VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

S Gajanana Nayak(1BM22CS227)

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 **S Gajanana Nayak (1BM22CS227),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

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

Index

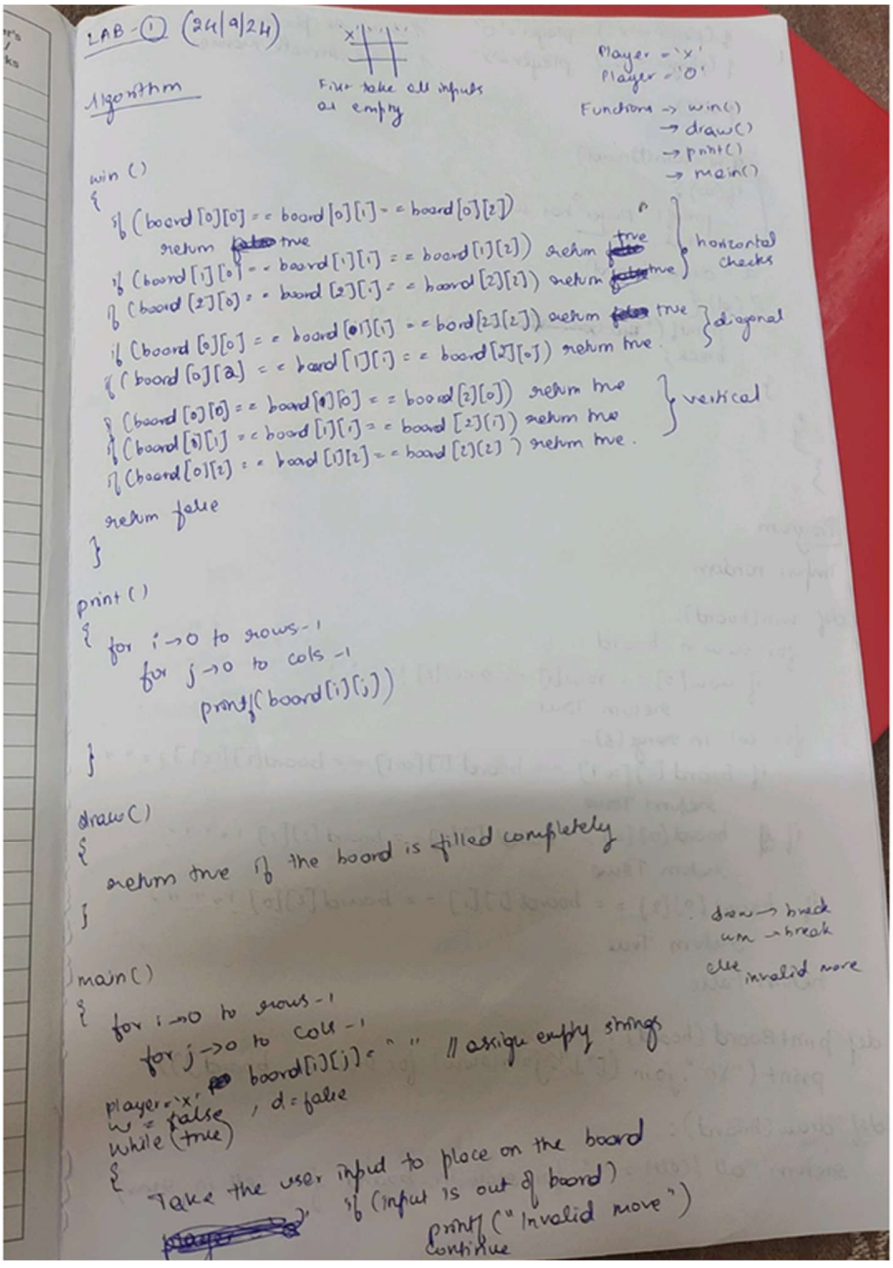
Github Link:

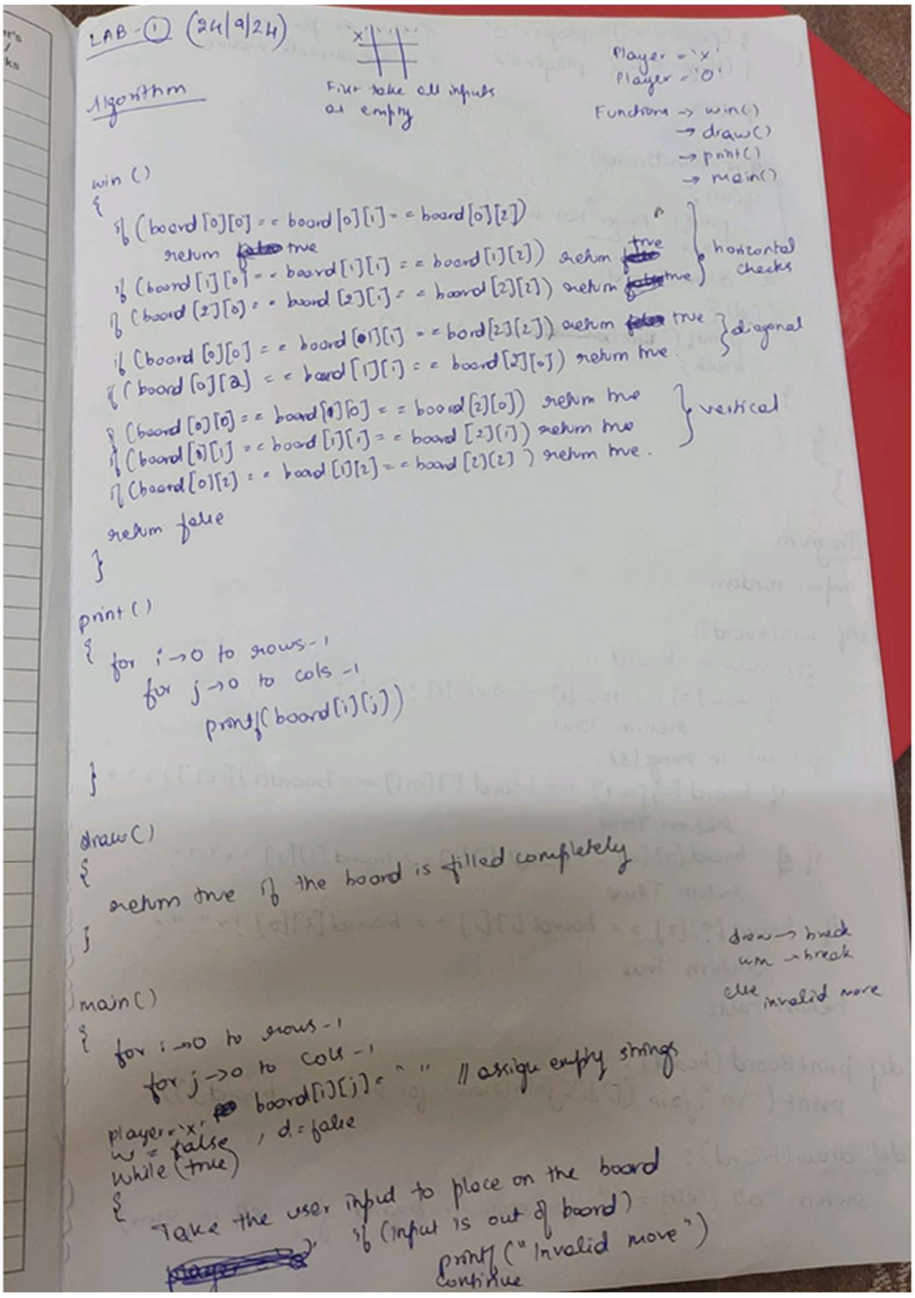
https://github.com/Gajanana227/AI

Program 1

Implement Tic –Tac –Toe Game

Algorithm:




LAB - 1 (24/9/24)

Algorithm

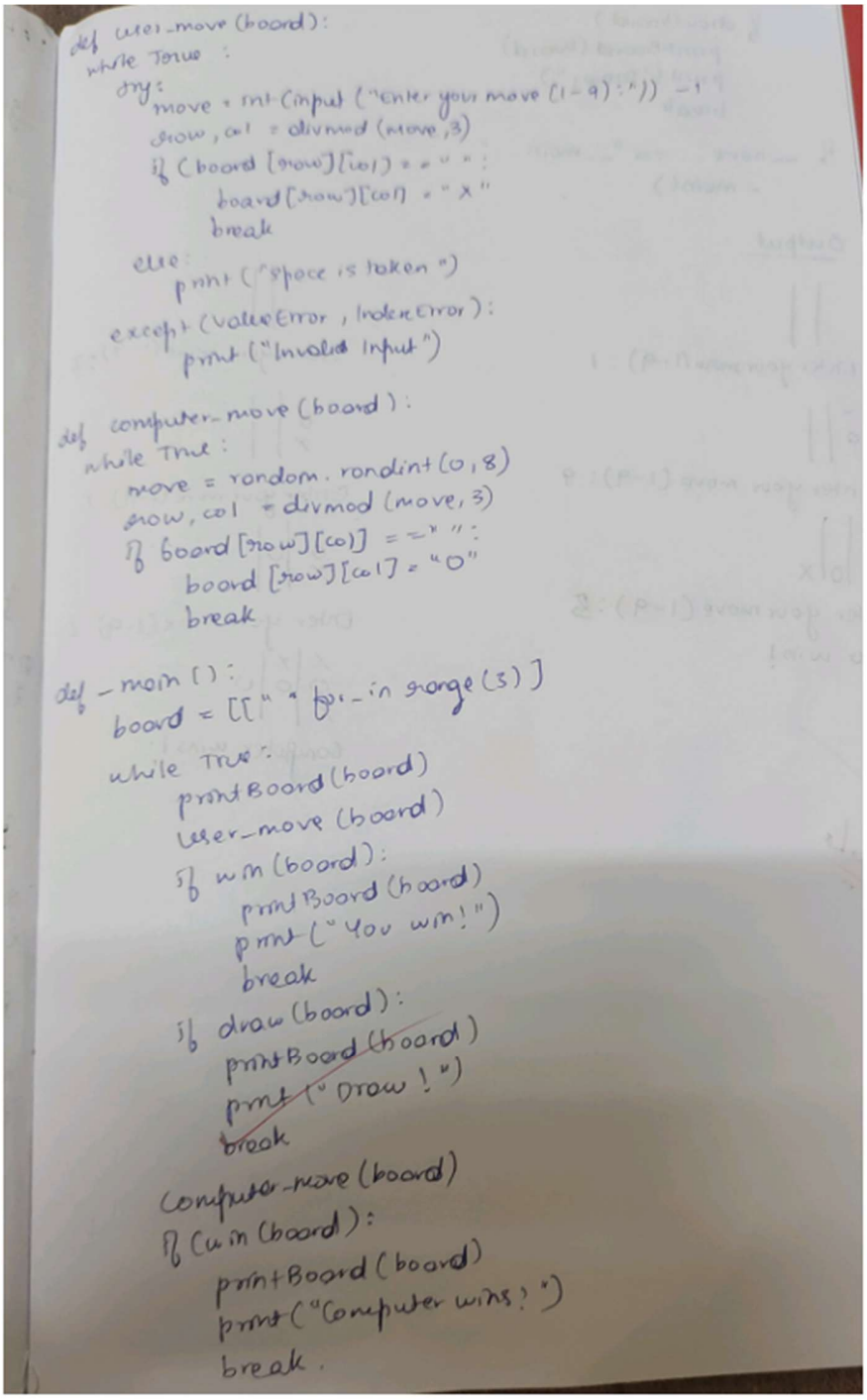
First take all inputs as empty

Player = 'X'
Player = 'O'

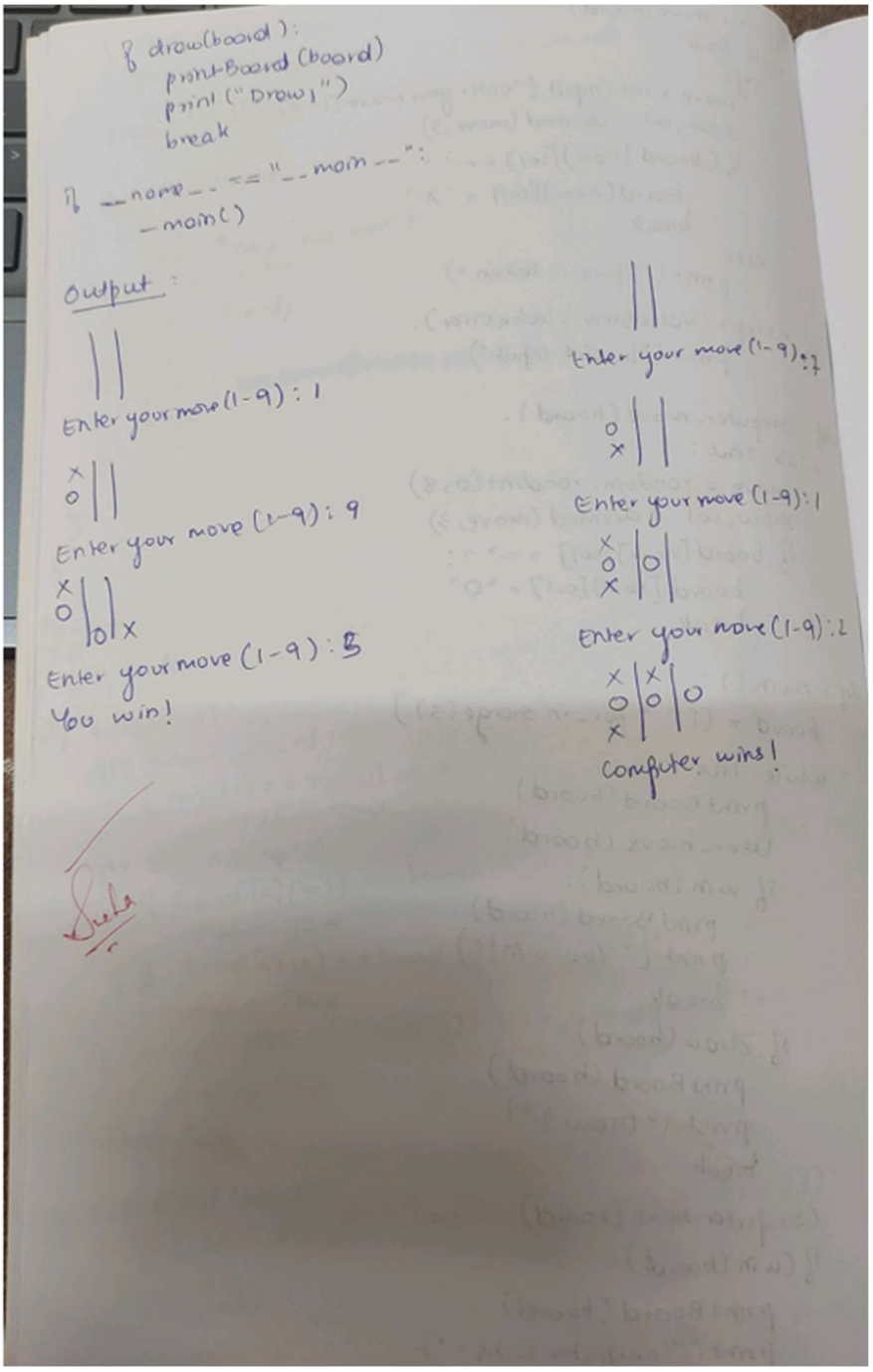
Functions → win()
→ draw()
→ print()
→ main()

```
win()
{
  if (board[0][0] == board[0][1] == board[0][2])
      return true
  if (board[1][0] == board[1][1] == board[1][2]) return true   } horizontal checks
  if (board[2][0] == board[2][1] == board[2][2]) return true
  if (board[0][0] == board[1][1] == board[2][2]) return true   } diagonal
  if (board[0][2] == board[1][1] == board[2][0]) return true.
  if (board[0][0] == board[1][0] == board[2][0]) return true   } vertical
  if (board[0][1] == board[1][1] == board[2][1]) return true
  if (board[0][2] == board[1][2] == board[2][2]) return true.
  return false
}

print()
{
  for i -> 0 to rows-1
      for j -> 0 to cols-1
          printf(board[i][j])
}

draw()
{
  return true if the board is filled completely
}

main()
{
  for i -> 0 to rows-1
      for j -> 0 to cols-1
          board[i][j] = " "  // assign empty strings
  player = 'X'
  w = false , d = false
  while (true)
  {
    Take the user input to place on the board
    if (input is out of board)
        printf("Invalid move")
        continue
```

draw → break
win → break
else invalid move



```
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("Space is taken")
    except (ValueError, IndexError):
        print("Invalid Input")

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)]

    while True:
        printBoard(board)
        user_move(board)
        if win(board):
            printBoard(board)
            print("You win!")
            break
        if draw(board):
            printBoard(board)
            print("Draw!")
            break
        computer_move(board)
        if win(board):
            printBoard(board)
            print("Computer wins!")
            break.
```



```
    draw(board):
        print_Board(board)
        print("Draw!")
        break

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

Output:

```
  |  |
```

Enter your move (1-9): 1

```
X |  |
O |  |
```

Enter your move (1-9): 9

```
X |  |
O | O| X
```

Enter your move (1-9): 5

You win!

```
  |  |
```

Enter your move (1-9): 7

```
O |  |
X |  |
```

Enter your move (1-9): 1

```
X |  |
O | O|
X |  |
```

Enter your move (1-9): 2

```
X | X|
O | O| O
X |  |
```

Computer wins!

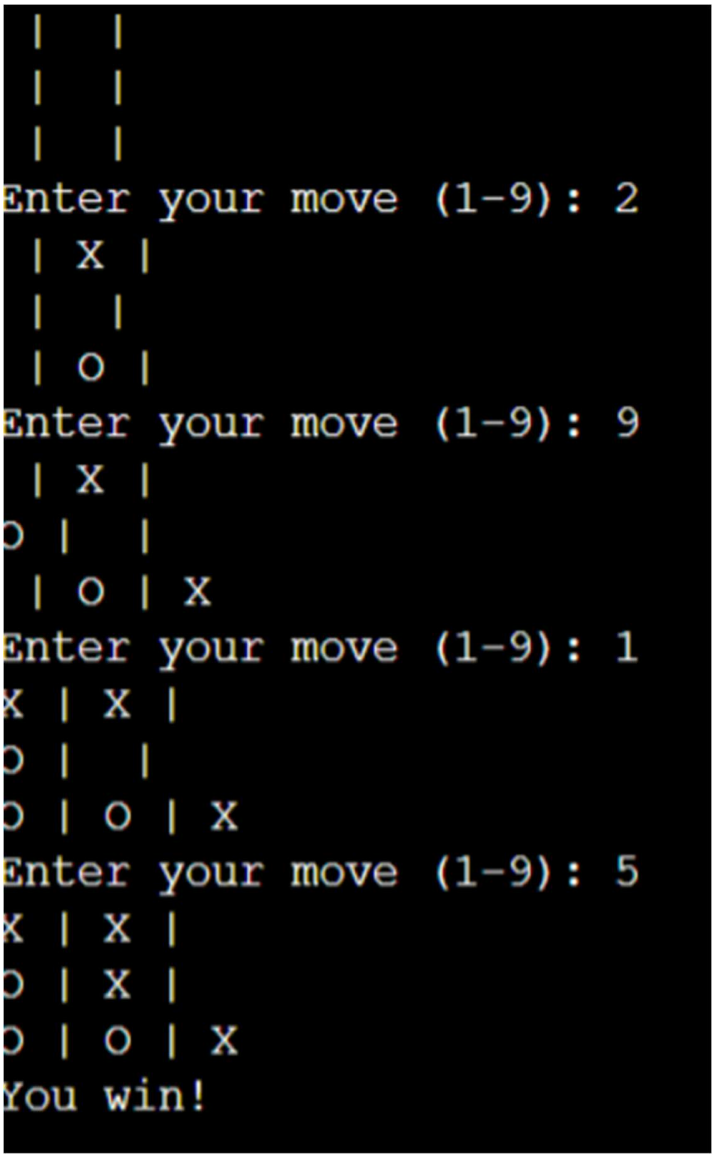
Code:

```
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()
```

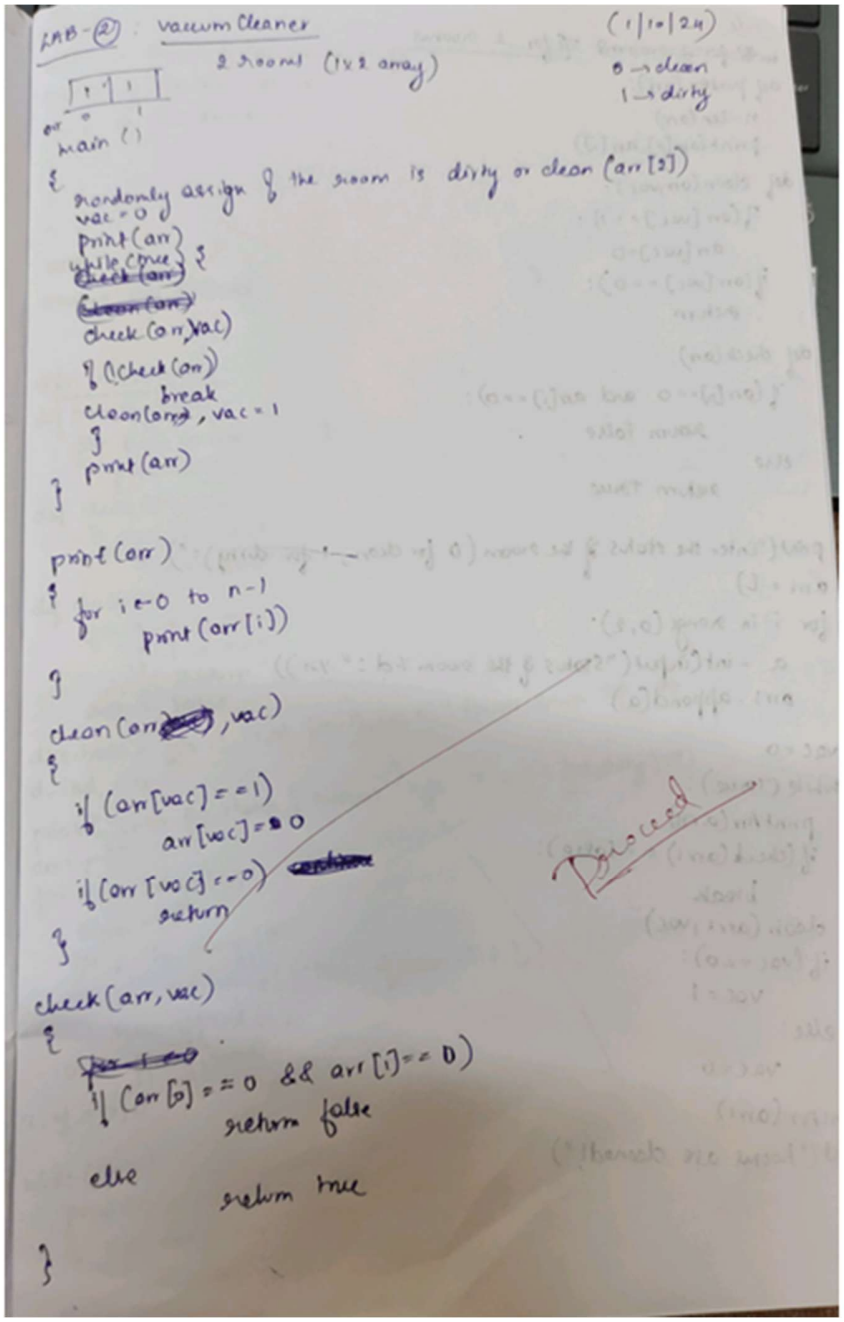
Output:

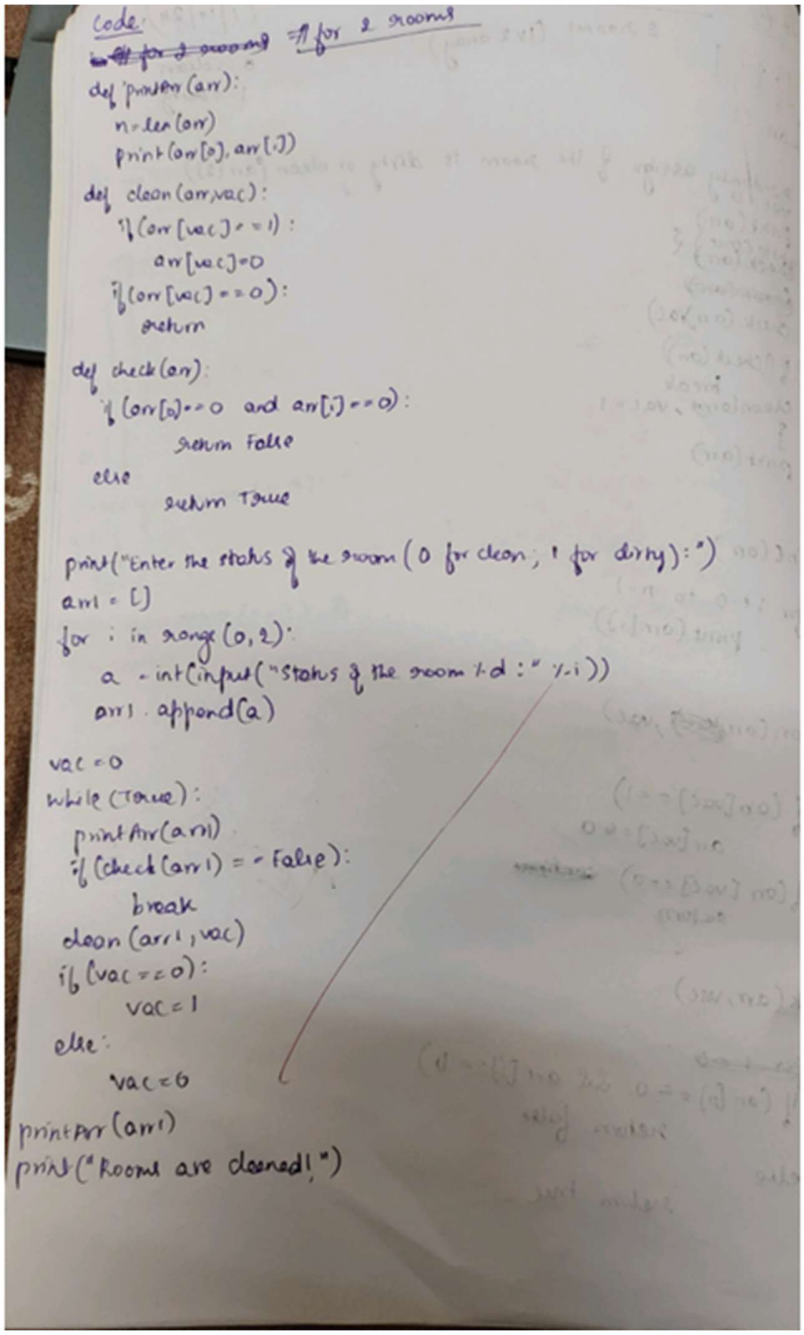


Program 2

Implement vacuum cleaner agent

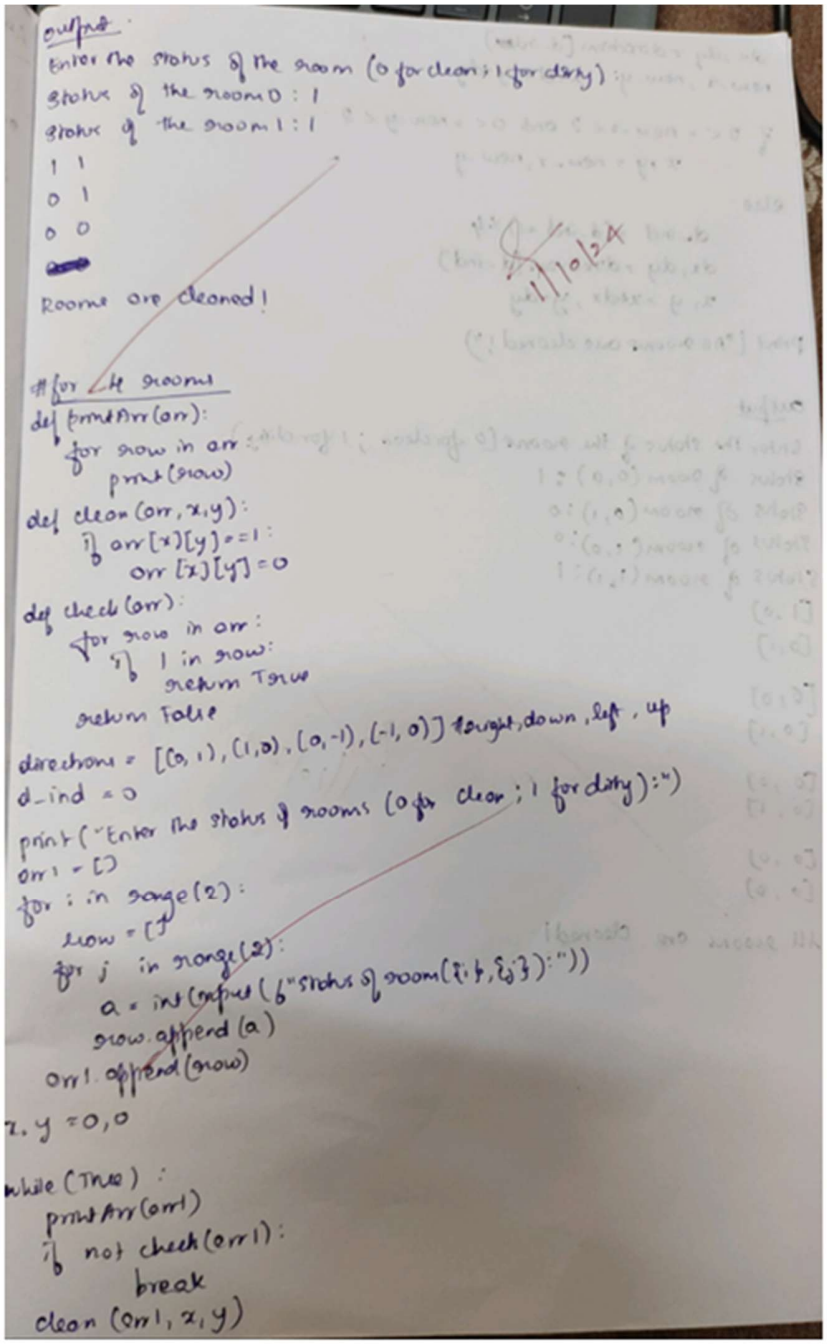
Algorithm:





Code:

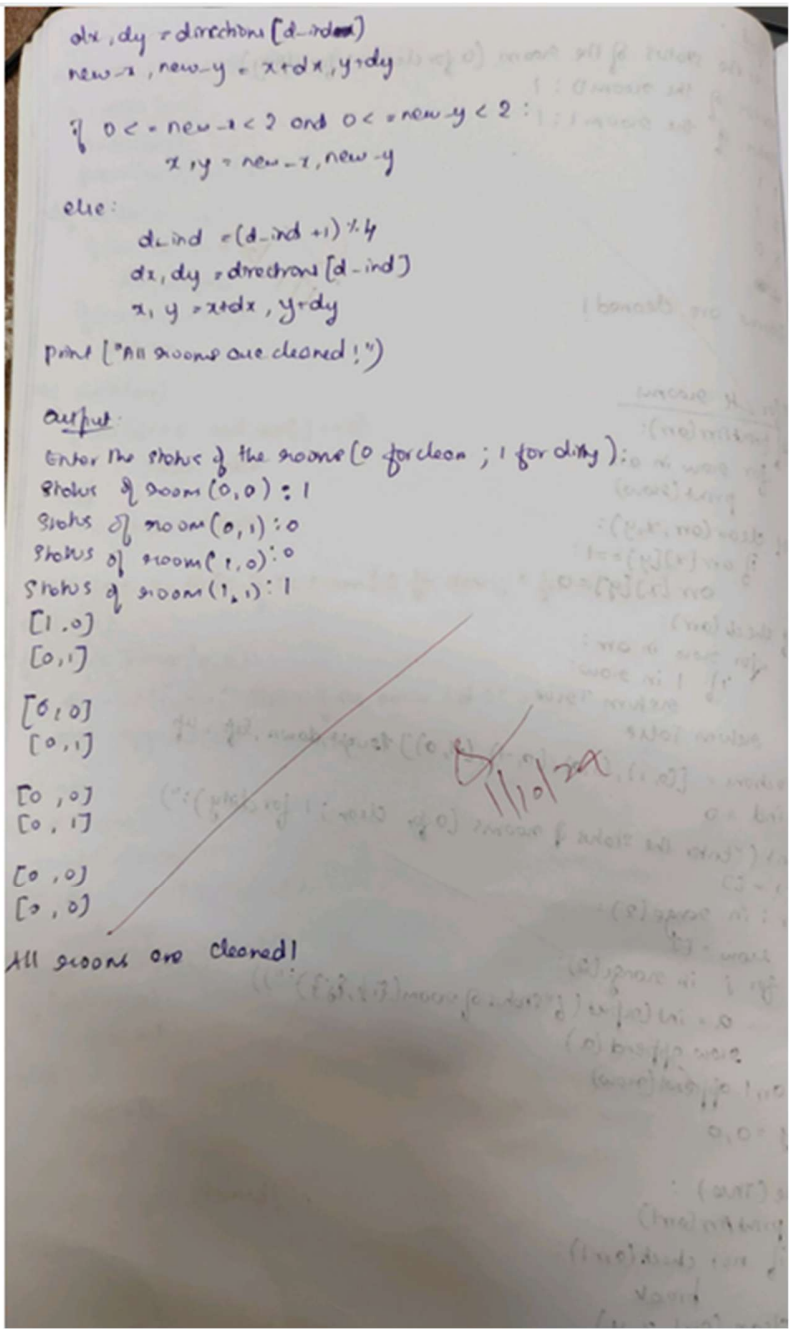
```
# for 2 rooms
def printArr(arr):
    n = len(arr)
    print(arr[0], arr[1])
def clean(arr,vac):
    if(arr[vac] == 1):
        arr[vac]=0
    if(arr[vac] == 0):
        return
def check(arr):
    if(arr[0]== 0 and arr[1]==0):
        return False
    else
        return True

print("Enter the status of the room (0 for clean, 1 for dirty):")
arr1 = []
for i in range(0,2):
    a = int(input("Status of the room %d :" %i))
    arr1.append(a)

vac=0
while(True):
    printArr(arr1)
    if(check(arr1) == False):
        break
    clean(arr1,vac)
    if(vac==0):
        vac=1
    else:
        vac=0
printArr(arr1)
print("Rooms are cleaned!")
```



output:

```
Enter the status of the room (0 for clean; 1 for dirty):
status of the room 0 : 1
status of the room 1 : 1
1 1
0 1
0 0
Rooms are cleaned!
```

```
# for 4 rooms
def printArr(arr):
    for row in arr:
        print(row)
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 = [(0, 1), (1,0), (0, -1), (-1, 0)] #right, down, left, up
d_ind = 0
print("Enter the status of rooms (0 for clear ; 1 for dirty):")
arr1 = []
for i in range(2):
    row = []
    for j in range(2):
        a = int(input(f"status of room[{i+1}, {j+1}]:"))
        row.append(a)
    arr1.append(row)
x, y = 0,0

while(True):
    printArr(arr1)
    if not check(arr1):
        break
    clean(arr1, x, y)
```



```
dx, dy = directions[d_ind]
new_x, new_y = x+dx, y+dy

if 0 <= new_x < 2 and 0 <= new_y < 2:
        x, y = new_x, new_y

else:
        d_ind = (d_ind + 1) % 4
        dx, dy = directions[d_ind]
        x, y = x+dx, y+dy

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) : 0
Status of room (1,1) : 1
[1,0]
[0,1]

[0,0]
[0,1]

[0,0]
[0,1]

[0,0]
[0,0]
All rooms are cleaned!
```

11/10/24

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!")
```

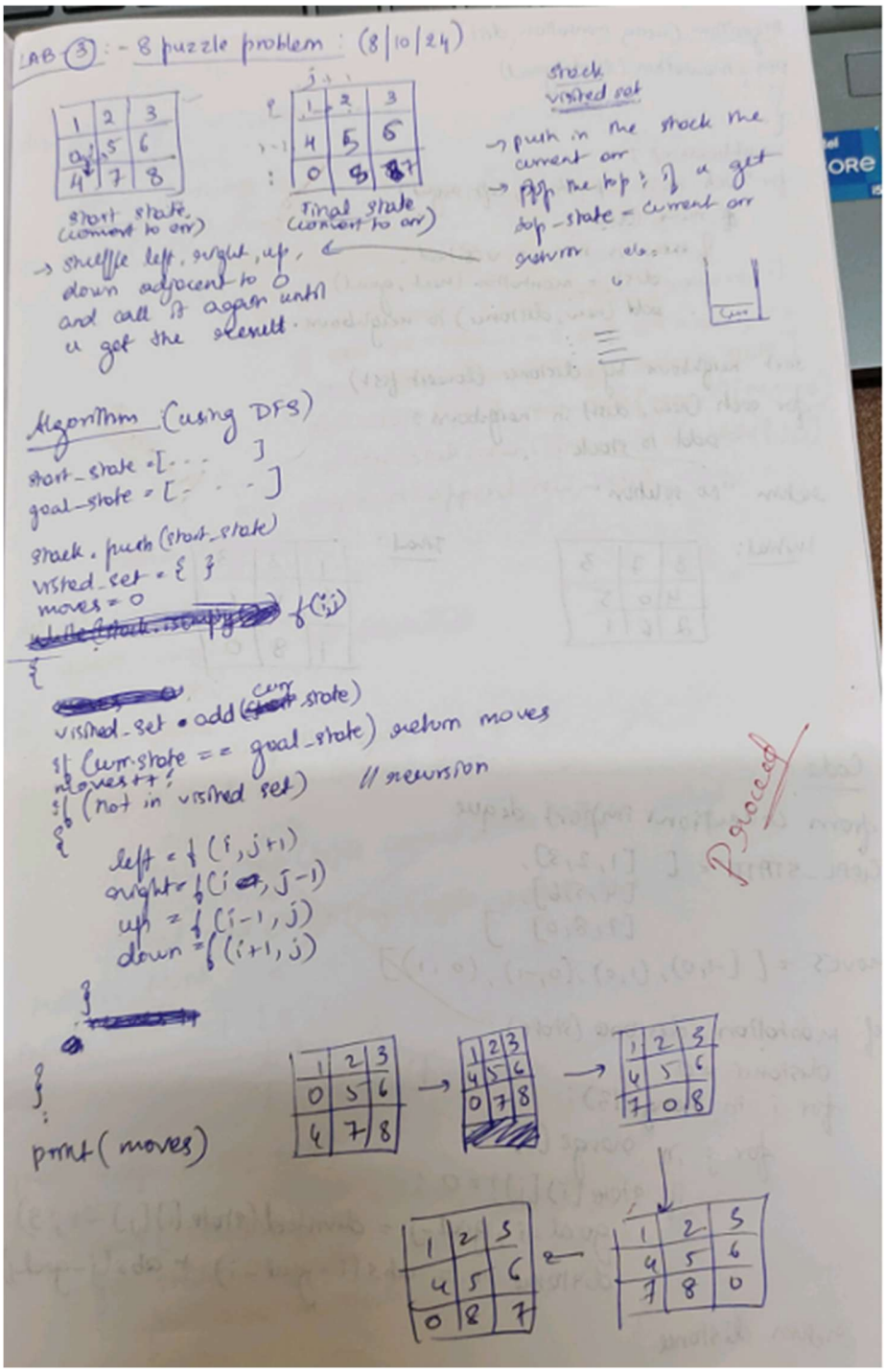
Output:

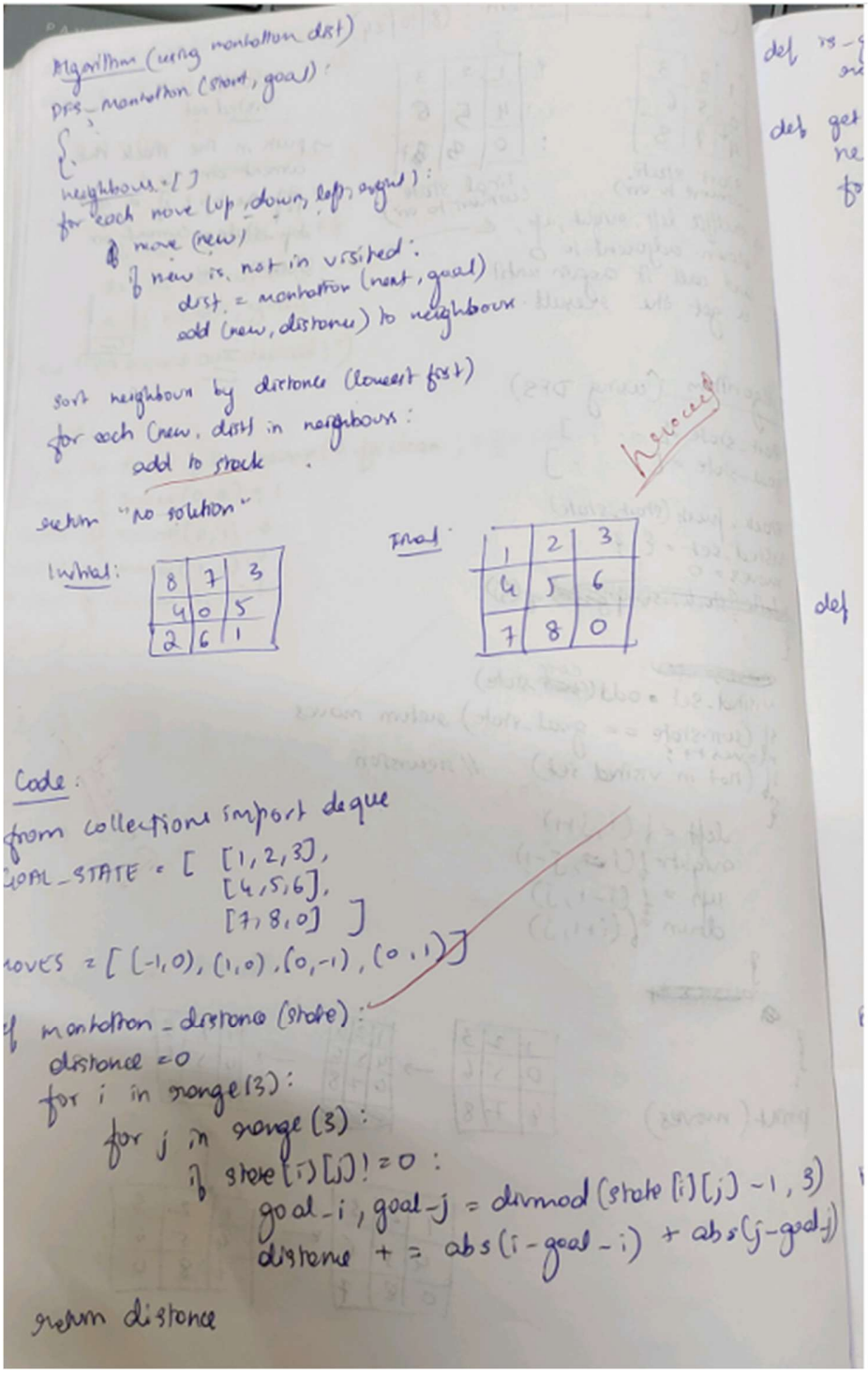
```
Enter the status of the rooms (0 for clean; 1 for dirty):
Status of room (0, 0): 1
Status of room (0, 1): 0
Status of room (1, 0): 1
Status of room (1, 1): 0
[1, 0]
[1, 0]

[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:





Algorithm (using manhattan dist)

DFS_manhattan (start, goal):

{

neighbours = [ ]

for each move (up, down, left, right):

move (new)

if new is not in visited:

dist = manhattan (next, goal)

add (new, distance) to neighbour

sort neighbour by distance (lowest first)

for each (new, dist) in neighbours:

add to stack .

success

return "no solution"

Initial:

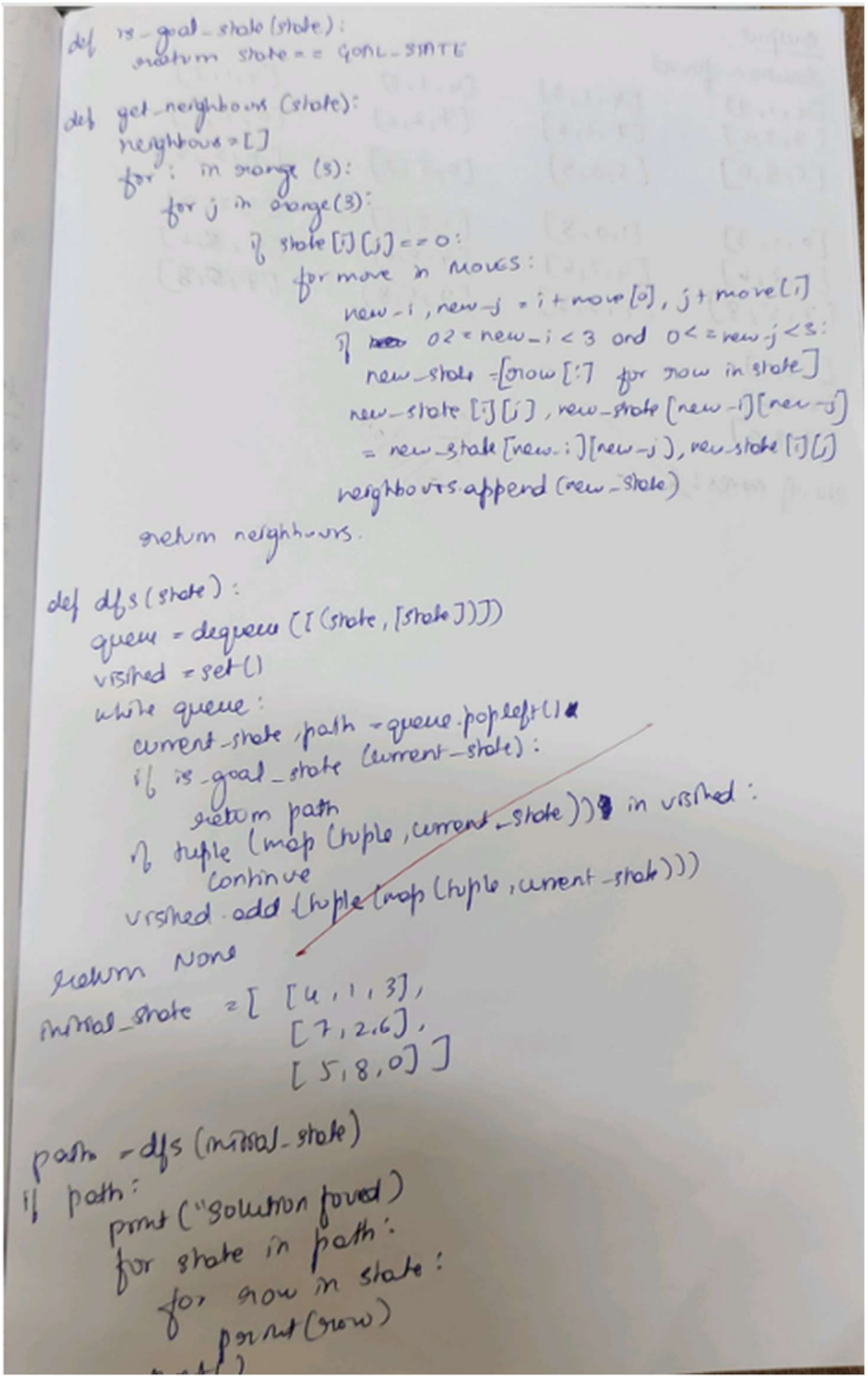
8	7	3
4	0	5
2	6	1

Final:

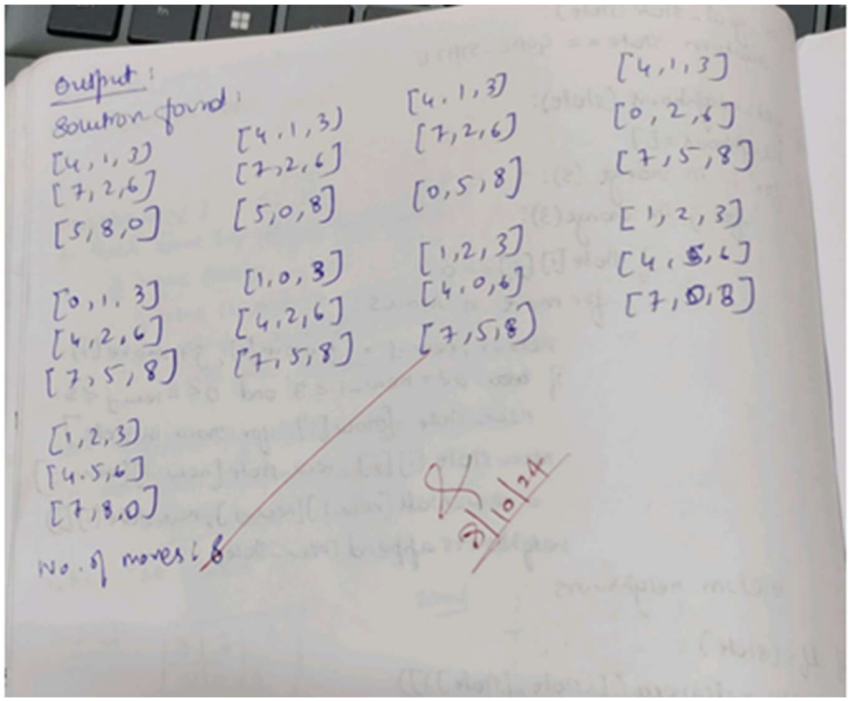
1	2	3
4	5	6
7	8	0

Code:

```
from collections import deque
GOAL_STATE = [ [1,2,3],
               [4,5,6],
               [7,8,0] ]
MOVES = [ [-1,0), (1,0), (0,-1), (0,1)]

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

return distance
```



```
def is_goal_state(state):
    return state == GOAL_STATE

def get_neighbours(state):
    neighbours = []
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                for move in moves:
                    new_i, new_j = i + move[0], j + move[1]
                    if 0 <= new_i < 3 and 0 <= new_j < 3:
                        new_state = [row[:] for row in state]
                        new_state[i][j], new_state[new_i][new_j]
                        = new_state[new_i][new_j], new_state[i][j]
                        neighbours.append(new_state)
    return neighbours

def dfs(state):
    queue = deque([(state, [state])])
    visited = set()
    while queue:
        current_state, path = queue.popleft()
        if is_goal_state(current_state):
            return path
        if tuple(map(tuple, current_state)) in visited:
            continue
        visited.add(tuple(map(tuple, current_state)))
    return None

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

path = dfs(initial_state)
if path:
    print("Solution found")
    for state in path:
        for row in state:
            print(row)
```



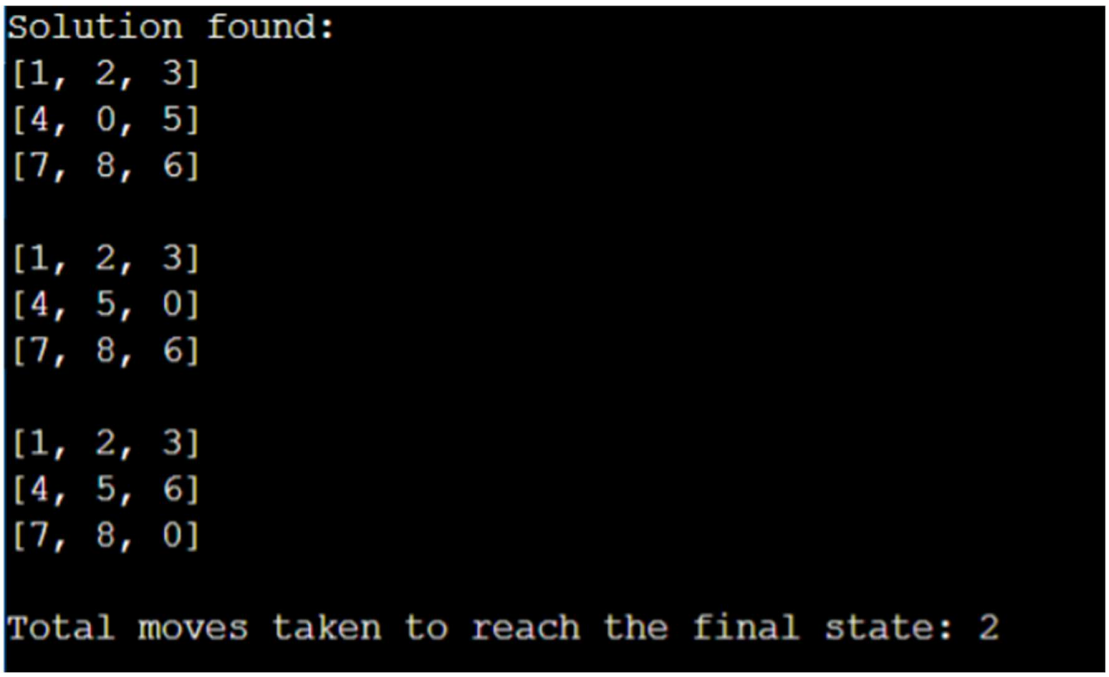
Code:

```
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.")
```

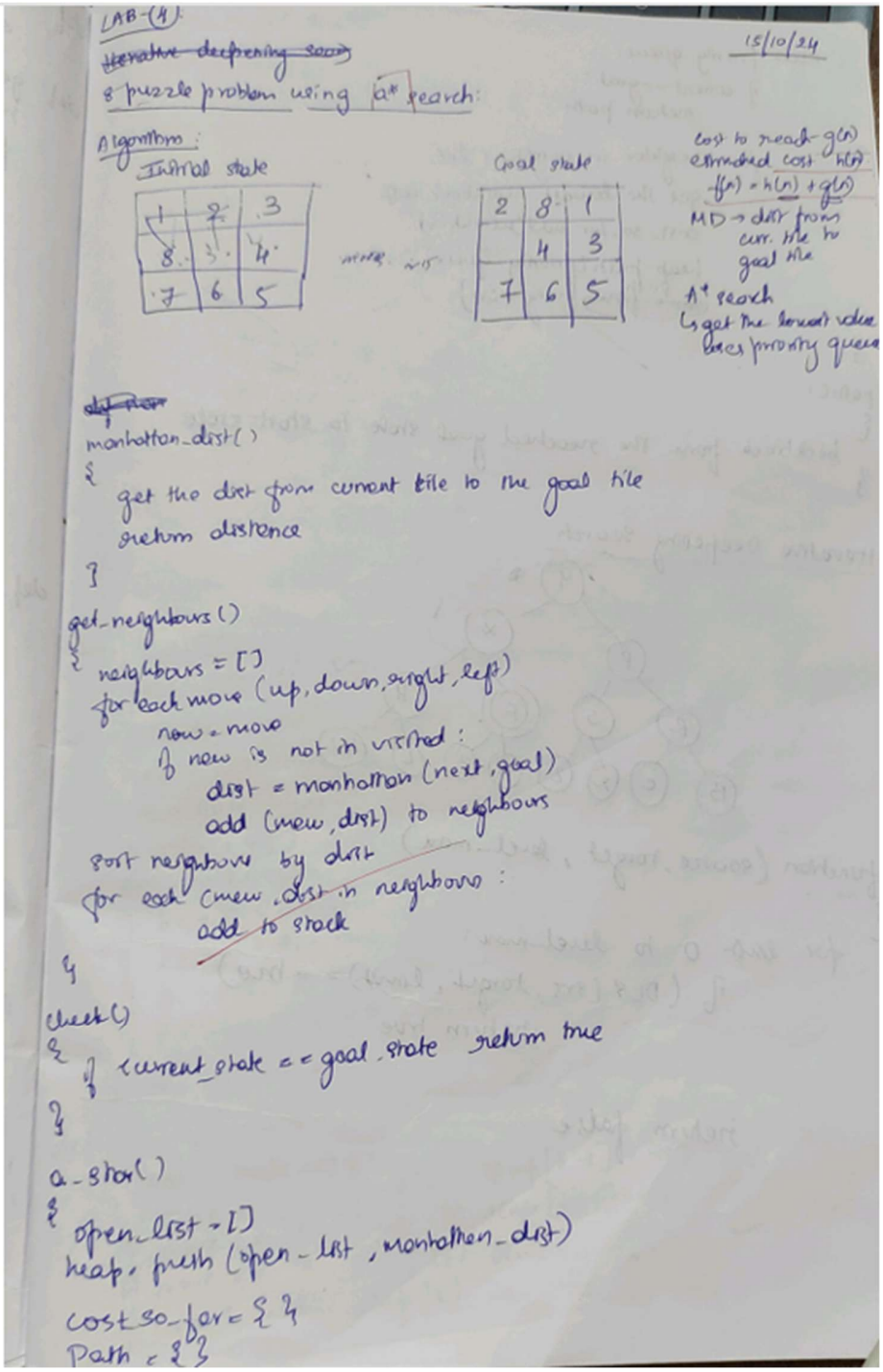
Output:

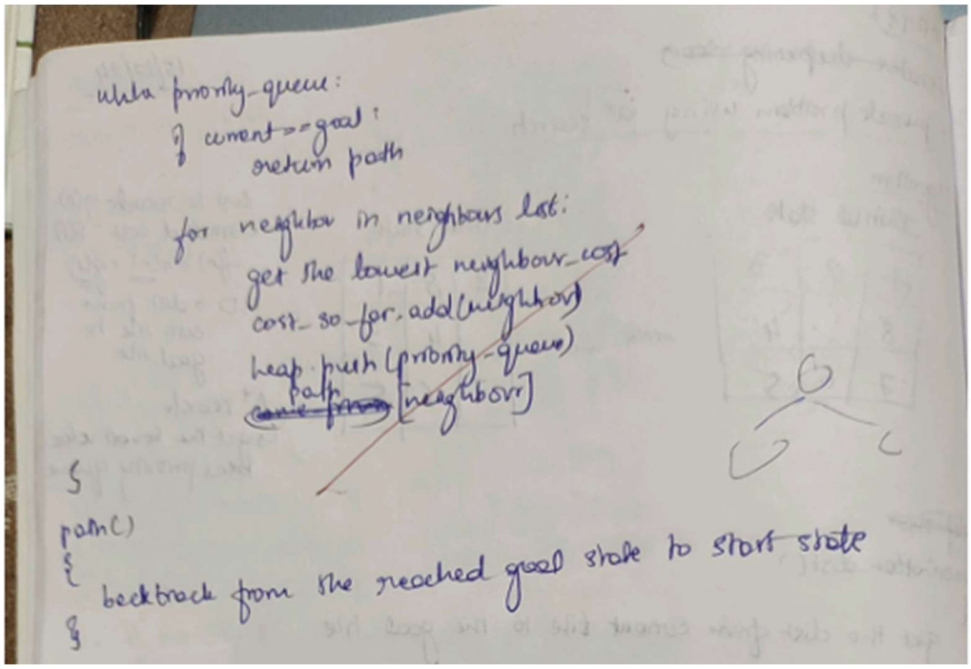

Solution found:
[1, 2, 3]
[4, 0, 5]
[7, 8, 6]
[1, 2, 3]
[4, 5, 0]
[7, 8, 6]
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Total moves taken to reach the final state: 2

Program 4

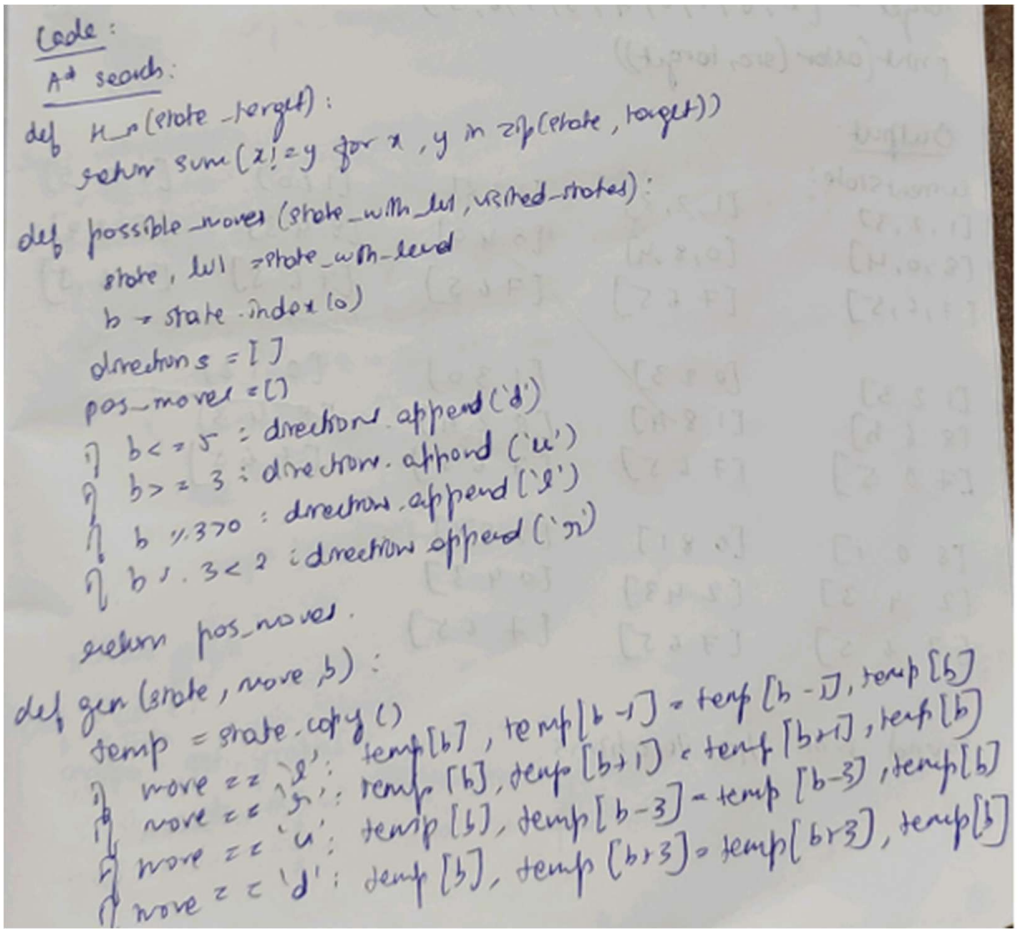
Implement A* search algorithm

Algorithm:





```
while priority_queue:
    if current == goal:
        return path

    for neighbor in neighbors list:
        get the lowest neighbour_cost
        cost_so_far.add(neighbor)
        heap.push(priority_queue)
        path [neighbor]

}

path()
{
    backtrack from the reached goal state to start state
}
```



Code:

A* search:

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

def possible_moves(state_with_lvl, visited_states):
    state, lvl = state_with_level
    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')
    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]
```

```
def astar(src, target):
    arr = [[src, 0]]
    visited_states = []
    th = 0
    while arr:
        then += 1
        current = min(arr, key = lambda x: f_n(x, target))
        arr.remove(current)
        if current[0] == target:
            return 'Found'
        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 = [1, 8, 4, 0, 2, 3, 7, 6, 5]
print(astar(src, target))
```

Output:

current state:

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

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

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

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

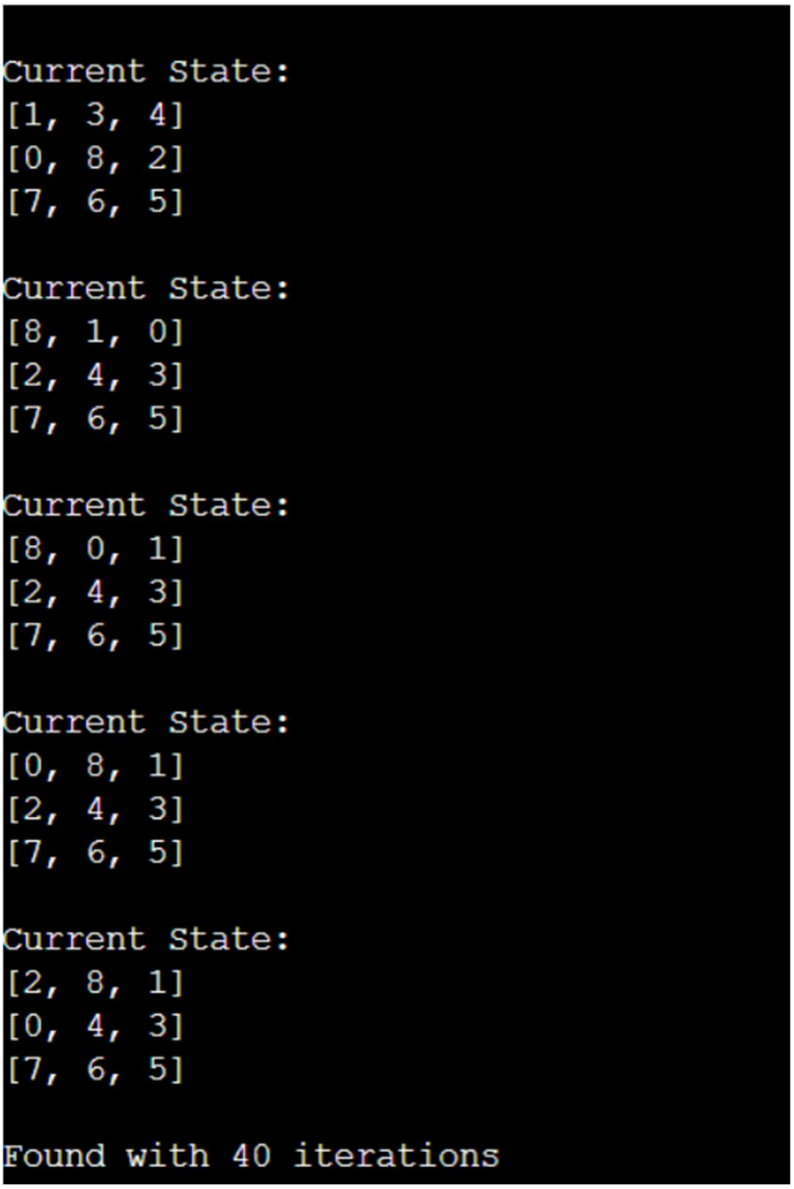
Found with 40 iterations.

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))
```

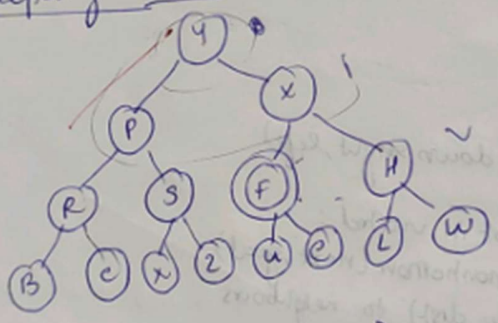
Output:



Implement Iterative deepening search algorithm

Algorithm:

Iterative Deepening Search

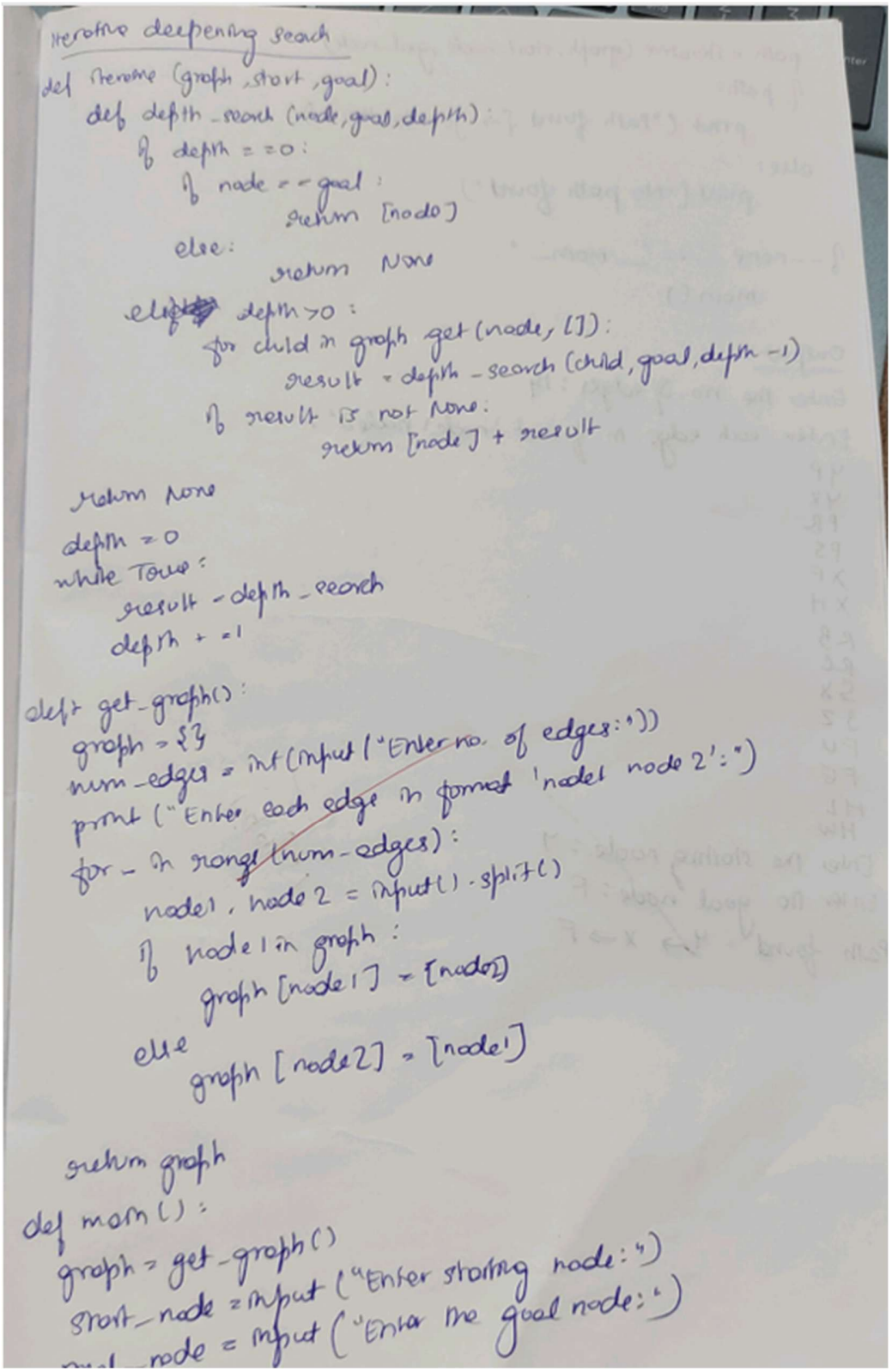



```
function (source, target, level_max)
{
    for limit 0 to level_max:
        if (DLS (src, target, limit) == true)
            return true

    return false
}
```

```
DLS (src, target, limit)
{ if (src == target)
        return true
  if (limit <= 0)
        return false
  for each i adjacent of src
        if DLS (i, target, limit - 1)
              return true

        return false
}
```

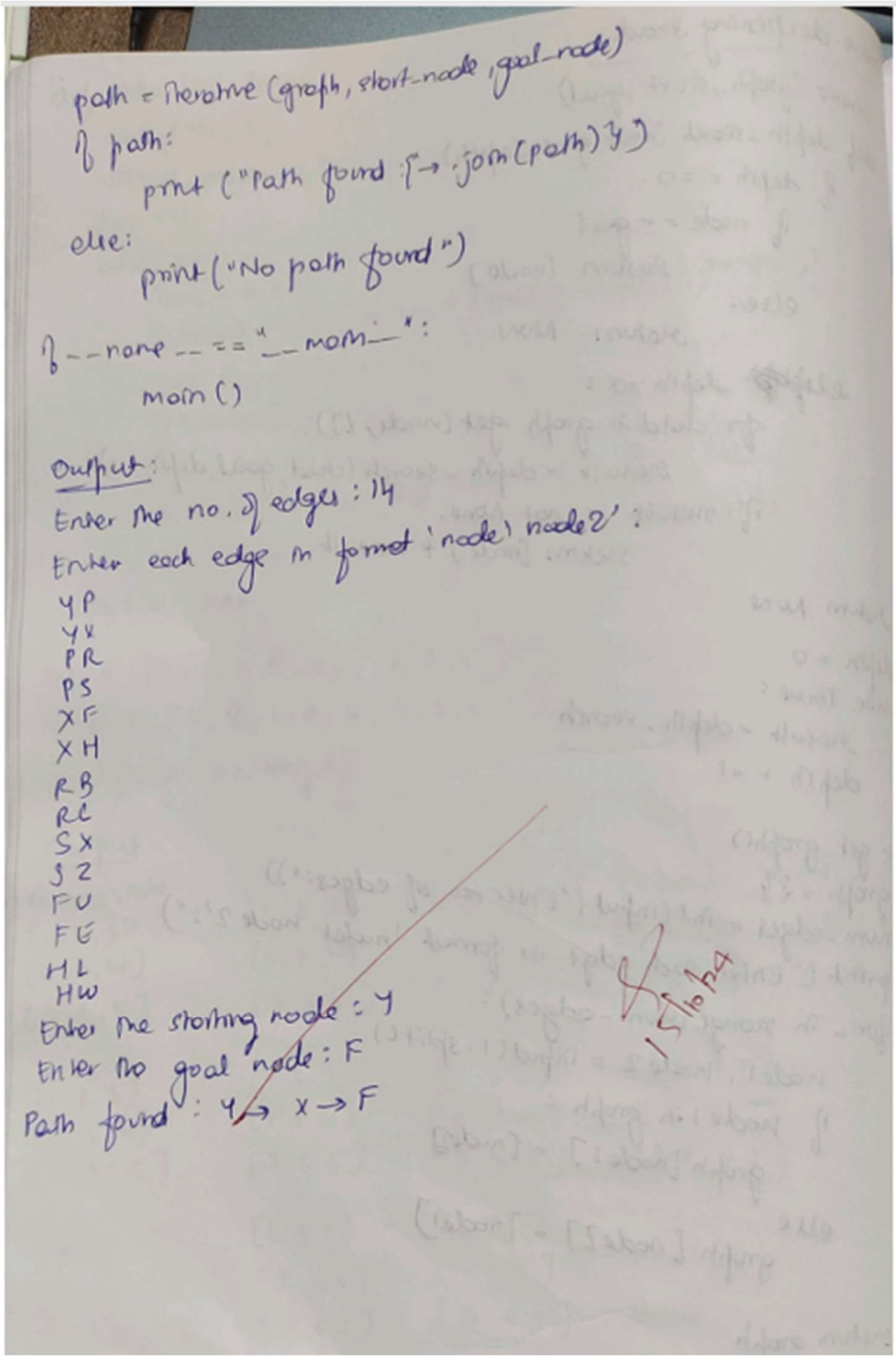
Proceed

Path



Iterative deepening search

```
def iterative (graph, start, goal):
    def depth_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_search (child, goal, depth-1)
                if result is not None:
                    return [node] + result
    return None
    depth = 0
    while True:
        result = depth_search
        depth += 1

def get_graph():
    graph = {}
    num_edges = int(input("Enter no. of edges:"))
    print("Enter each edge in format 'node1 node2':")
    for _ in range(num_edges):
        node1, node2 = input().split()
        if node1 in graph:
            graph[node1] = [node2]
        else
            graph[node2] = [node1]

    return graph
def main():
    graph = get_graph()
    start_node = input("Enter starting node:")
    goal_node = input("Enter the goal node:")
```



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()
```

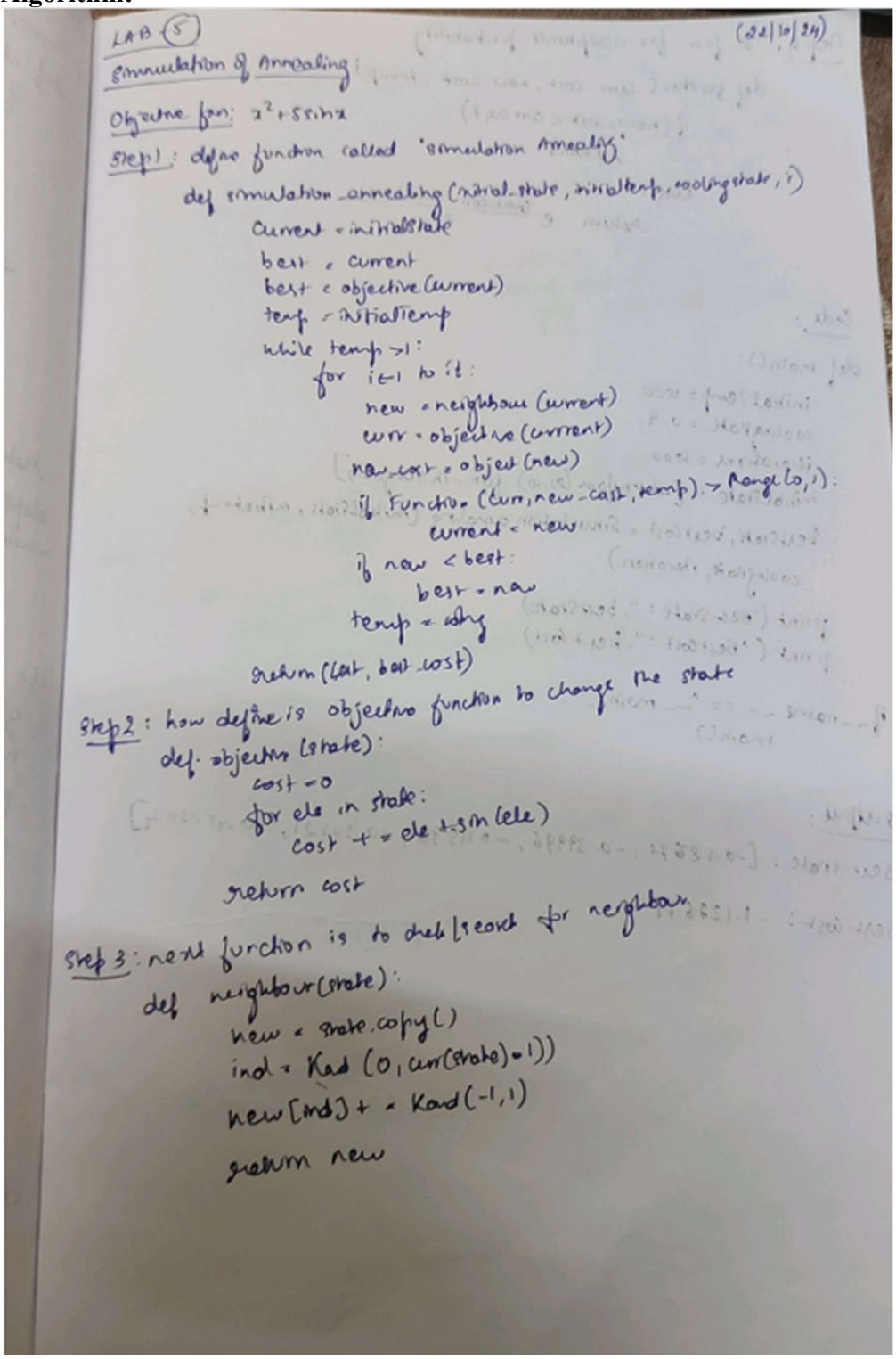
Output:

```
Enter the number of edges: 14
Enter each edge in the format 'node1 node2':
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 the starting node: Y
Enter the goal node: F
Path found: Y -> X -> F
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



LAB - 5 (22/10/24)

Simulation of Annealing:

Objective fun: $x^2 + 5\sin x$

Step 1: define function called 'simulation Annealing'

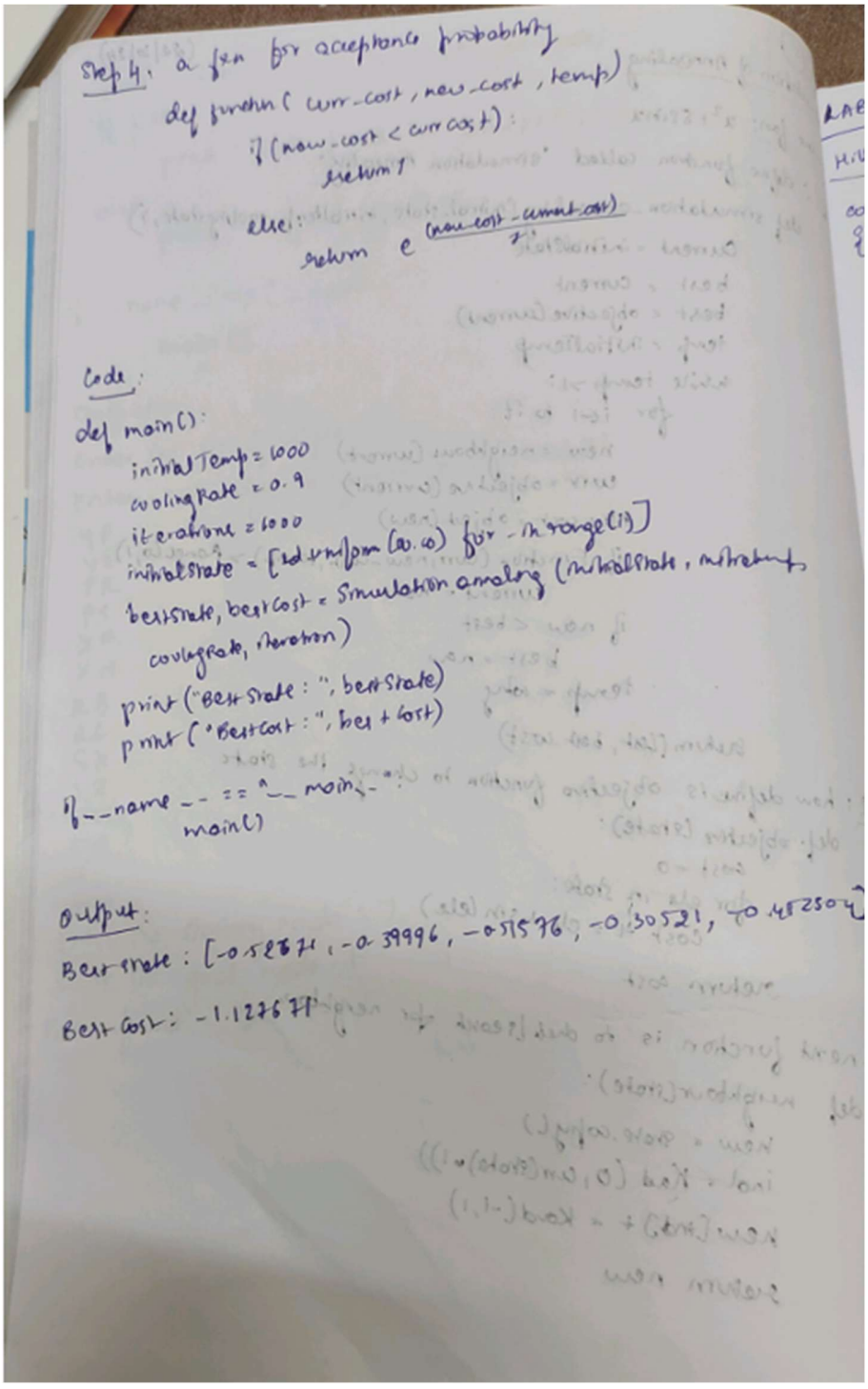
```
def simulation_annealing(initial_state, initialtemp, coolingstate, i)
    Current = initialstate
    best = current
    best = objective(current)
    temp = initialtemp
    while temp >1:
        for i=1 to it:
            new = neighbour(current)
            curr = objective(current)
            new_cost = obj(new)
            if Function(curr, new_cost, temp) > Rand(0,1):
                current = new
            if new < best:
                best = new
            temp = obj
    return(last, best_cost)
```

Step 2: how define is objective function to change the state

```
def objective(state):
    cost = 0
    for ele in state:
        cost += ele + sin(ele)
    return cost
```

Step 3: next function is to check [search for neighbour

```
def neighbour(state):
    new = state.copy()
    ind = Rand(0, len(state)-1)
    new[ind] += Rand(-1,1)
    return new
```



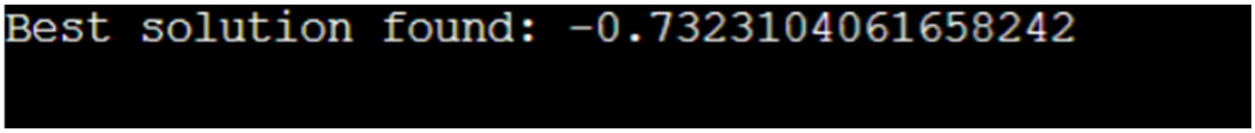
Code:

```
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}")
```

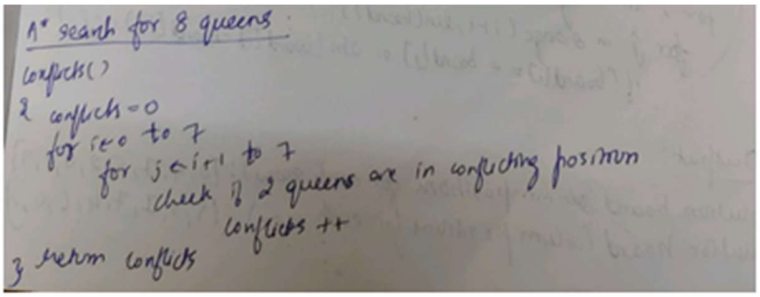
Output:

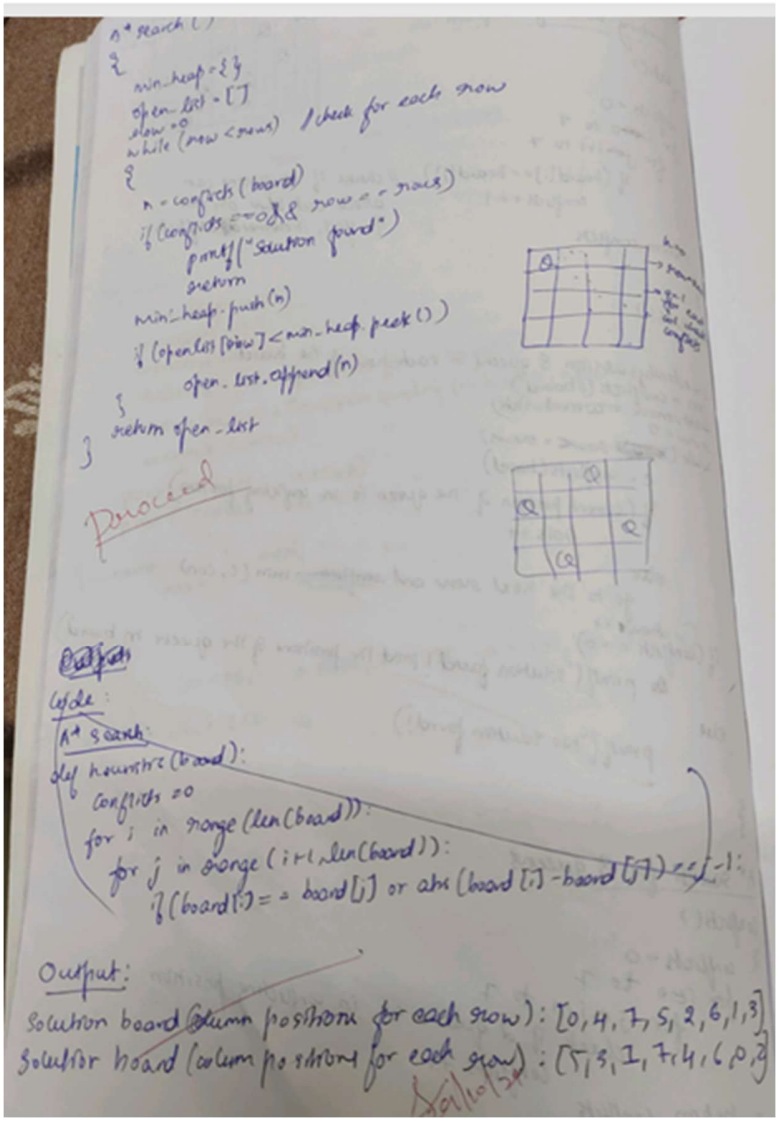


Program 6

Implement A* search algorithm for N queens

Algorithm:





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))
```

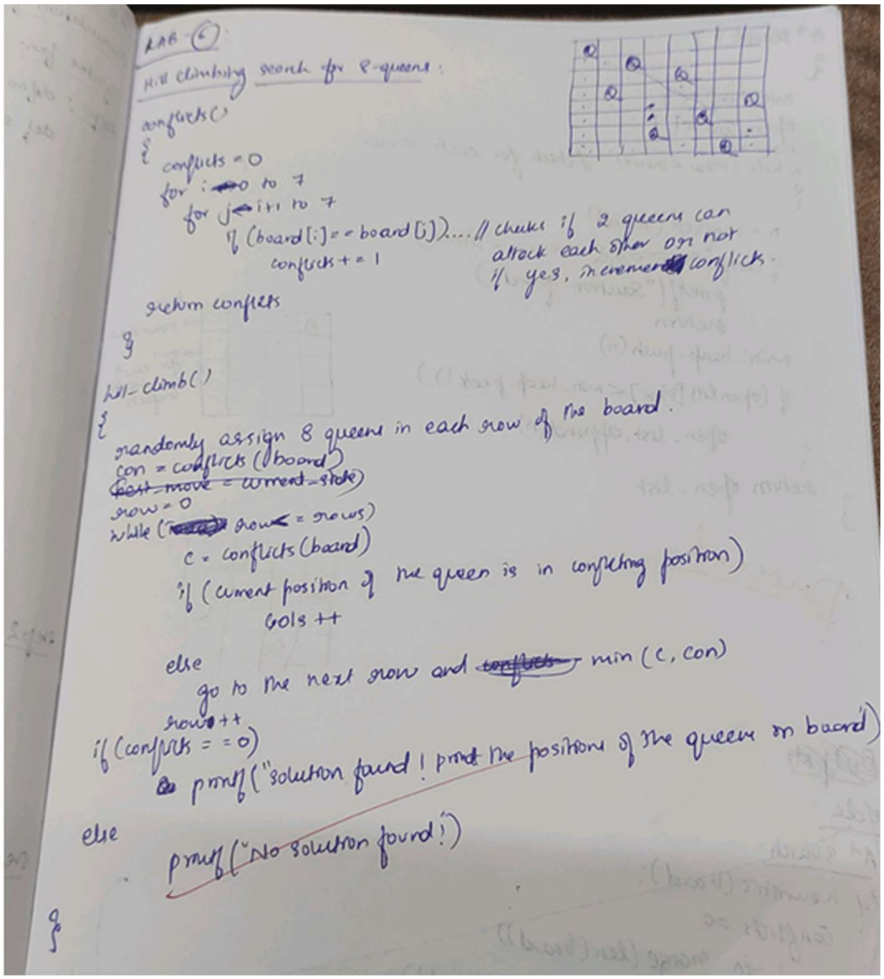
```
    return None  # No solution found

# 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)
```

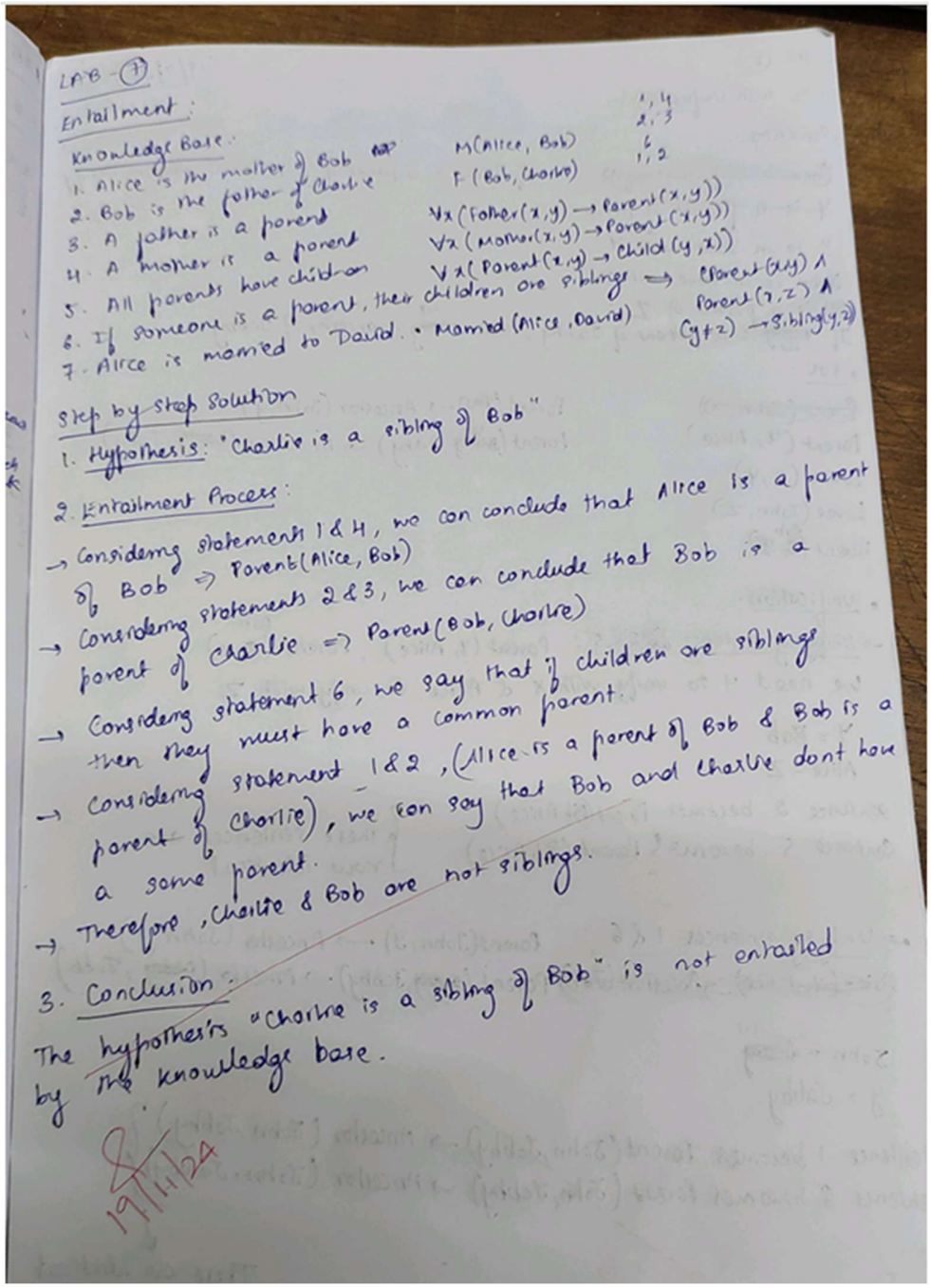
Output:

```
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:



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
```

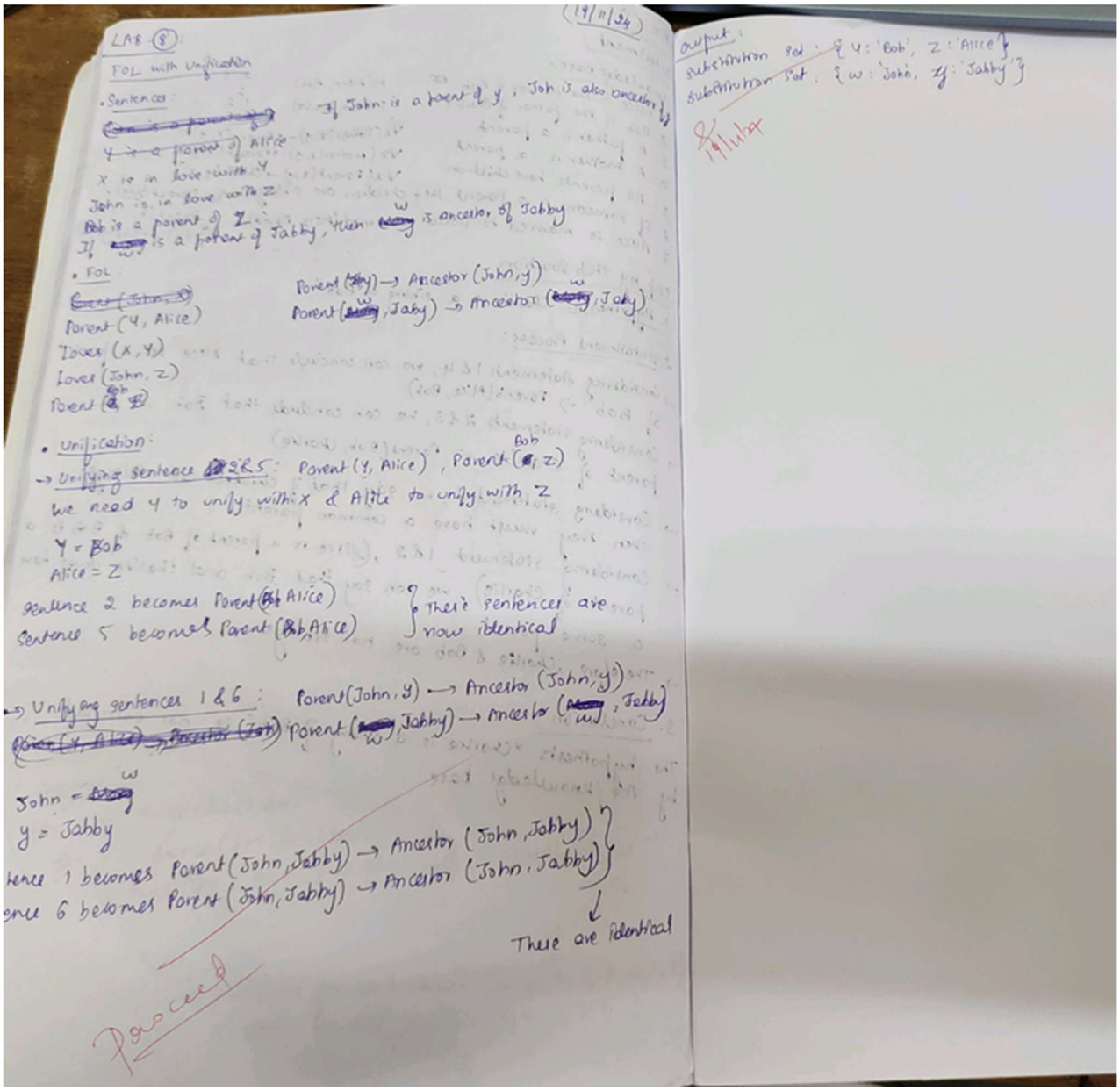
```
            new_h = heuristic(new_board)

            # 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:

Program 8

Implement unification in first order logic

Algorithm:



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))
```

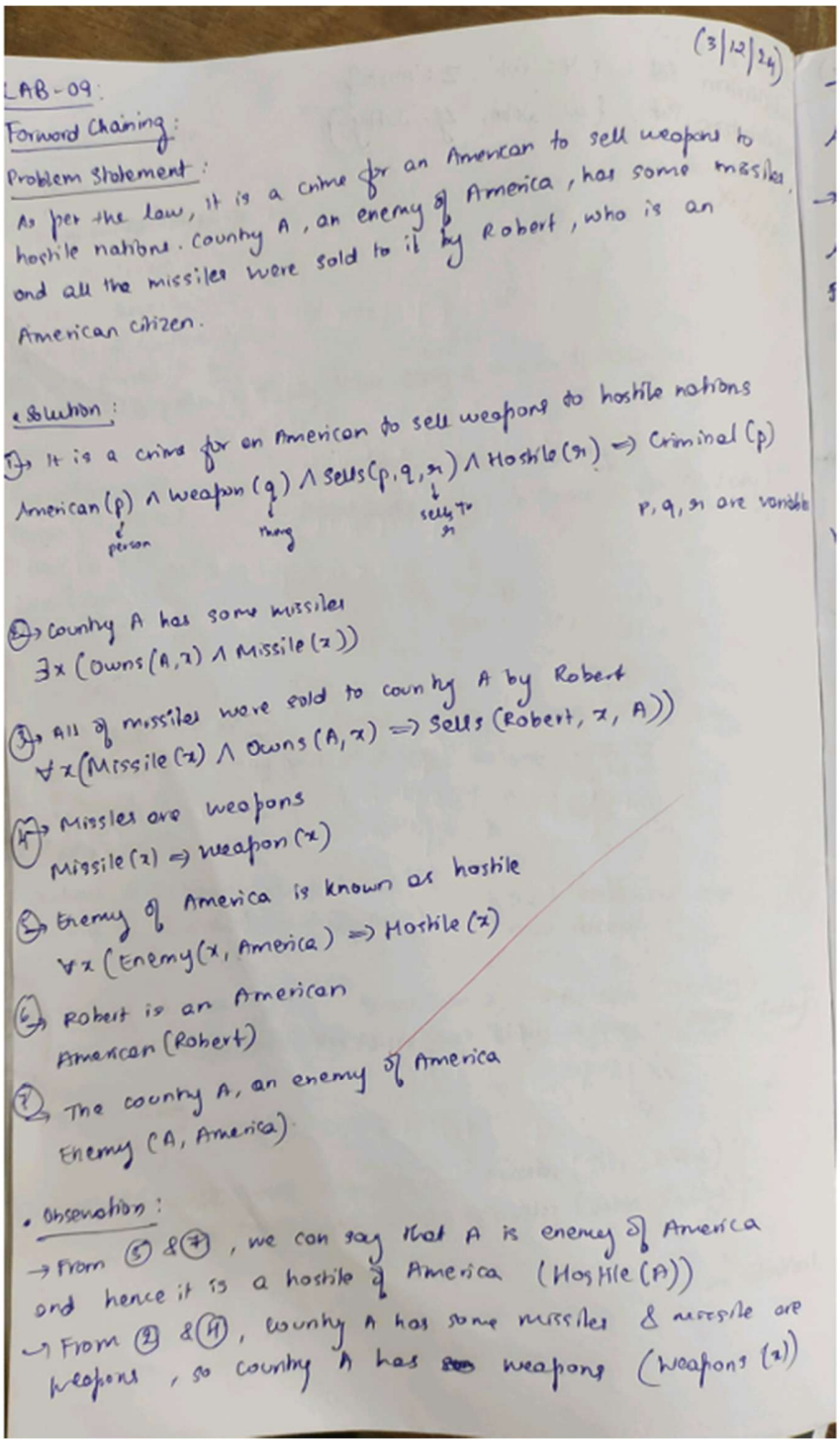
Output:

```
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:



(3/12/24)

LAB-09:

Forward Chaining:

Problem Statement:

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

• Solution:

1) It is a crime for an American to sell weapons to hostile nations

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

(p: person, q: thing, r: sells to)

p, q, r are variable

2) Country A has some missiles

∃x (Owns(A, x) ∧ Missile(x))

3) All of missiles were sold to country A by Robert

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

4) Missles are weapons

Missile(x) ⇒ weapon(x)

5) Enemy of America is known as hostile

∀x (Enemy(x, America) ⇒ Hostile(x))

6) Robert is an American

American(Robert)

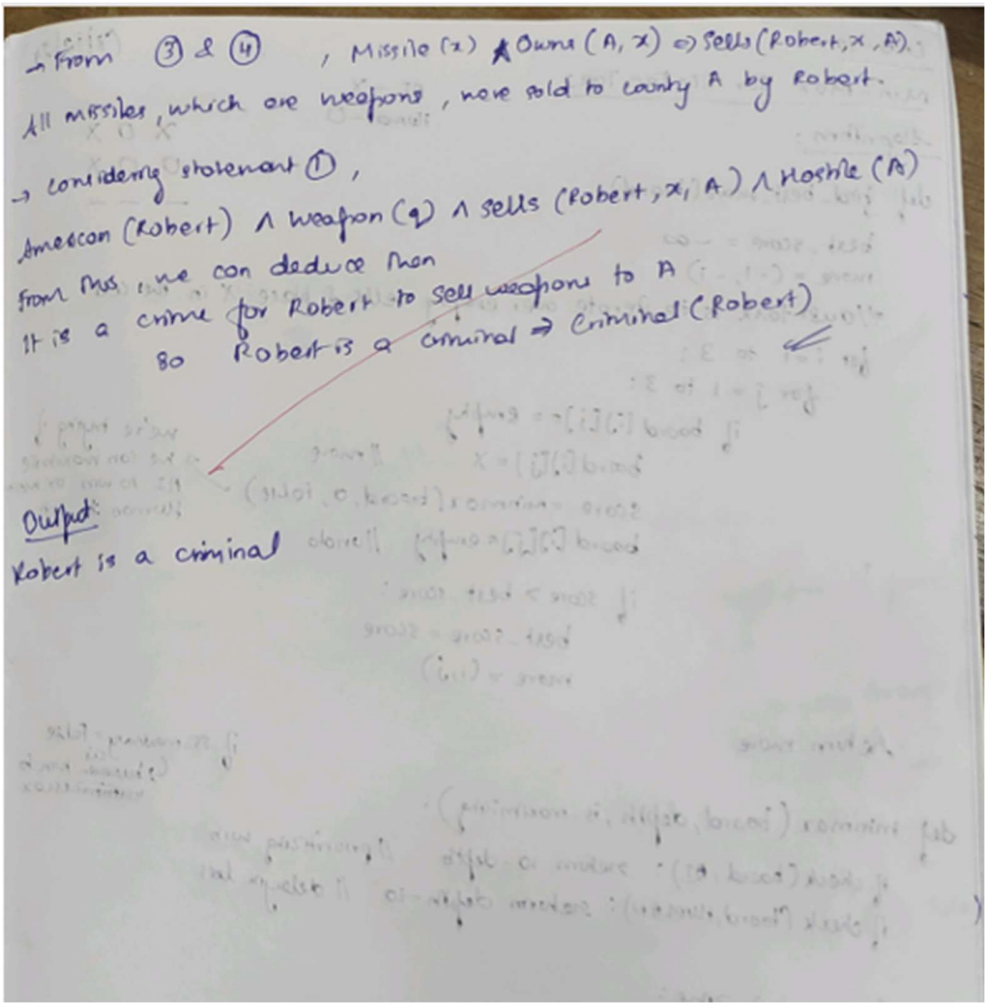
7) The country A, an enemy of America

Enemy(A, America)

• Observation:

→ From (5) & (7), we can say that A is enemy of America and hence it is a hostile of America (Hostile(A))

→ From (2) & (4), country A has some missiles & missile are weapons, so country A has ~~some~~ weapons (weapons(x))



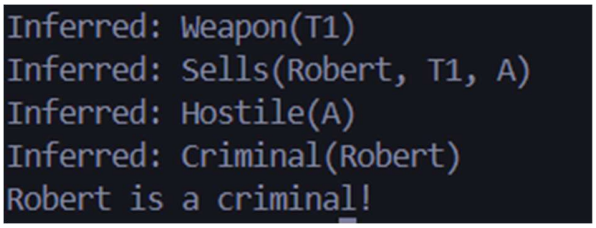
Code:

```
# 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()
```

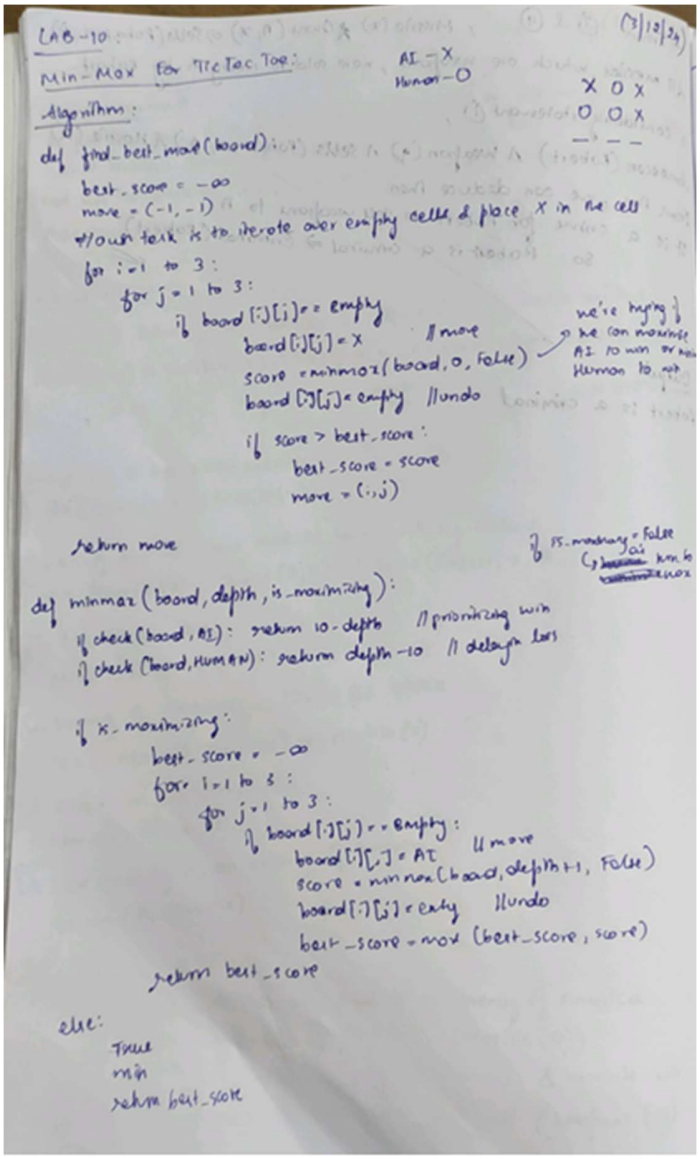
Output:



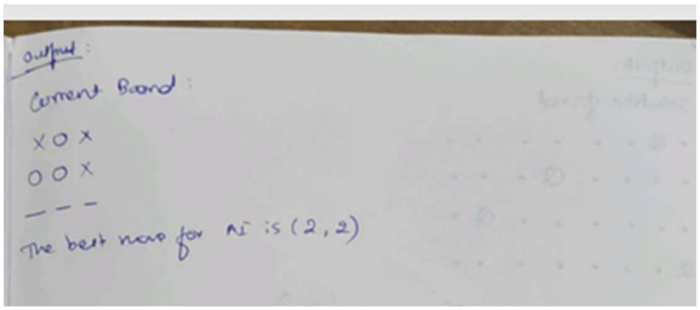
Program 10

Implement Min-Max Algorithm for Tic Tac Toe

Algorithm:



```
LAB-10:                                                    (3/10/24)
Min-Max for TicTac Toe:
                                          AI - X
                                          Human - O
                                                          X O X
                                                          O O X
                                                          _ _ _
Algorithm:
def find_best_move(board):
    best_score = -∞
    move = (-1, -1)
    #own task is to iterate over empty cells & place X in the cell
    for i=1 to 3:
        for j = 1 to 3:
            if board[i][j] = empty
                board[i][j] = X          # move
                score = minmax(board, 0, False)      <- we're trying to see can maximise AI to win or not Human to not
                board[i][j] = empty      #undo
                if score > best_score:
                    best_score = score
                    move = (i,j)

    return move                                        if is_maximizing = False
                                                       (human turn to minimize)
def minmax(board, depth, is_maximizing):
    if check(board, AI): return 10 - depth   #prioritizing with
    if check(board, HUMAN): return depth - 10   # delaying loss

    if is_maximizing:
        best_score = -∞
        for i=1 to 3:
            for j=1 to 3:
                if board[i][j] == empty:
                    board[i][j] = AI          #move
                    score = minmax(board, depth+1, False)
                    board[i][j] = empty       #undo
                    best_score = max(best_score, score)
        return best_score

    else:
        True
        min
        return best_score
```



```
output:
Current Board:
X O X
O O X
_ _ _
The best move for AI is (2, 2)
```

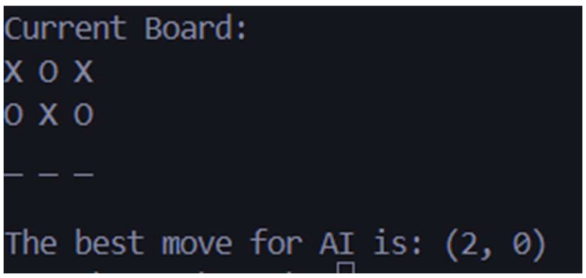
Code:

```
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

Algorithm:

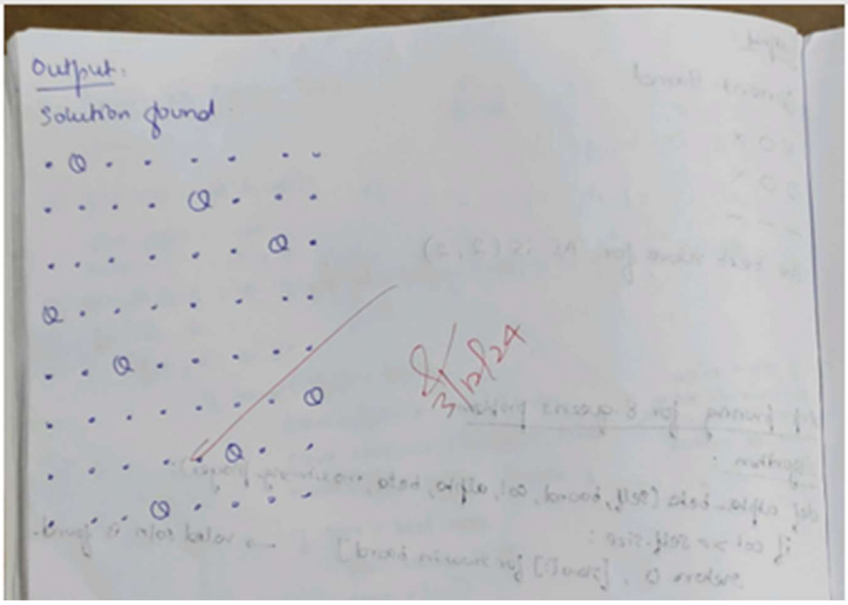
α-β pruning for 8-queens problem:

Algorithm:

```
def alpha_beta (self, board, col, alpha, beta, maximizing_player):
    if col >= self.size :
        return 0 , [row[:] for row in board]          → valid soln is found.

    if maximizing_player :                  # maximizing score
        max_eval = -∞                                  → Iterates through each col & if placing a queen is safe.
        best_board = None
        for row in range (self.size):
            if self.is_safe (board,row,col):
                board[row][col]=1
                eval_score , pot_board = alpha_beta (board, col+1 ,at,β, False
                board[row][col]= 0
                if eval_score > max :
                    max = eval_score
                    best_board = pot_board
                    alpha = max (alpha, eval_score)
                if beta <= alpha:          # Beta cutoff
                    break
        return max_eval, best_board

                                           # minimizing score
    else:
        True
    beta= min( )
        ~~beta~~
        beta <= alpha :        # alpha cutoff
            break
        return min_eval, best_board
```

Output:

Solution found:



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: