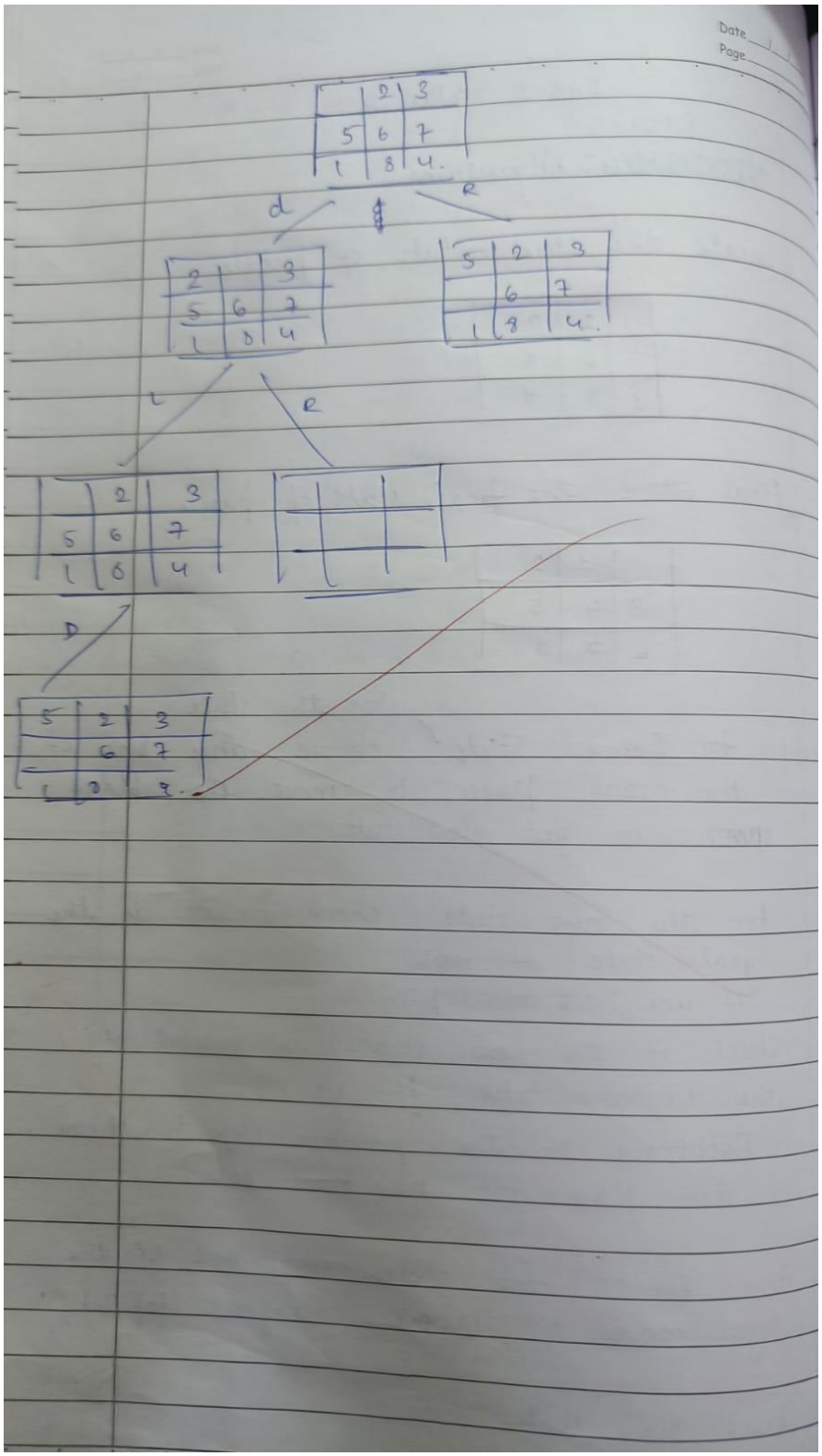
Step2. For the new state check if it is the goal state if not if yes go to step3.

Check if the new state is present in the visited states, if yes:

Backtrack to the previous step & choose a diff. place for the blank space.

if no. for the new choose any one of the legal moves & traverse. (Repeat Step 2).

Step3: Print the state.



Code:

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

```
    self.board = board
    self.moves = moves
    self.previous = previous
    self.empty_pos = self.find_empty()

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

def manhattan_distance(self):
    dist = 0
    for i in range(3):
        for j in range(3):
            tile = self.board[i][j]
            if tile != 0:
                target_x = (tile - 1) // 3
                target_y = (tile - 1) % 3
                dist += abs(i - target_x) + abs(j - target_y)
    return dist

def generate_moves(self):
    moves = []
    x, y = self.empty_pos
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_board = [row[:] for row in self.board]
            new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
```

```
new_board[x][y]
                moves.append(PuzzleState(new_board, self.moves + 1, self))

        return moves

def dfs(start_board, max_depth):
    stack = [PuzzleState(start_board)]
    visited = set()
    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

    while stack:
        current_state = stack.pop()
        if current_state.board == goal_state:
            return current_state
        visited.add(tuple(map(tuple, current_state.board)))
        if current_state.moves < max_depth:
            for next_state in current_state.generate_moves():
                if tuple(map(tuple, next_state.board)) not in visited:
                    if next_state.manhattan_distance() < 10:
                        stack.append(next_state)
    return None

def print_solution(solution):
    path = []
    while solution:
        path.append(solution.board)
        solution = solution.previous
    for step in reversed(path):
        for row in step:
            print(row)
        print()
    print(f"Total moves taken to reach the final state: {len(path) - 1}")
```

```
initial_board = [[1, 2, 3], [4, 0, 5], [7, 8, 6]]
max_depth = 10
solution = dfs(initial_board, max_depth)
if solution:
    print("Solution found:")
    print_solution(solution)
else:
    print("No solution found.")
```

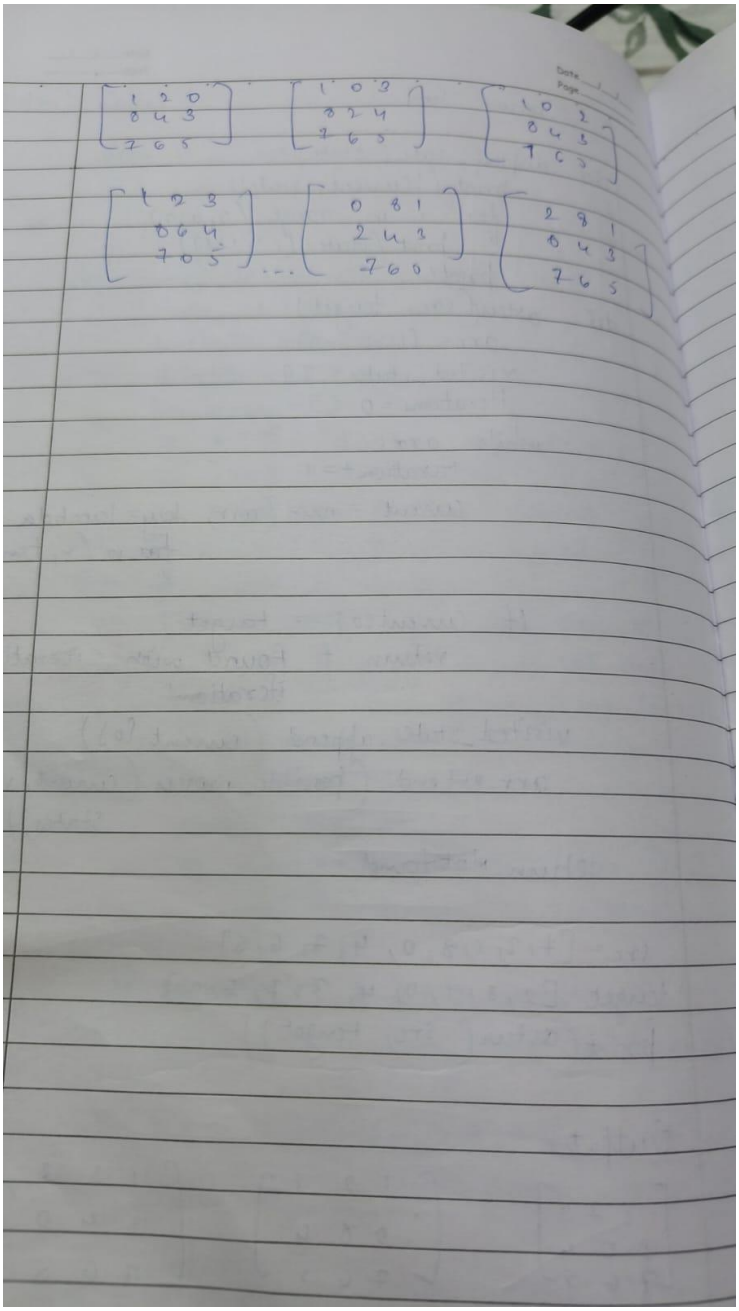
Output:

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

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

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

Total moves taken to reach the final state: 2
```

Program 4

Implement A* search algorithm

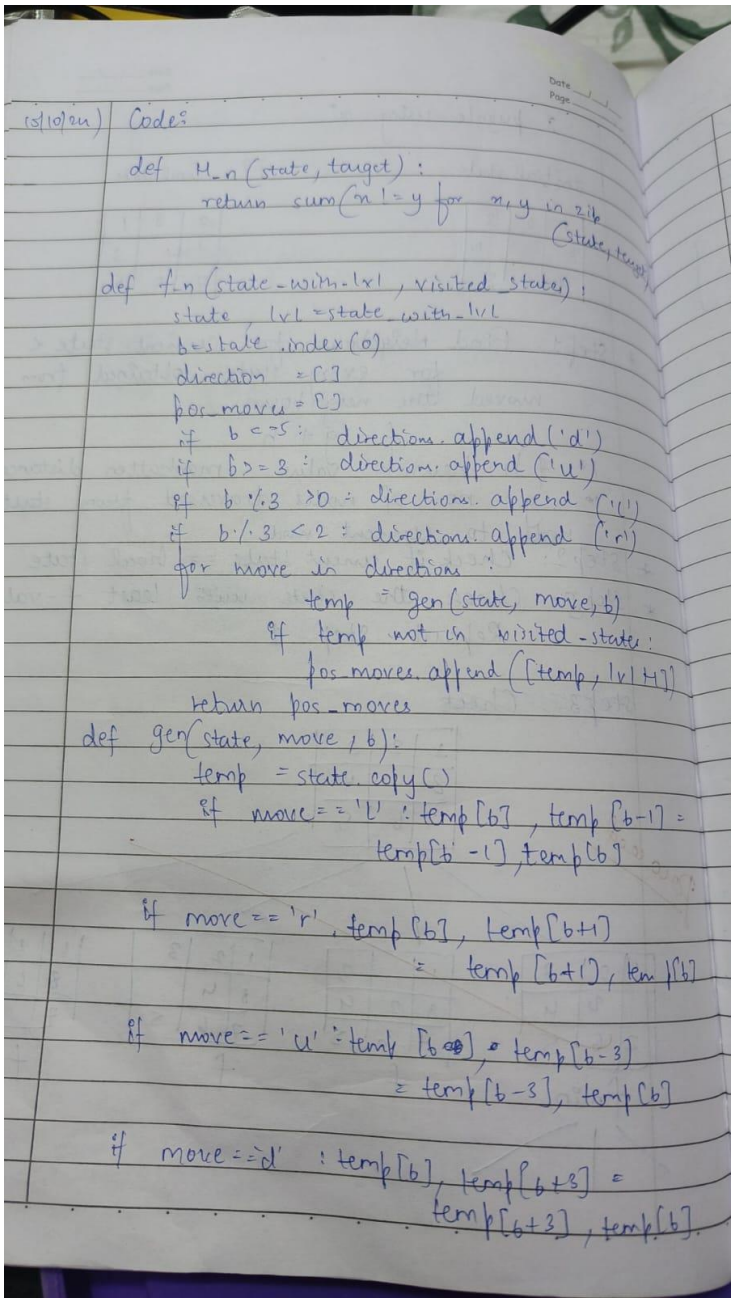
Algorithm:



```
    return temp

def display_state(state):
    print('Current State')
    for i in range(0,9,3)
        print(state[i:(i+3)])
    print()
def astar(src, target):
    arr = [(src, 0)]
    visited_states = []
    iterations = 0
    while arr:
        iterations += 1
        current = min(arr, key= lambda n:
                      for_n(n, target))

        if current[0] == target
            return f' Found with <iterations>
                    iterations'
        visited_states.append(current[0])
        arr.extend(possible_moves(current, visited_
                                  states))

    return Notfound

src = [1,2,3,8,0,4,7,6,5]
target = [2,8,1,0,4,3,7,6,5]
print(astar(src, target))
```

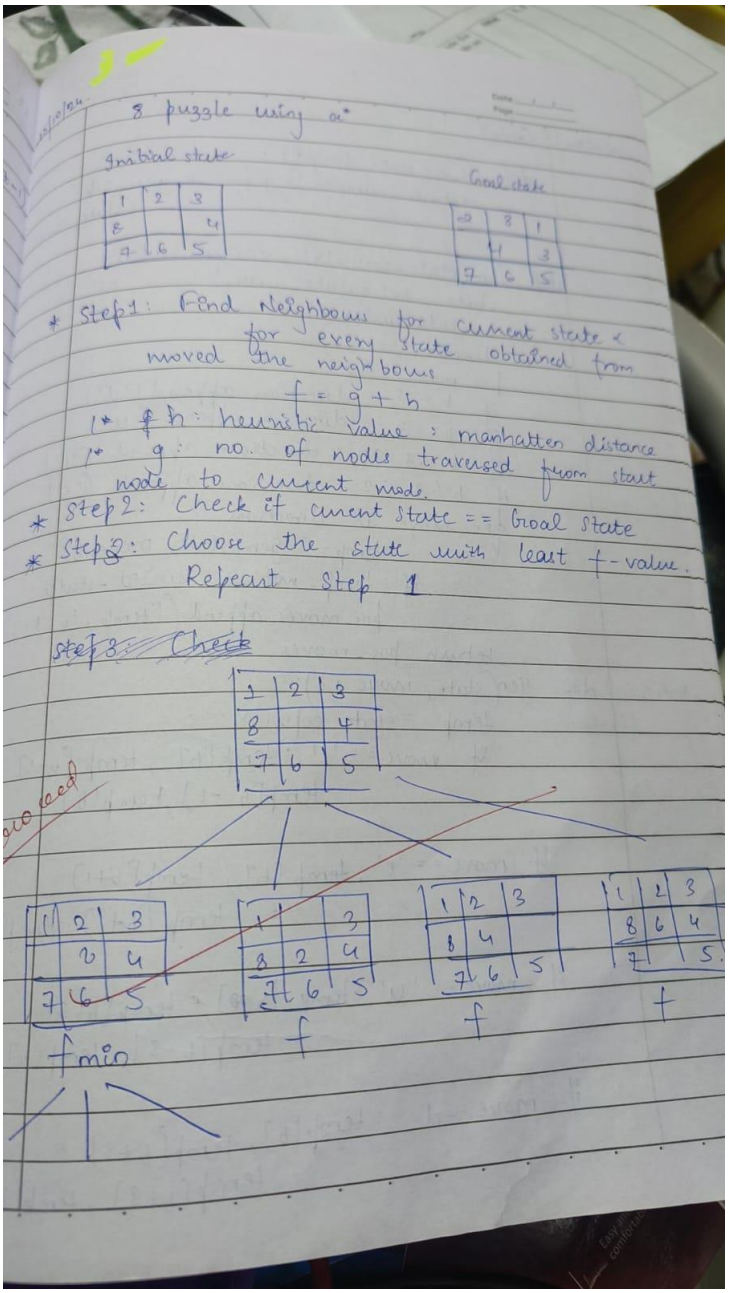
Output:

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



15/10/24) Code:

```
def H_n(state, target):
    return sum(x != y for x, y in zip(state, target))

def f_n(state_with_lvl, visited_states):
    state, lvl = state_with_lvl
    b = state.index(0)
    direction = []
    pos_moves = []
    if b <= 5: direction.append('d')
    if b >= 3: direction.append('u')
    if b % 3 > 0: direction.append('l')
    if b % 3 < 2: direction.append('r')
    for move in directions:
        temp = gen(state, move, b)
        if temp not in visited_states:
            pos_moves.append([temp, lvl+1])
    return pos_moves
def gen(state, move, b):
    temp = state.copy()
    if move == 'l': temp[b], temp[b-1] =
                    temp[b-1], temp[b]

    if move == 'r', temp[b], temp[b+1]
                = temp[b+1], temp[b]

    if move == 'u': temp[b], temp[b-3]
                = temp[b-3], temp[b]

    if move == 'd' : temp[b], temp[b+3] =
                    temp[b+3], temp[b].
```



Code:

```
def H_n(state, target):
    return sum(x != y for x, y in zip(state, target))
def F_n(state_with_lvl, target):
    state, lvl = state_with_lvl
    return H_n(state, target) + lvl
def possible_moves(state_with_lvl, visited_states):
```

```
    state, lvl = state_with_lvl
    b = state.index(0)
    directions = []
    pos_moves = []
    if b <= 5: directions.append('d')
    if b >= 3: directions.append('u')
    if b % 3 > 0: directions.append('l')
    if b % 3 < 2: directions.append('r')
    for move in directions:
        temp = gen(state, move, b)
        if temp not in visited_states:
            pos_moves.append([temp, lvl + 1])
    return pos_moves
def gen(state, move, b):
    temp = state.copy()
    if move == 'l': temp[b], temp[b - 1] = temp[b - 1], temp[b]
    if move == 'r': temp[b], temp[b + 1] = temp[b + 1], temp[b]
    if move == 'u': temp[b], temp[b - 3] = temp[b - 3], temp[b]
    if move == 'd': temp[b], temp[b + 3] = temp[b + 3], temp[b]
    return temp
def display_state(state):
    print("Current State:")
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()
def astar(src, target):
    arr = [[src, 0]]
    visited_states = []
    iterations = 0
    while arr:
        iterations += 1
        current = min(arr, key=lambda x: F_n(x, target))
```

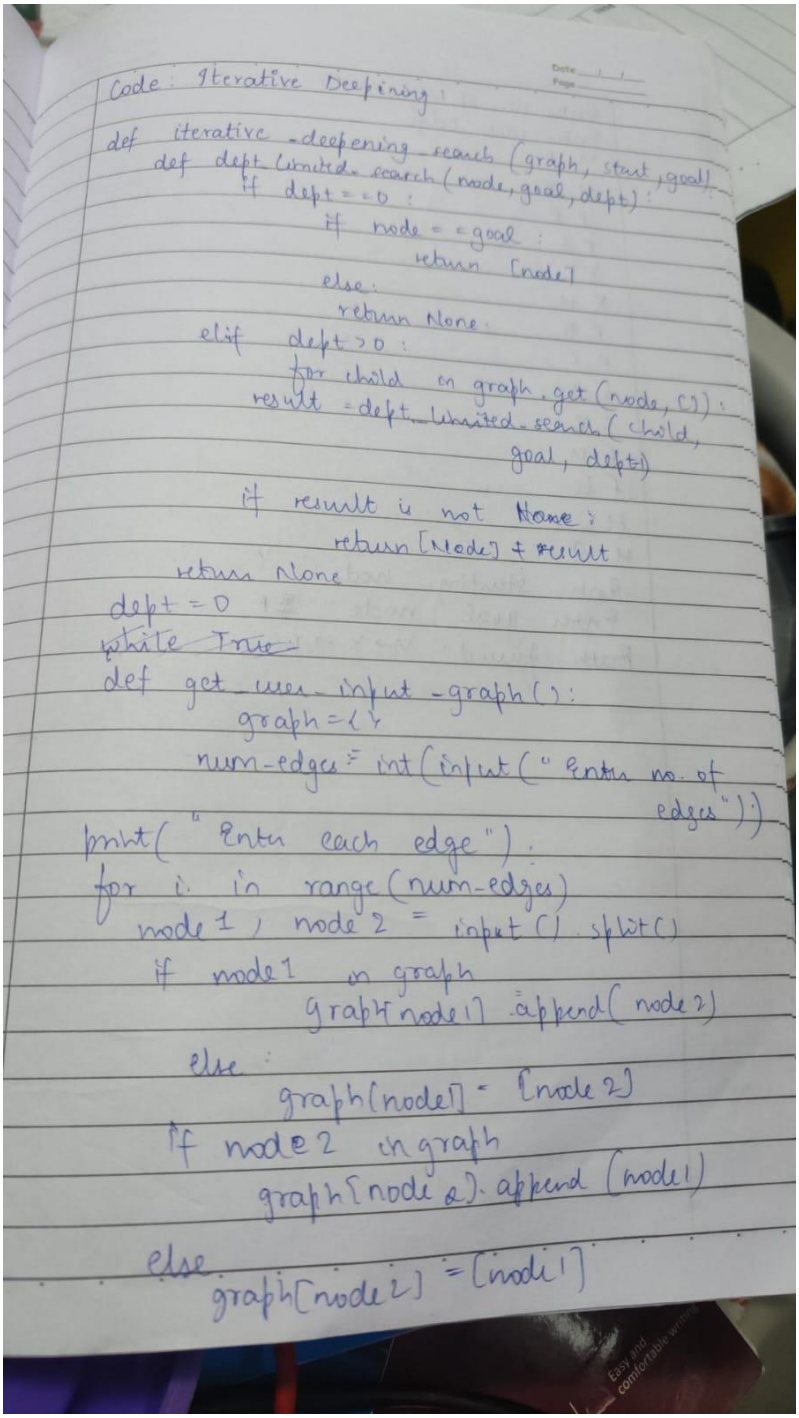
```
        arr.remove(current)
        display_state(current[0])
        if current[0] == target:
            return f'Found with {iterations} iterations'
        visited_states.append(current[0])
        arr.extend(possible_moves(current, visited_states))
    return 'Not found'
src = [1, 2, 3, 8, 0, 4, 7, 6, 5]
target = [2, 8, 1, 0, 4, 3, 7, 6, 5]
print(astar(src, target))
```

Output:

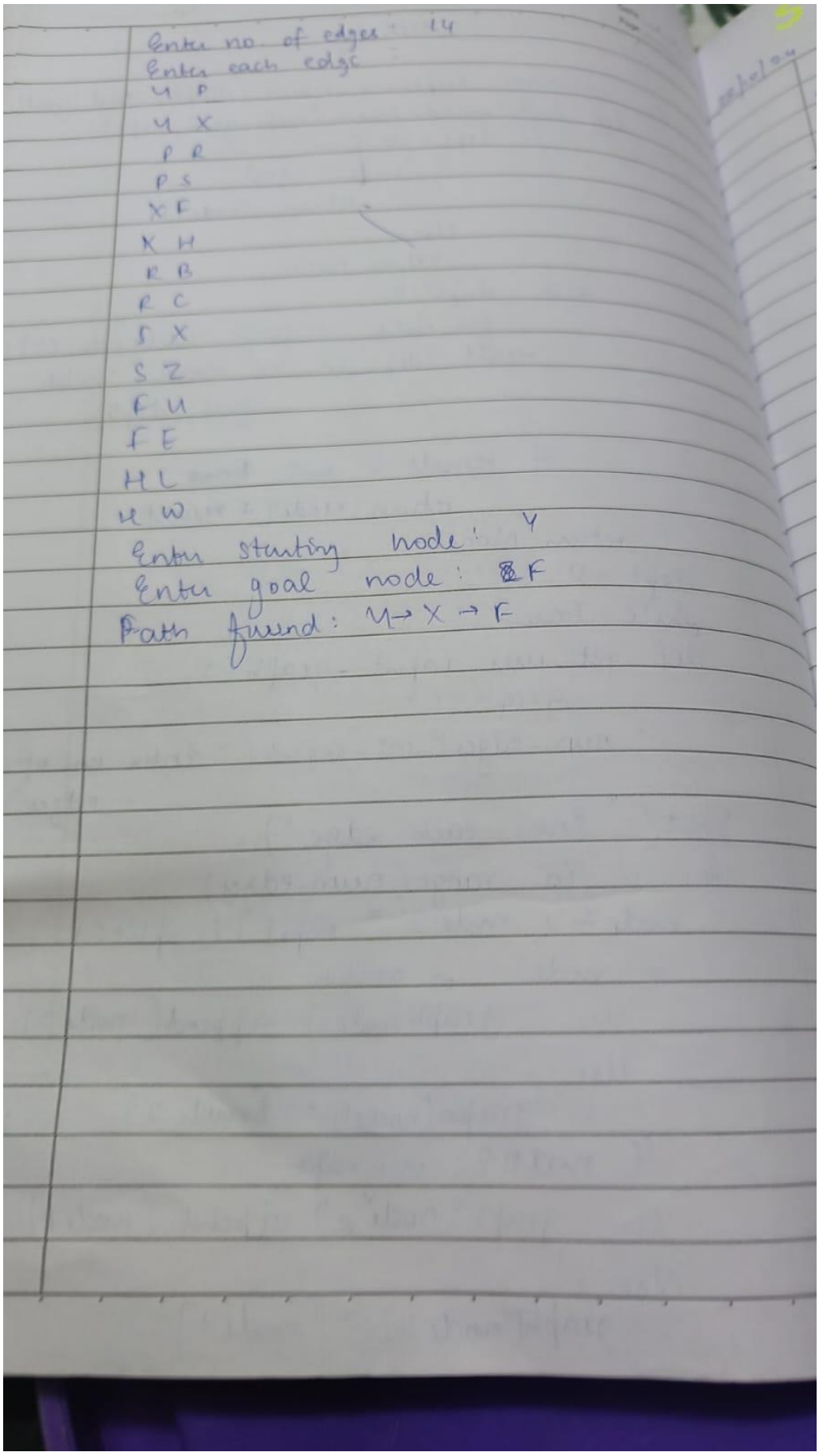
```
Current State:
[1, 3, 4]
[0, 8, 2]
[7, 6, 5]

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

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

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

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

Found with 40 iterations
```

Implement Iterative deepening search algorithm

Algorithm:



```
Code: Iterative Deepining:

def iterative_deepening_search(graph, start, goal):
    def depth_limited_search(node, goal, depth):
        if depth == 0:
            if node == goal:
                return [node]
            else:
                return None
        elif depth > 0:
            for child in graph.get(node, []):
                result = depth_limited_search(child,
                                              goal, depth-1)
                if result is not None:
                    return [node] + result
        return None
    depth = 0
    while True:
def get_user_input_graph():
    graph = {}
    num_edges = int(input("Enter no. of
                          edges"))
print("Enter each edge").
for i in range(num_edges)
    node1, node2 = input().split()
    if node1 in graph
        graph[node1].append(node2)
    else:
        graph[node1] = [node2]
    if node2 in graph
        graph[node2].append(node1)
    else
        graph[node2] = [node1]
```

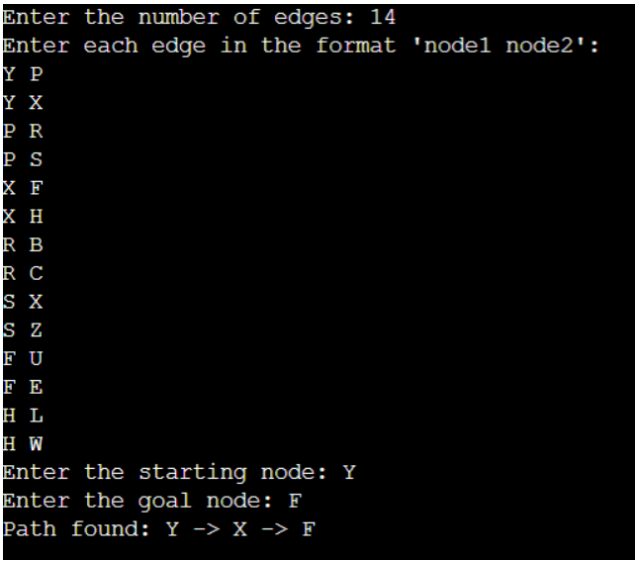

Enter no of edges: 14
Enter each edge
Y P
Y X
P R
P S
X F
X H
R B
R C
S X
S Z
F U
F E
H L
H W
Enter starting node: Y
Enter goal node: F
Path found: Y → X → F

Code:

```
def iterative_deepening_search(graph, start, goal):
    def depth_limited_search(node, goal, depth):
        if depth == 0:
            if node == goal:
                return [node]
            else:
                return None
        elif depth > 0:
            for child in graph.get(node, []):
                result = depth_limited_search(child, goal, depth - 1)
                if result is not None:
                    return [node] + result
        return None
    depth = 0
    while True:
        result = depth_limited_search(start, goal, depth)
        if result is not None:
            return result
        depth += 1
def get_user_input_graph():
    graph = {}
    num_edges = int(input("Enter the number of edges: "))
    print("Enter each edge in the format 'node1 node2':")
    for _ in range(num_edges):
        node1, node2 = input().split()
        if node1 in graph:
            graph[node1].append(node2)
        else:
            graph[node1] = [node2]
        if node2 in graph:
```

```
            graph[node2].append(node1)
        else:
            graph[node2] = [node1]
    return graph
def main():
    graph = get_user_input_graph()
    start_node = input("Enter the starting node: ")
    goal_node = input("Enter the goal node: ")
    path = iterative_deepening_search(graph, start_node, goal_node)
    if path:
        print(f"Path found: {' -> '.join(path)}")
    else:
        print("No path found")
if __name__ == "__main__":
    main()
```

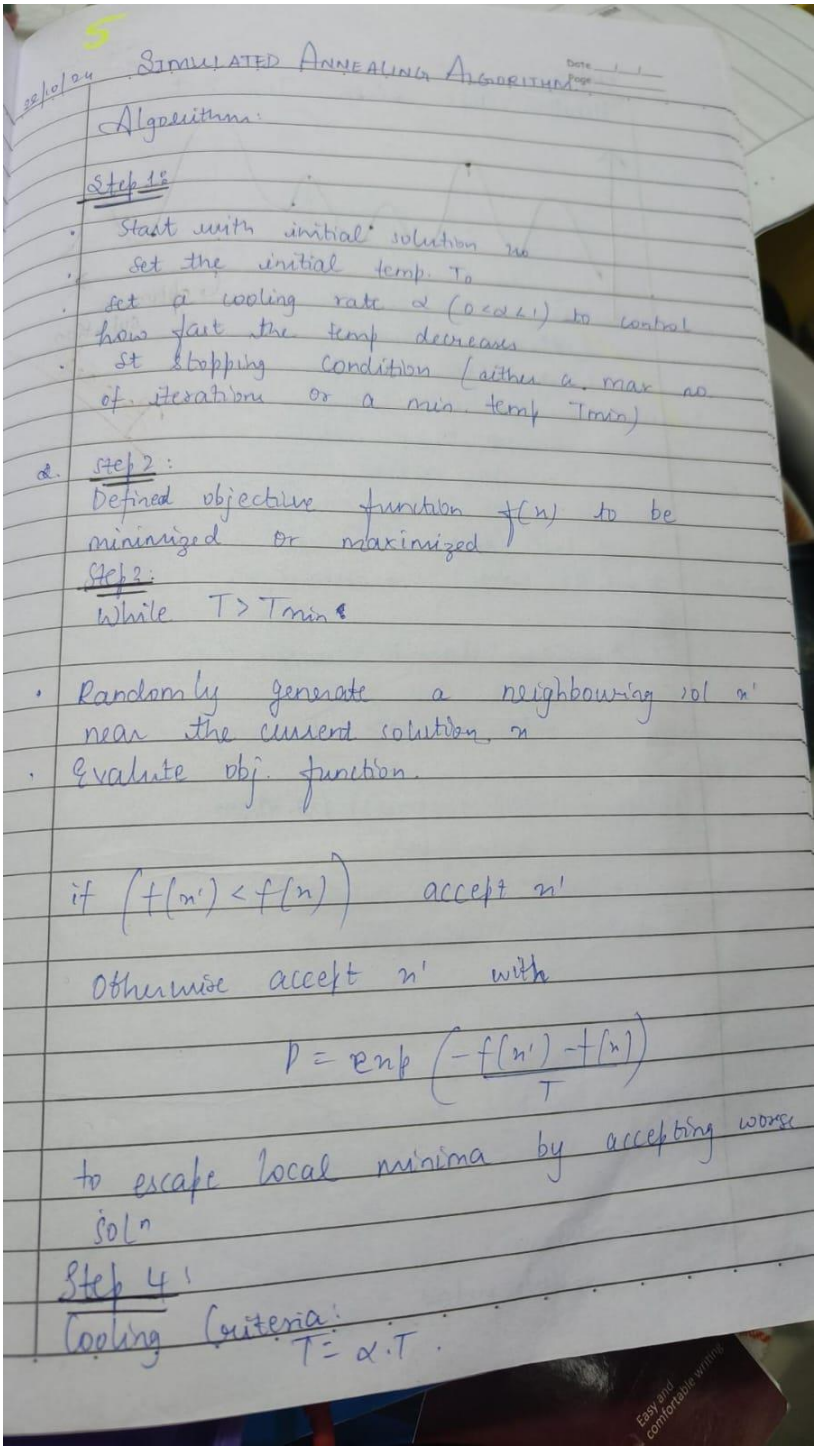
Output:

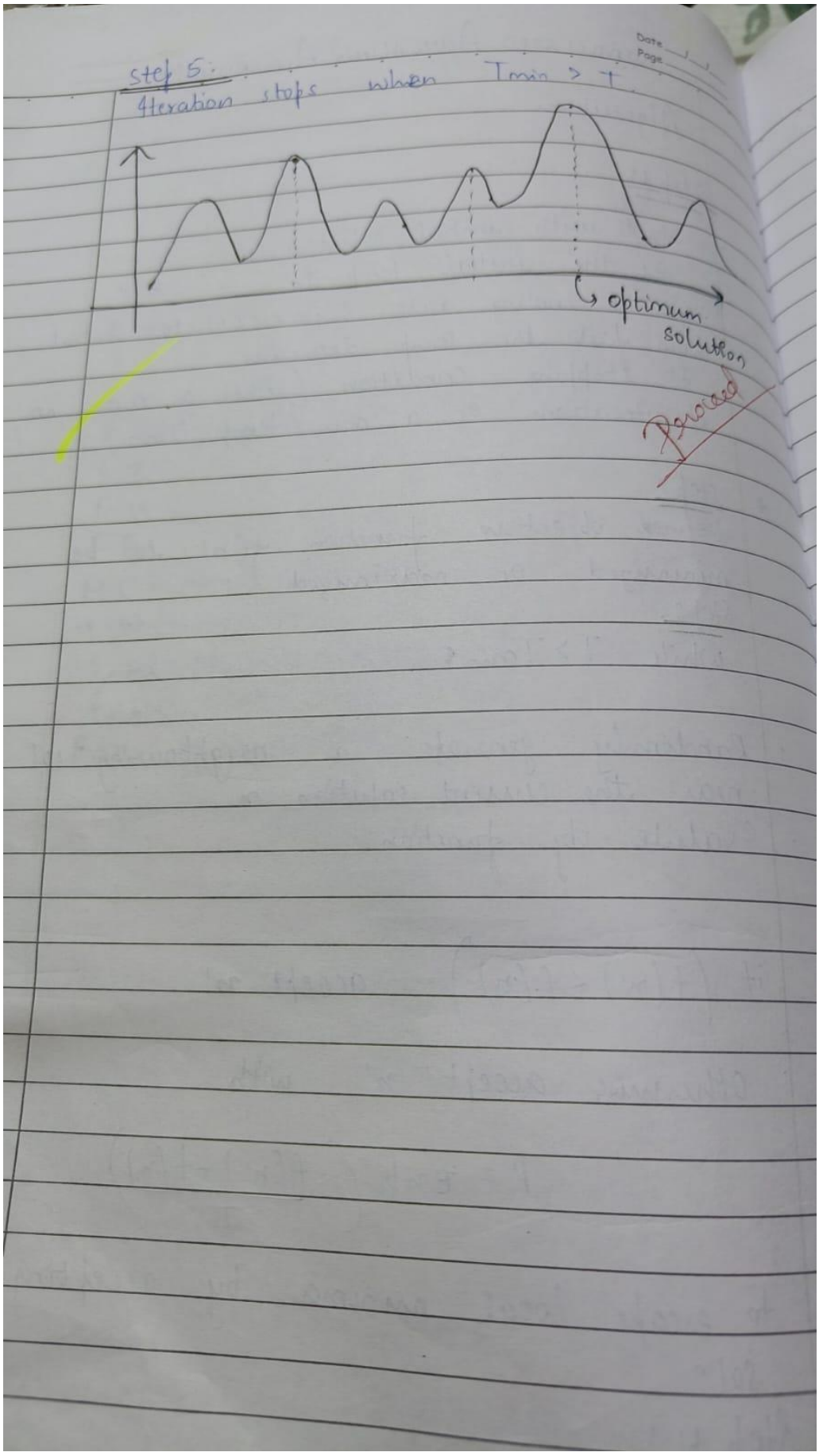


Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



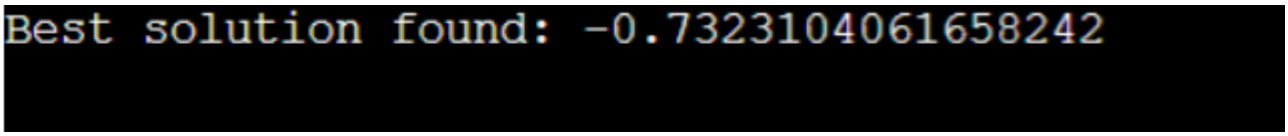

step 5:
Iteration stops when Tmin > T
optimum solution
Revised

Code:

```
import random
import math

def energy(x):
    return x ** 2 + 5 * math.sin(x) + math.exp(-x)

def adaptive_simulated_annealing(start, temp, cooling_rate, lower_limit, upper_limit):
    current = start
    current_energy = energy(current)

    while temp > 1:
        # Adaptive step size based on temperature (larger steps when hot)
        step_size = random.uniform(-1, 1) * temp
        new = current + step_size

        # Ensure new solution is within bounds
        if new < lower_limit or new > upper_limit:
            continue

        new_energy = energy(new)

        # If the new spot is better, move there
        if new_energy < current_energy:
            current = new
            current_energy = new_energy
        else:
            # Acceptance probability (explore worse spots)
            probability = math.exp((current_energy - new_energy) / temp)
            if random.uniform(0, 1) < probability:
                current = new
```

```
            current_energy = new_energy

        # Adaptive cooling based on progress
        if abs(new_energy - current_energy) < 0.01:
            temp *= 0.98  # Slow cooling near solution
        else:
            temp *= cooling_rate

    return current

# Run the simulation multiple times from different starting points
best_solution = None
for _ in range(10):  # 10 runs
    result = adaptive_simulated_annealing(start=random.uniform(-10, 10), temp=100,
cooling_rate=0.99, lower_limit=-10, upper_limit=10)
    if best_solution is None or energy(result) < energy(best_solution):
        best_solution = result

print(f"Best solution found: {best_solution}")
```

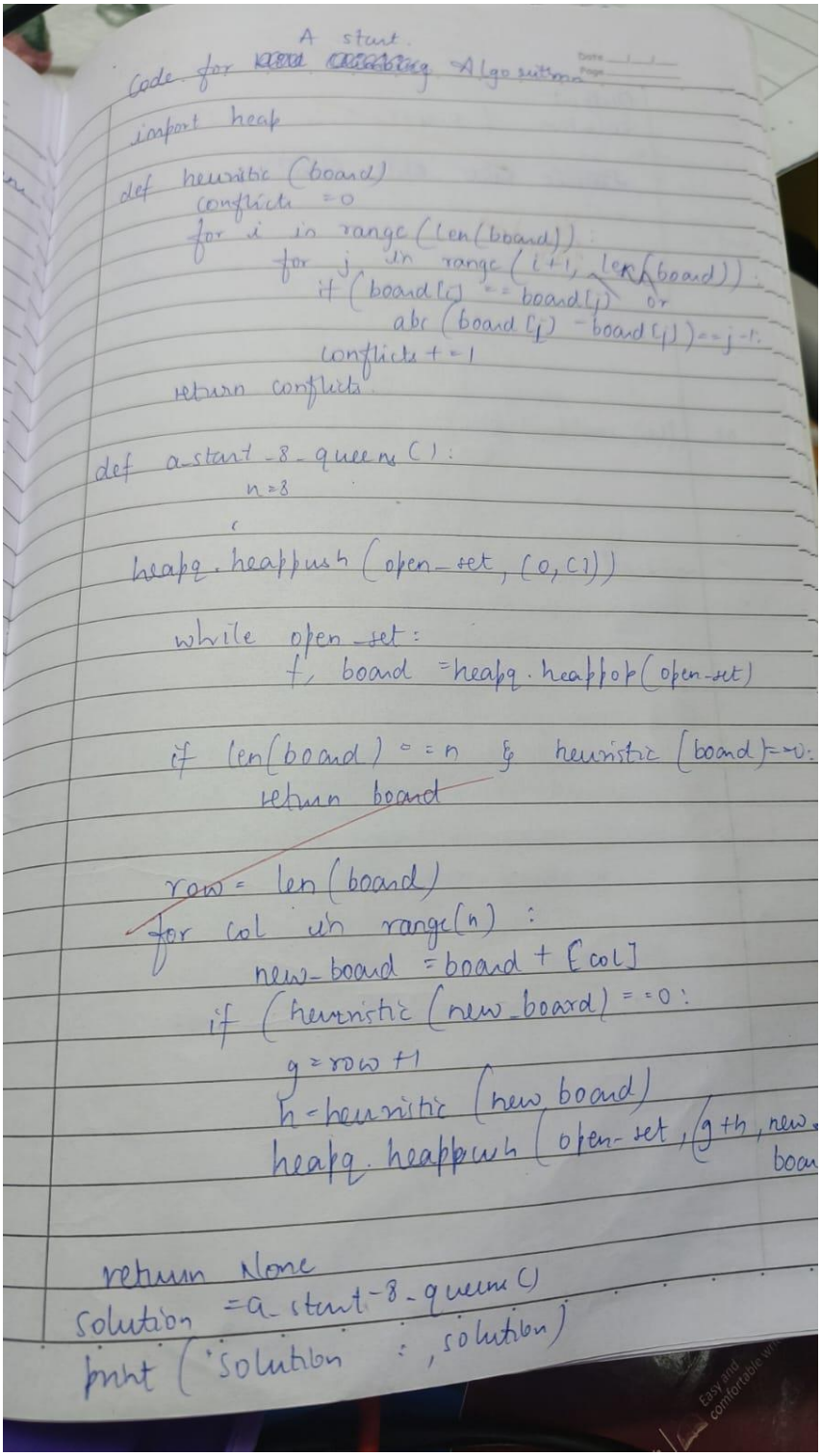
Output:



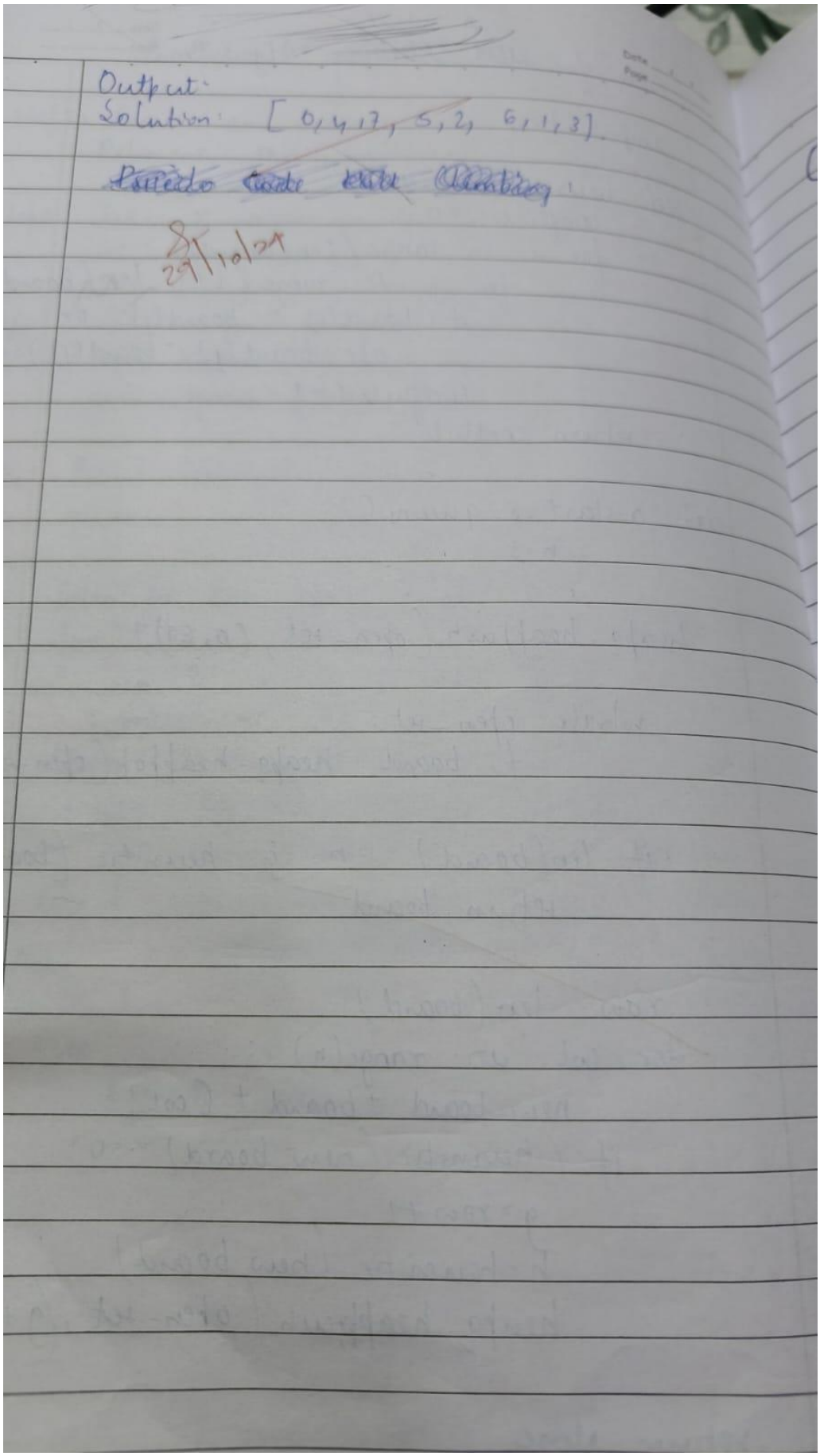
Program 6

Implement A* search algorithm for N queens

Algorithm:



```
A start.
Code for Algorithm
import heap
def heuristic (board)
    conflicts = 0
    for i in range (len(board)):
        for j in range (i+1, len(board)):
            if (board[i] == board[j] or
                abs (board[i] - board[j]) == j-i:
                conflicts += 1
    return conflicts

def a_start_8_queens ():
    n=8
    heapq.heappush (open_set, (0, []))
    while open_set:
        f, board = heapq.heappop(open_set)
    if len(board) == n & heuristic (board)==0:
        return board
    row = len (board)
    for col in range(n):
        new_board = board + [col]
        if (heuristic (new_board) == 0:
            g = row + 1
            h = heuristic (new board)
            heapq.heappush (open_set, (g+h, new_
                                       boa
    return None
solution = a_start_8_queens ()
print ('solution :, solution)
```


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

~~Pseudo code~~

29/10/24

8-Queens using A* Algorithm

Step 1: Create an initial state, with 0 queens. Represent this with queens[]

Step 2: Let 'g' value to 0
Set 'h' value to 0.

Step 3: Create a priority queue
Add states to pq based on priority over ~~queue~~ $f(n) = g(n) + h(n)$

Step 4: Insert initial empty state to priority queue.

Step 5: Identify the next empty row $g=0$ (here) for each col 0 to n-1, place a queen & create a new state (Increment g)

Step 6: Choose that state from priority queue with lowest $f(n)$ value

Check if $h=0$ & $g=N$
return solution.

Else
Repeat Step 5.

Code:

```
import heapq

# Helper function to calculate the heuristic (number of conflicts)
def heuristic(board):
    conflicts = 0
    for i in range(len(board)):
        for j in range(i + 1, len(board)):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                conflicts += 1
    return conflicts

# A* Search for 8-queens
def a_star_8_queens():
    n = 8
    open_set = []
    # Initial state: empty board
    heapq.heappush(open_set, (0, []))  # (f, board)

    while open_set:
        f, board = heapq.heappop(open_set)
        # Goal check
        if len(board) == n and heuristic(board) == 0:
          return board
        # Generate successors
        row = len(board)
        for col in range(n):
            new_board = board + [col]
            if heuristic(new_board) == 0:  # No conflicts so far
                g = row + 1
                h = heuristic(new_board)
```

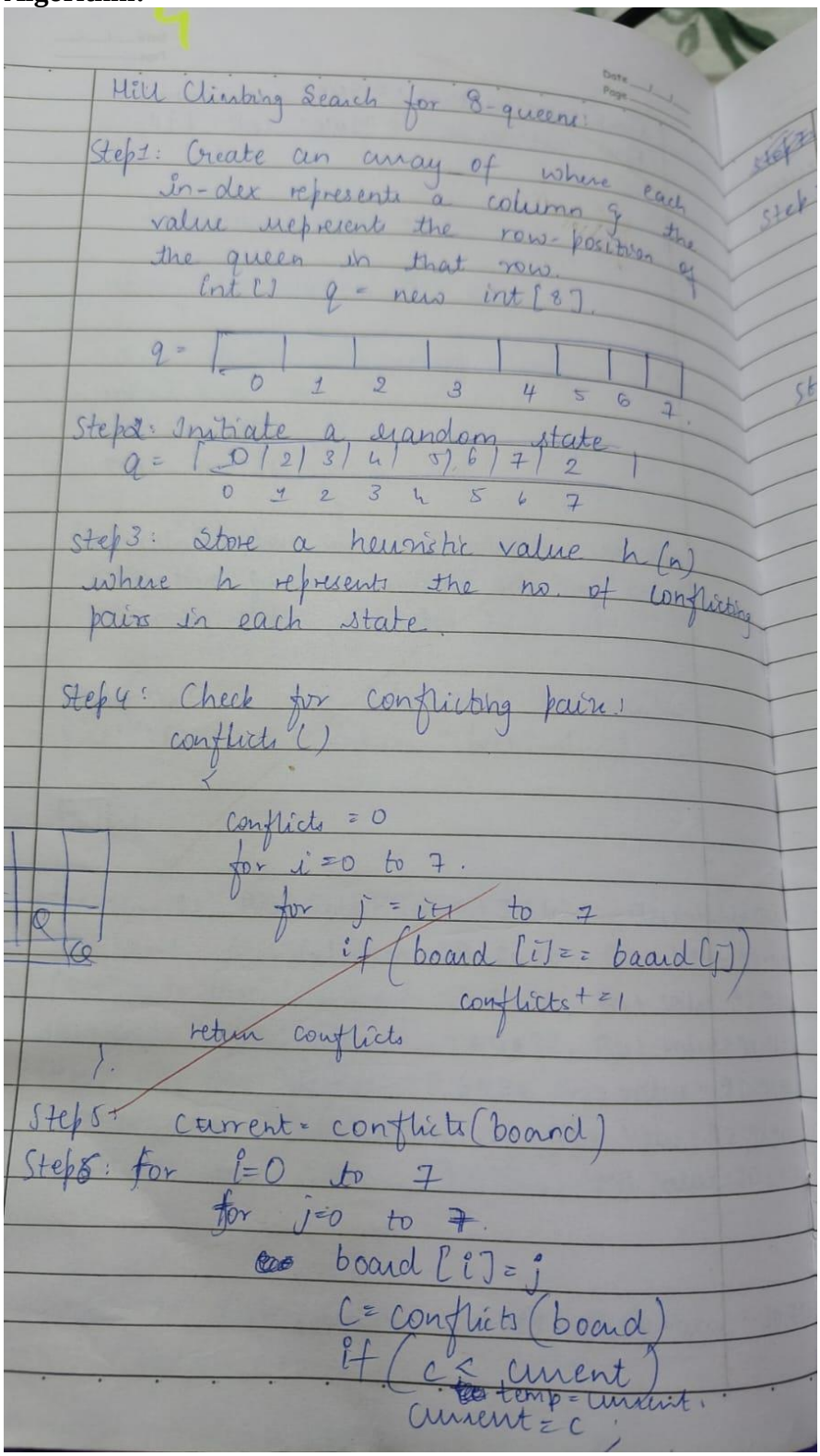
```
            heapq.heappush(open_set, (g + h, new_board))
    return None  # No solution found

# Run A* search
solution = a_star_8_queens()
print("Solution board (column positions for each row):", solution)
```

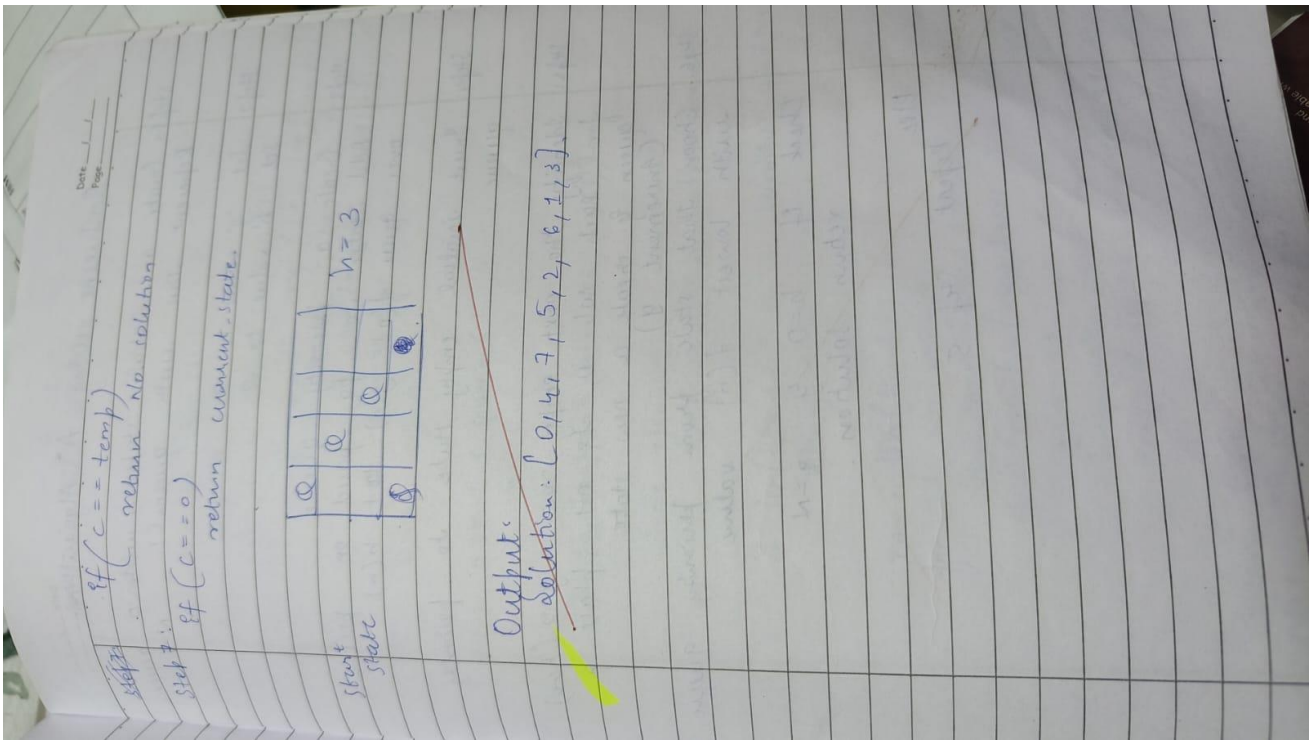
Output:

```
Solution board (column positions for each row): [0, 4, 7, 5, 2, 6, 1, 3]
```

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:





Code:

```
import random

# Helper function to calculate the heuristic (number of conflicts)
def heuristic(board):
    conflicts = 0
    for i in range(len(board)):
        for j in range(i + 1, len(board)):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                conflicts += 1
    return conflicts

# Hill climbing for 8-queens
def hill_climbing_8_queens():
    n = 8
    # Generate a random initial state
    board = [random.randint(0, n - 1) for _ in range(n)]
```

```
    while True:
        current_h = heuristic(board)
        if current_h == 0:
            return board  # Solution found

        # Find the best neighbor by moving each queen to every other column in its row
        best_board = board[:]
        best_h = current_h
        for row in range(n):
            for col in range(n):
                if col == board[row]:
                    continue
                new_board = board[:]
                new_board[row] = col
                new_h = heuristic(new_board)

                # If the new board has fewer conflicts, update the best board
                if new_h < best_h:
                    best_h = new_h
                    best_board = new_board

        # If no improvement, we're stuck in a local minimum; restart
        if best_h >= current_h:
            board = [random.randint(0, n - 1) for _ in range(n)]
        else:
            board = best_board

# Run hill climbing search
solution = hill_climbing_8_queens()
print("Solution board (column positions for each row):", solution)
```

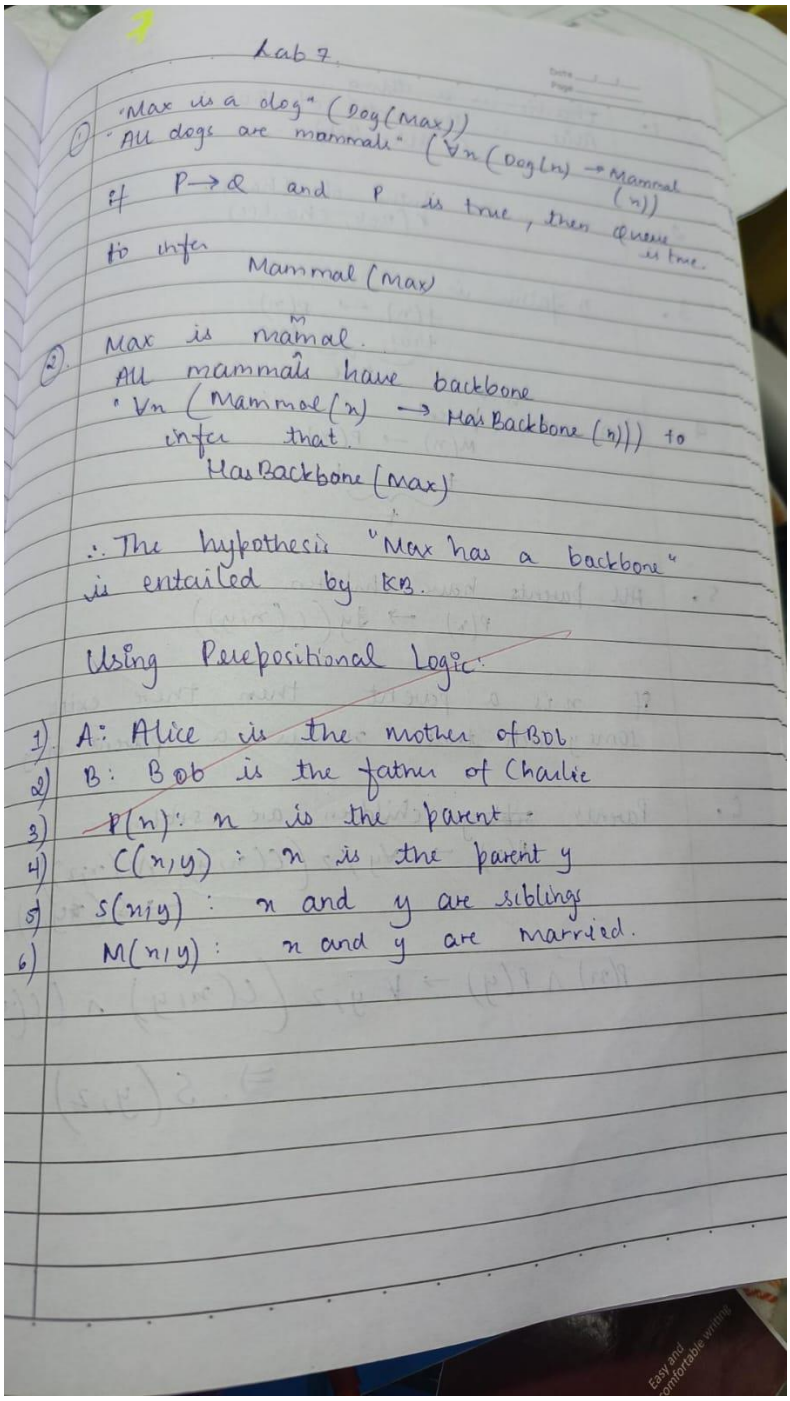
Output:

```
Solution board (column positions for each row): [0, 6, 3, 5, 7, 1, 4, 2]
```

Program 7

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

Algorithm:



1. ~~Charlie is a sibling~~
Alice is the mother of Bob:
C(Alice, Bob)

2. Bob is the father of Charlie
C(Bob, Charlie)

3. A father is a parent
F(x) → P(x)
true true
true

4. A mother is a parent -
M(x) → P(x)
t t
t

5. All parents have children.
$P(x) \rightarrow \exists y\,(C(x,y))$

If x is a parent then there exists some y such that x is a parent of y.

6. Parent's ~~siblings~~ children are sibling.
$P(x) \rightarrow \forall y, z\,(C(x,y) \wedge C(x,z)) \rightarrow S(y,z))$

$P(x) \wedge P(y) \rightarrow \forall y, z\,(C(x,y) \wedge (C($

$\Rightarrow S(y,z)$

to prove:

Charlie is a sibling of Bob
S(Bob, Charlie)

1) From Fact 4 & 4.
M(Alice) → P(Alice)

2) from F2 & F3.
F(Bob) → P(Bob)

3) If some x is a parent, their children are siblings.

from Fact 6.
P(Alice) P(Bob)
T T.

∴ S(Bob, Charlie).

P(Alice) ∧ P(Charlie)

∴ it is entailed by KB

from 5.
P(Alice) → C(Alice, Bob)
P(Bob) → C(Bob, Charlie)

from 6.

P(Alice) ∧ P(Bob) ⇒ ∃C((Alice, Bob) ∧ (Bob, Charlie))

⇒ S(Bob, Charlie)

Code:

```
import random

# Helper function to calculate the heuristic (number of conflicts)
def heuristic(board):
    conflicts = 0
    for i in range(len(board)):
        for j in range(i + 1, len(board)):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                conflicts += 1
    return conflicts

# Hill climbing for 8-queens
def hill_climbing_8_queens():
    n = 8
    # Generate a random initial state
    board = [random.randint(0, n - 1) for _ in range(n)]

    while True:
        current_h = heuristic(board)
        if current_h == 0:
            return board  # Solution found

        # Find the best neighbor by moving each queen to every other column in its row
        best_board = board[:]
        best_h = current_h
        for row in range(n):
            for col in range(n):
                if col == board[row]:
                    continue
                new_board = board[:]
```

```
            new_board[row] = col
            new_h = heuristic(new_board)

            # If the new board has fewer conflicts, update the best board
            if new_h < best_h:
                best_h = new_h
                best_board = new_board

        # If no improvement, we're stuck in a local minimum; restart
        if best_h >= current_h:
            board = [random.randint(0, n - 1) for _ in range(n)]
        else:
            board = best_board

# Run hill climbing search
solution = hill_climbing_8_queens()
print("Solution board (column positions for each row):", solution)
```

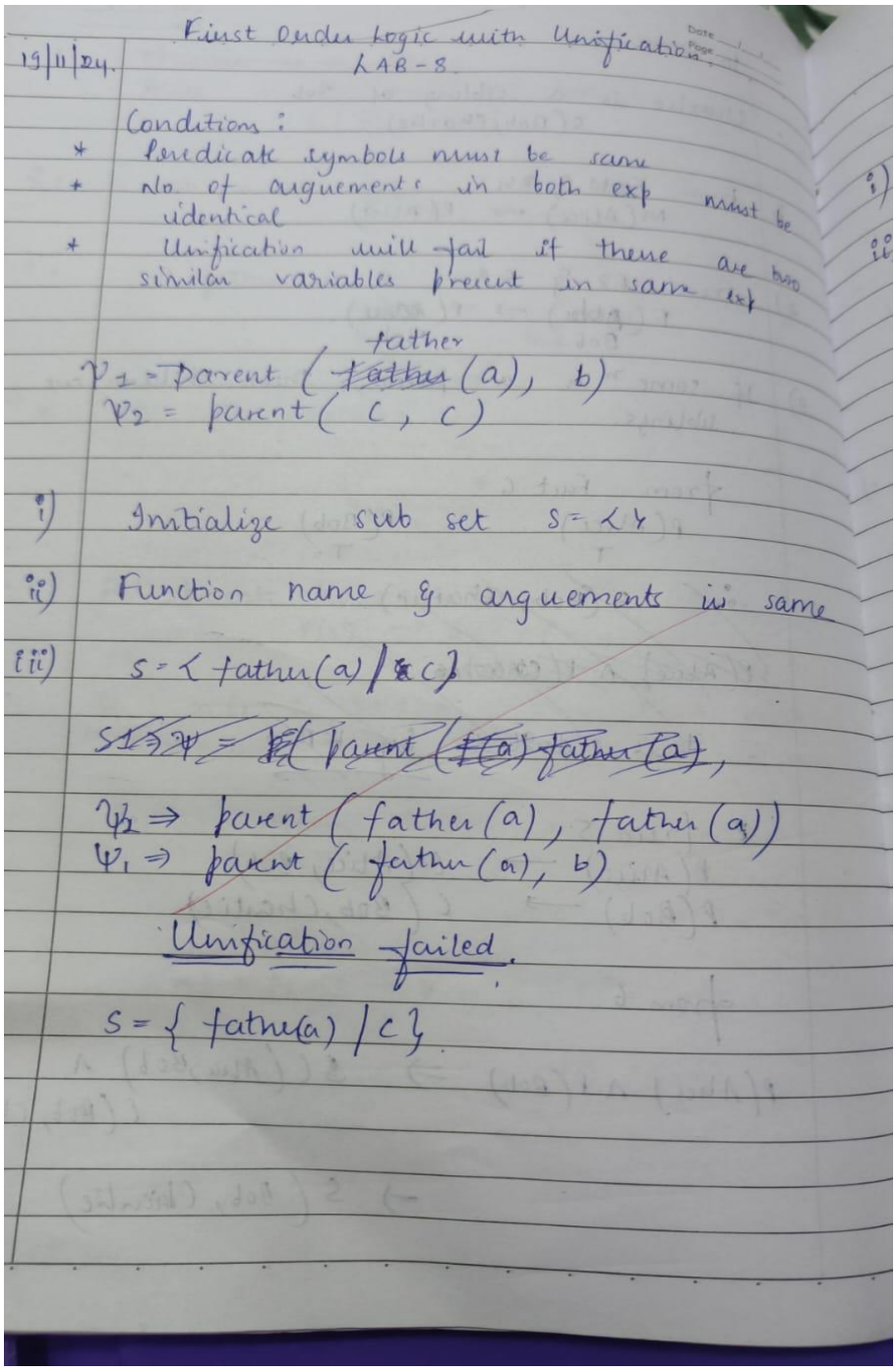
Output:

```
The hypothesis 'Charlie is a sibling of Bob' is FALSE.
```

Program 8

Implement unification in first order logic

Algorithm:



ψ_1 = parent (father (a), b)
ψ_2 = parent (X, Y)

i) Initialize subset $S = \{\}$

ii) for variable x in ψ_2, term father (a) in ψ_1

$S = \{$ father (a) / X $\}$

$S \Rightarrow \psi_1$ = parent (father (a), b)
ψ_2 = parent (father (a), Y)

iii) for variable Y, term b.

$S = \{$ father (a) / X, b / Y $\}$

$S \Rightarrow \psi_1$ = parent (father (a), b)
ψ_2 = parent (father (a), b)

Unified Successfully

$\therefore$ MGU = S.

Code Output:

Unification failed:
Unified Successfully : {'b': 2, 'X': f(Y), 'Y': g(Z)}

Code:

```
def unify(x, y, subst=None):
    """
    Unification Algorithm: Unifies two terms, X and Y.
    """
    if subst is None:
        subst = {}

    if x == y:  # Step 1(a): If X and Y are identical
        return subst
    elif isinstance(x, str) and x.islower():  # Step 1(b): If X is a variable
        return unify_variable(x, y, subst)
    elif isinstance(y, str) and y.islower():  # Step 1(c): If Y is a variable
        return unify_variable(y, x, subst)
    elif isinstance(x, tuple) and isinstance(y, tuple):  # Step 2: Check predicates and arguments
        if x[0] != y[0] or len(x) != len(y):  # Predicate symbol or argument count mismatch
            return None
        for x_i, y_i in zip(x[1:], y[1:]):  # Step 5: Recurse through arguments
            subst = unify(x_i, y_i, subst)
            if subst is None:
                return None
        return subst
    else:
        return None  # Step 1(d): Failure case


def unify_variable(var, x, subst):
    """
    Unify variable with another term.
    """
    if var in subst:
```

```
        return unify(subst[var], x, subst)
    elif occurs_check(var, x, subst):  # Check if var occurs in x
        return None
    else:
        subst[var] = x
        return subst


def occurs_check(var, x, subst):
    """
    Check if a variable occurs in a term.
    """
    if var == x:
        return True
    elif isinstance(x, tuple):
        return any(occurs_check(var, xi, subst) for xi in x)
    elif isinstance(x, str) and x in subst:
        return occurs_check(var, subst[x], subst)
    return False


# Test cases for unification
x1 = ("P", "a", "x")
y1 = ("P", "a", "b")

x2 = ("Q", "x", ("R", "x"))
y2 = ("Q", "a", ("R", "a"))

print("Unifying", x1, "and", y1, "=>", unify(x1, y1))
print("Unifying", x2, "and", y2, "=>", unify(x2, y2))
```

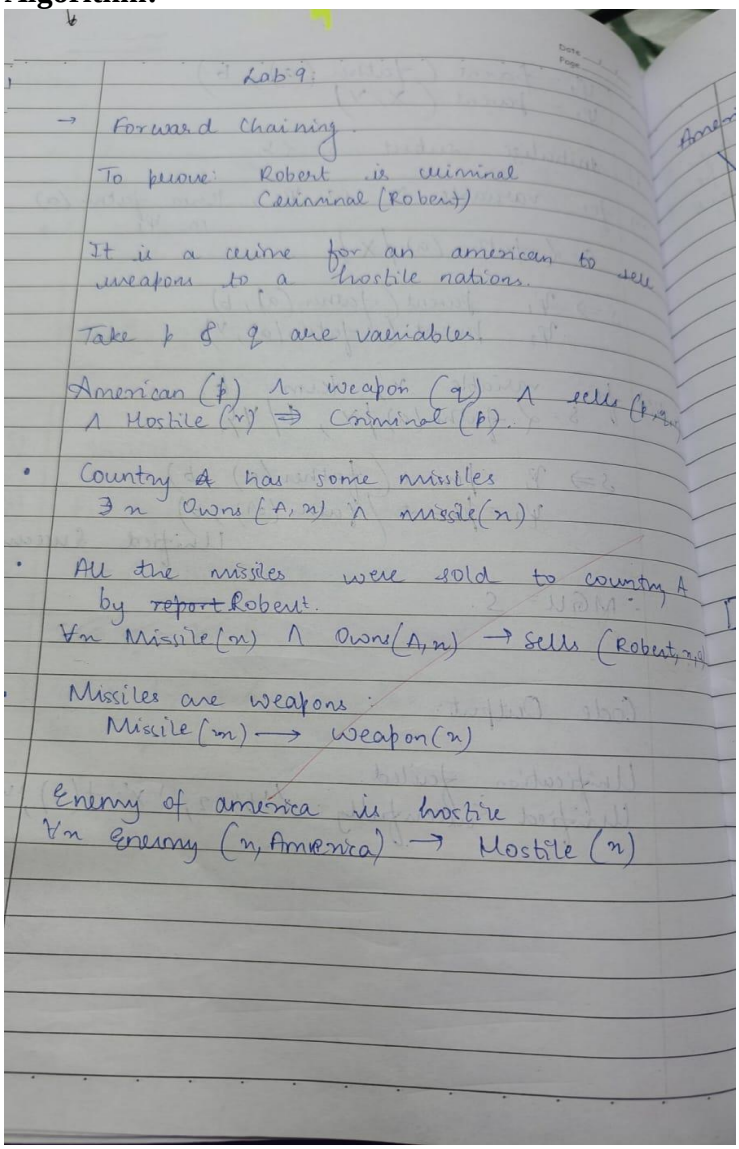
Output:

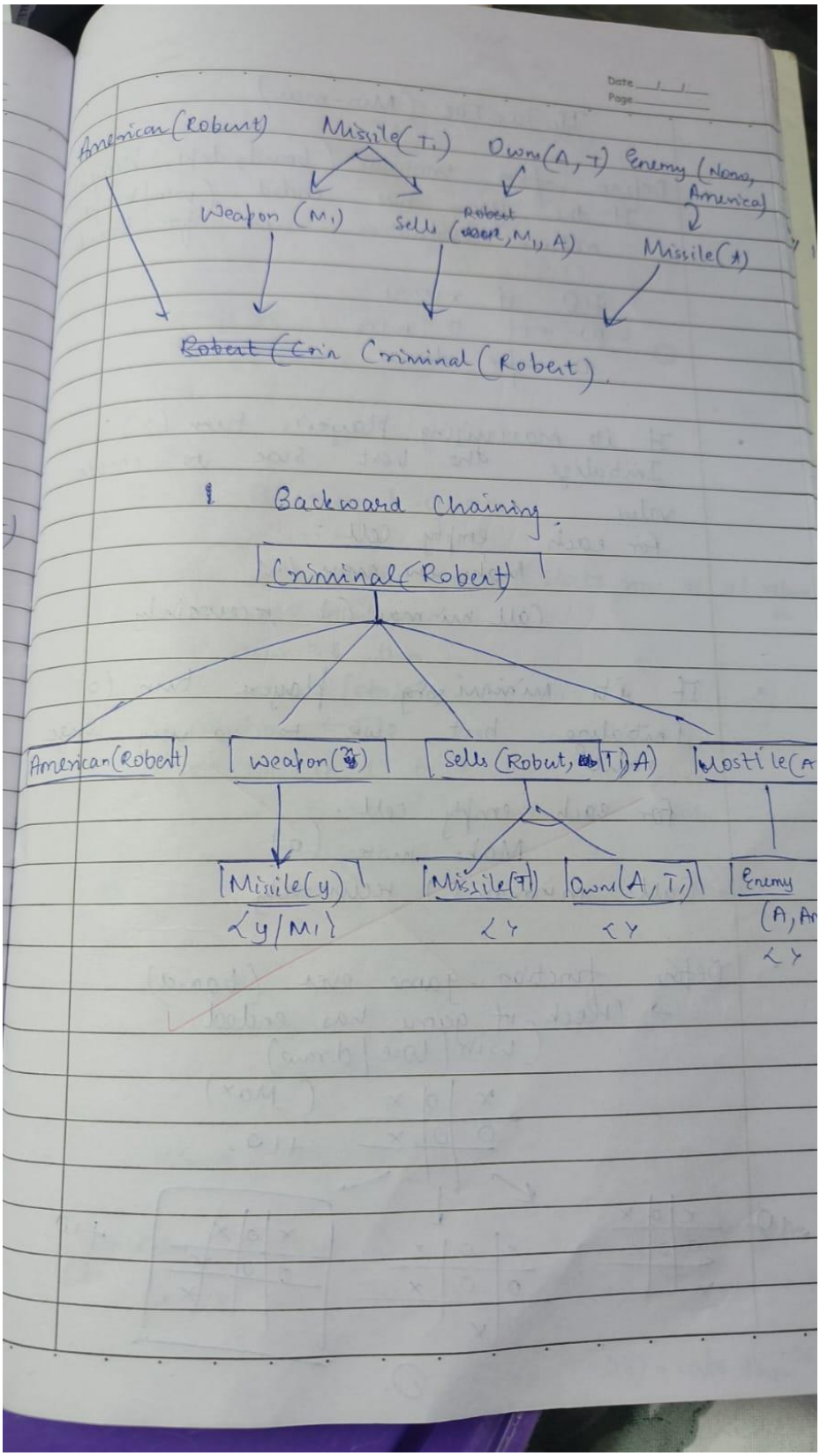
```
Unifying ('P', 'a', 'x') and ('P', 'a', 'b') => {'x': 'b'}
Unifying ('Q', 'x', ('R', 'x')) and ('Q', 'a', ('R', 'a')) => {'x': 'a'}
```

Program 9

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

Algorithm:




American(Robert)
Missile(T1)
Own(A, T)
Enemy(Nono, America)
Weapon(M1)
Sells(Robert, M1, A)
Missile(T)
Criminal(Robert)
Backward Chaining
Criminal(Robert)
American(Robert)
Weapon(y)
Sells(Robert, T1, A)
Hostile(A
Missile(y)
Missile(T1)
Own(A, T1)
Enemy
{y/M1}

Code:

```
# Define the knowledge base (KB) as a set of facts
KB = set()

# Premises based on the provided FOL problem
KB.add('American(Robert)')
KB.add('Enemy(America, A)')
KB.add('Missile(T1)')
KB.add('Owns(A, T1)')

# Define inference rules
def modus_ponens(fact1, fact2, conclusion):
    """ Apply modus ponens inference rule: if fact1 and fact2 are true, then conclude conclusion """
    if fact1 in KB and fact2 in KB:
        KB.add(conclusion)
        print(f"Inferred: {conclusion}")

def forward_chaining():
    """ Perform forward chaining to infer new facts until no more inferences can be made """
    # 1. Apply: Missile(x) → Weapon(x)
    if 'Missile(T1)' in KB:
        KB.add('Weapon(T1)')
        print(f"Inferred: Weapon(T1)")

    # 2. Apply: Sells(Robert, T1, A) from Owns(A, T1) and Weapon(T1)
    if 'Owns(A, T1)' in KB and 'Weapon(T1)' in KB:
        KB.add('Sells(Robert, T1, A)')
        print(f"Inferred: Sells(Robert, T1, A)")

    # 3. Apply: Hostile(A) from Enemy(A, America)
    if 'Enemy(America, A)' in KB:
```

```
        KB.add('Hostile(A)')
        print(f"Inferred: Hostile(A)")

    # 4. Now, check if the goal is reached (i.e., if 'Criminal(Robert)' can be inferred)
    if 'American(Robert)' in KB and 'Weapon(T1)' in KB and 'Sells(Robert, T1, A)' in KB and 'Hostile(A)' in KB:
        KB.add('Criminal(Robert)')
        print("Inferred: Criminal(Robert)")

    # Check if we've reached our goal
    if 'Criminal(Robert)' in KB:
        print("Robert is a criminal!")
    else:
        print("No more inferences can be made.")

# Run forward chaining to attempt to derive the conclusion
forward_chaining()
```

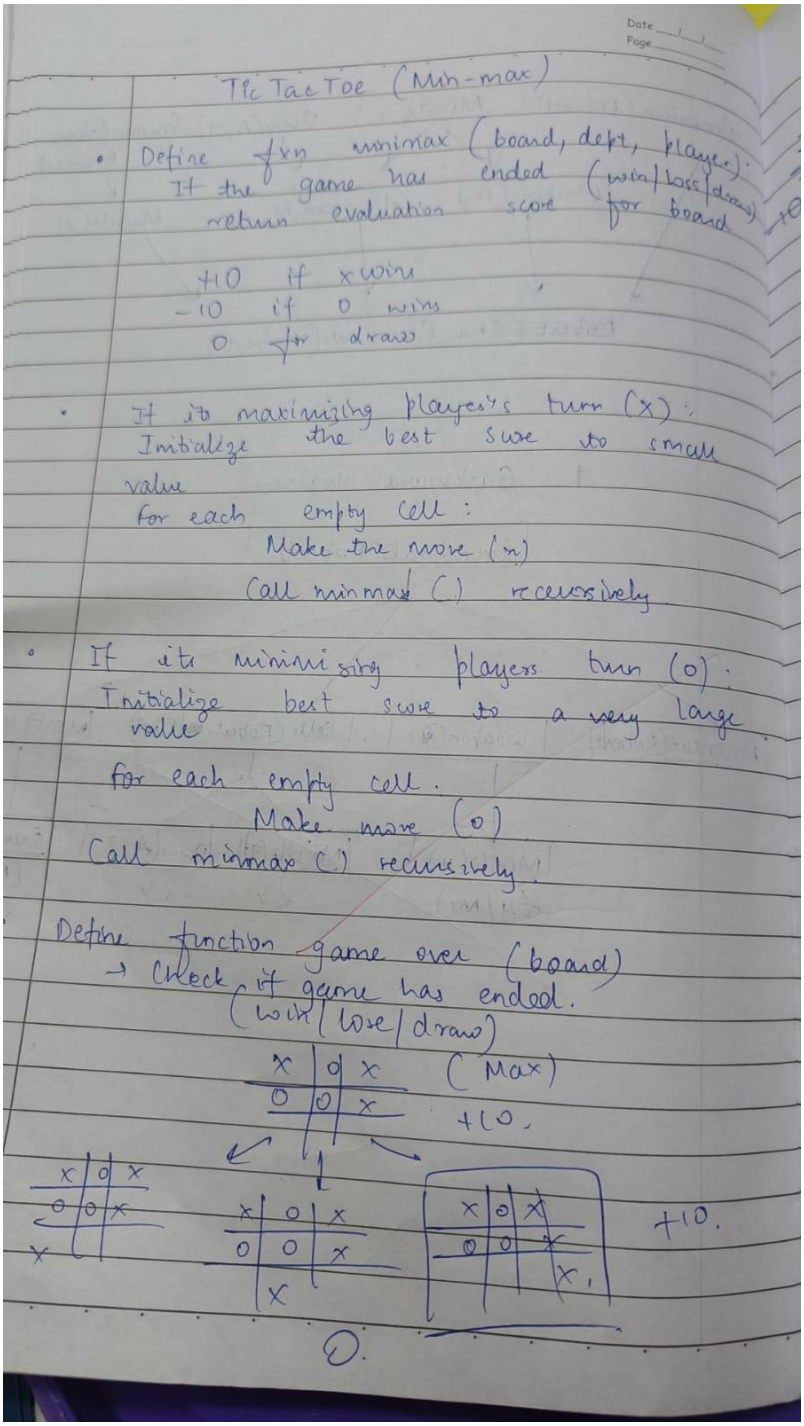
Output:

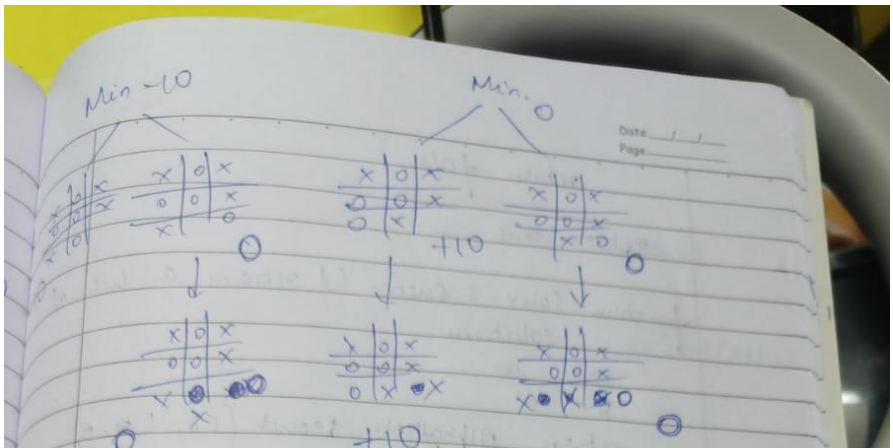
```
Inferred: Weapon(T1)
Inferred: Sells(Robert, T1, A)
Inferred: Hostile(A)
Inferred: Criminal(Robert)
Robert is a criminal!
```

Program 10

Implement Min-Max Algorithm for Tic Tac Toe

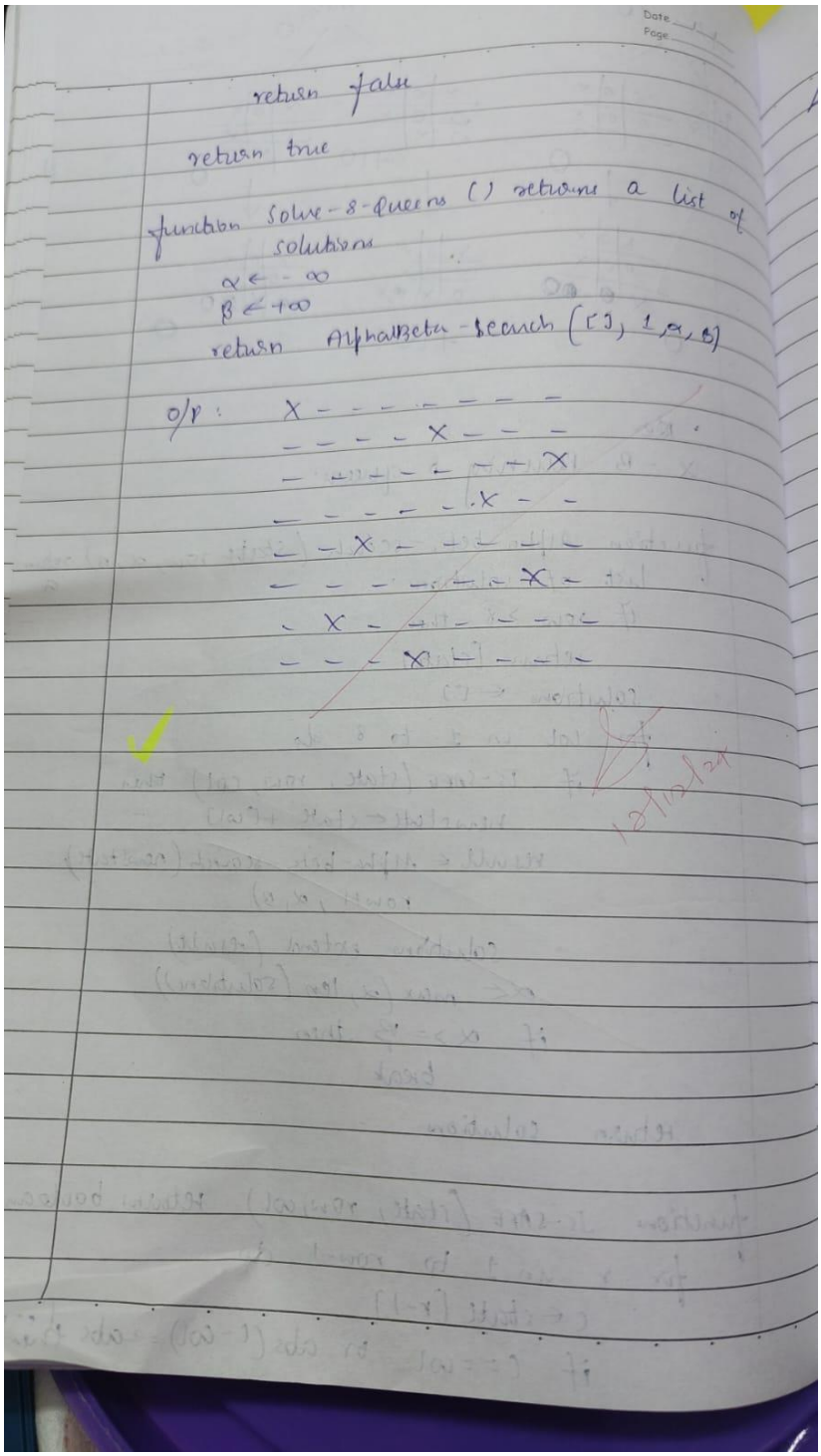
Algorithm:





α - β Pruning 8 queens:

```
function Alpha-beta-search (state row, α, β) return a
    list of solutions.
    if row > 8 then
        return [state]
    solutions ← []
    for col in 1 to 8 do
        if IS-SAFE (state, row, col) then
            newstate ← state + [col]
            result ← Alpha-beta-search (newstate)
                row+1, α, β)
            solutions.extend (result)
            α ← max (α, len (solutions))
            if α >= β then
                break
    return solutions

function IS-SAFE (state, row, col) returns boolean
    for r in 1 to row-1 do
        c ← state [r-1]
        if c == col or abs (c - col) == abs (r-row) then
```



Code:

import math

Constants for the players

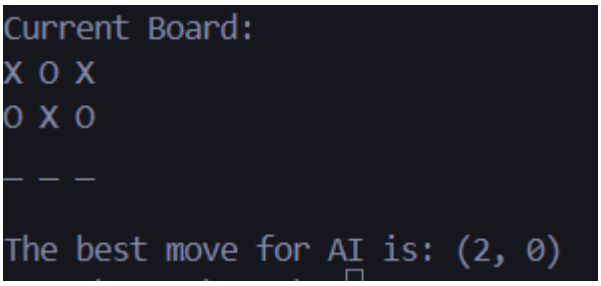
AI = 'X'

```
HUMAN = 'O'
EMPTY = '_'

# Function to print the board
def print_board(board):
    for row in board:
        print(" ".join(row))
    print()

# Function to check if a player has won
def check_winner(board, player):
    # Check rows, columns, and diagonals
    for row in board:
        if all(cell == player for cell in row):
            return True
    for col in range(3):
        if all(row[col] == player for row in board):
            return True
    if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in range(3)):
        return True
    return False

# Function to check if the game is a draw
def is_draw(board):
    return all(cell != EMPTY for row in board for cell in row)

# Minimax algorithm
def minimax(board, depth, is_maximizing):
    if check_winner(board, AI):
        return 10 - depth
    if check_winner(board, HUMAN):
        return depth - 10
```

```
    if is_draw(board):
        return 0

    if is_maximizing:
        best_score = -math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == EMPTY:
                    board[i][j] = AI
                    score = minimax(board, depth + 1, False)
                    board[i][j] = EMPTY
                    best_score = max(best_score, score)
        return best_score
    else:
        best_score = math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == EMPTY:
                    board[i][j] = HUMAN
                    score = minimax(board, depth + 1, True)
                    board[i][j] = EMPTY
                    best_score = min(best_score, score)
        return best_score

# Function to find the best move for AI
def find_best_move(board):
    best_score = -math.inf
    move = (-1, -1)
    for i in range(3):
        for j in range(3):
            if board[i][j] == EMPTY:
                board[i][j] = AI
```

```
                score = minimax(board, 0, False)
                board[i][j] = EMPTY
                if score > best_score:
                    best_score = score
                    move = (i, j)
    return move

# Example usage
if __name__ == "__main__":
    # Initialize a sample board
    board = [
        ['X', 'O', 'X'],
        ['O', 'X', 'O'],
        ['_', '_', '_']
    ]
    print("Current Board:")
    print_board(board)

    best_move = find_best_move(board)
    print(f"The best move for AI is: {best_move}")
```

Output:



Implement Alpha-Beta Pruning for 8 queens

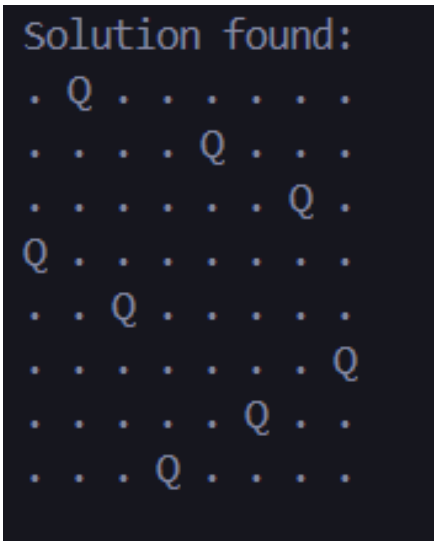
Code:

```
class EightQueens:
    def __init__(self, size=8):
        self.size = size

    def is_safe(self, board, row, col):
        """Check if placing a queen at board[row][col] is safe."""
        for i in range(col):
            if board[row][i] == 1:  # Check this row on the left
                return False

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

        for i, j in zip(range(row, self.size), range(col, -1, -1)):  # Check lower diagonal
            if board[i][j] == 1:
                return False

        return True
```

```
def alpha_beta_search(self, board, col, alpha, beta, maximizing_player):
    """Alpha-Beta Pruning Search."""
    if col >= self.size:  # If all queens are placed
        return 0, [row[:] for row in board]  # Return 0 as heuristic since it's a valid solution

    if maximizing_player:
        max_eval = float('-inf')
        best_board = None
        for row in range(self.size):
            if self.is_safe(board, row, col):
                board[row][col] = 1
                eval_score, potential_board = self.alpha_beta_search(board, col + 1, alpha, beta, False)
                board[row][col] = 0
                if eval_score > max_eval:
                    max_eval = eval_score
                    best_board = potential_board
                alpha = max(alpha, eval_score)
                if beta <= alpha:  # Beta cutoff
                    break
        return max_eval, best_board
    else:
        min_eval = float('inf')
        best_board = None
        for row in range(self.size):
            if self.is_safe(board, row, col):
                board[row][col] = 1
                eval_score, potential_board = self.alpha_beta_search(board, col + 1, alpha, beta, True)
                board[row][col] = 0
                if eval_score < min_eval:
                    min_eval = eval_score
                    best_board = potential_board
```

```
                beta = min(beta, eval_score)
                if beta <= alpha:  # Alpha cutoff
                    break
            return min_eval, best_board

    def solve(self):
        """Solve the 8-Queens problem."""
        board = [[0] * self.size for _ in range(self.size)]
        _, solution = self.alpha_beta_search(board, 0, float('-inf'), float('inf'), True)
        return solution

    def print_board(self, board):
        """Print the chessboard."""
        for row in board:
            print(" ".join("Q" if col else "." for col in row))
        print()


if __name__ == "__main__":
    game = EightQueens()
    solution = game.solve()
    if solution:
        print("Solution found:")
        game.print_board(solution)
    else:
        print("No solution exists.")
```

Output:


Solution found:
. Q . . . . . .
. . . . Q . . .
. . . . . . Q .
Q . . . .
. . Q . . . . .
. . . . Q
. . . . . Q . .
. . . Q . . . .