VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Pannaga R Bhat (1BM22CS189)

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 **Pannaga R Bhat (1BM22CS189),** who is a 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.

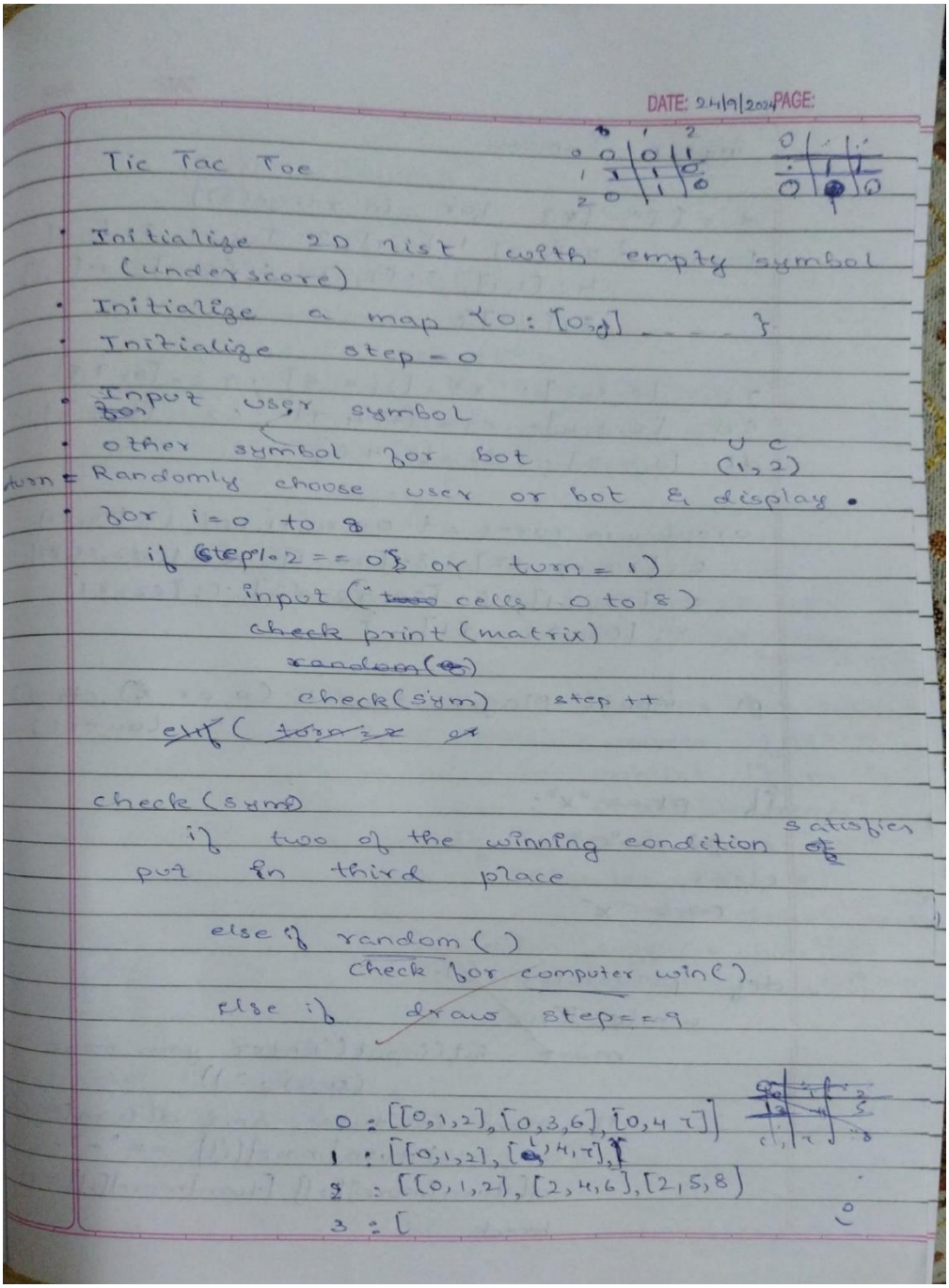
Prof. Sneha S Bagalkot	Dr. Kavitha Sooda
Assistant Professor	Professor & HOD
Department of CSE, BMSCE	Department of CSE, BMSCE

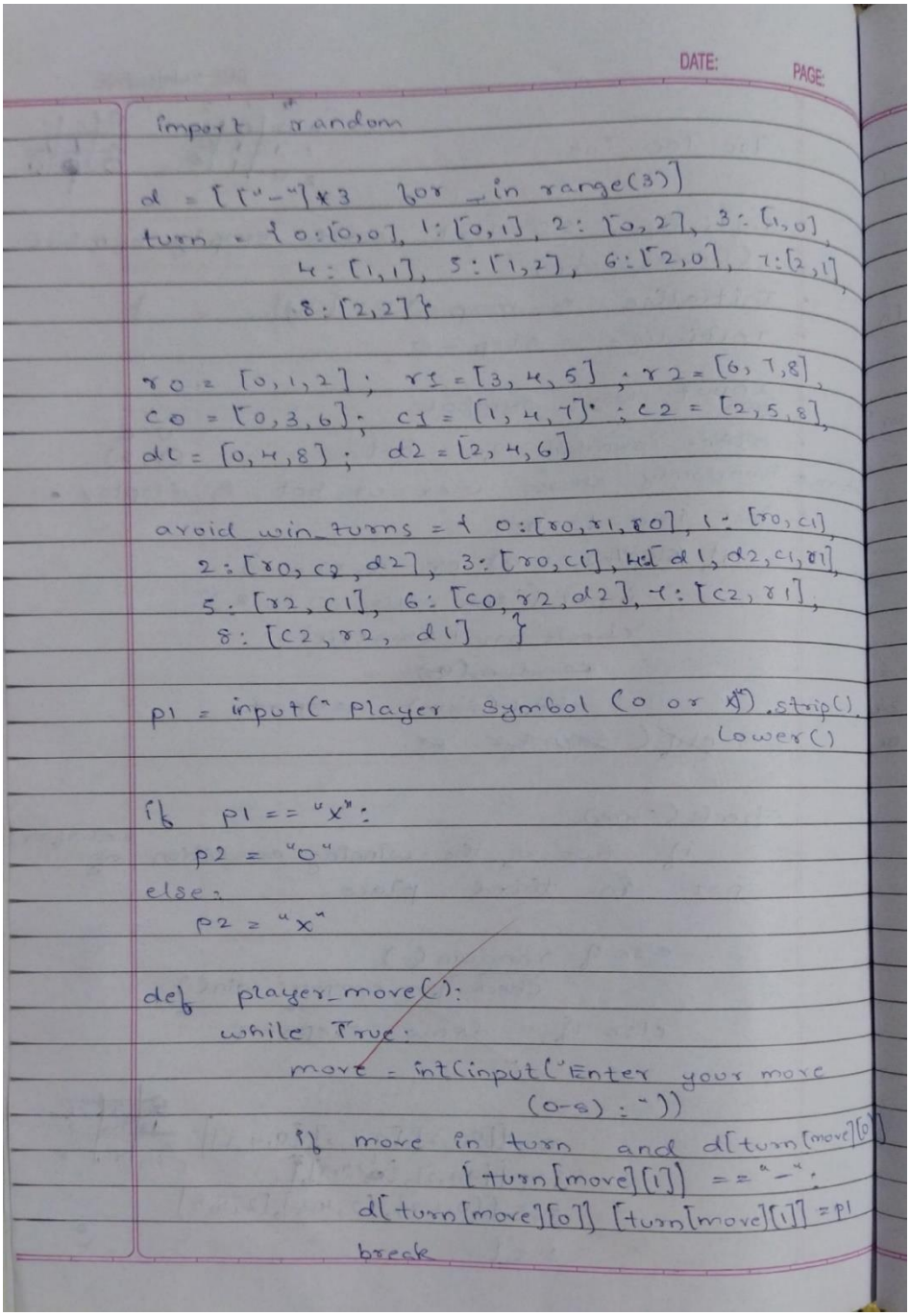
Index

Github Link: https://github.com/pannaga-rj/AI-LAB-1BM22CS189

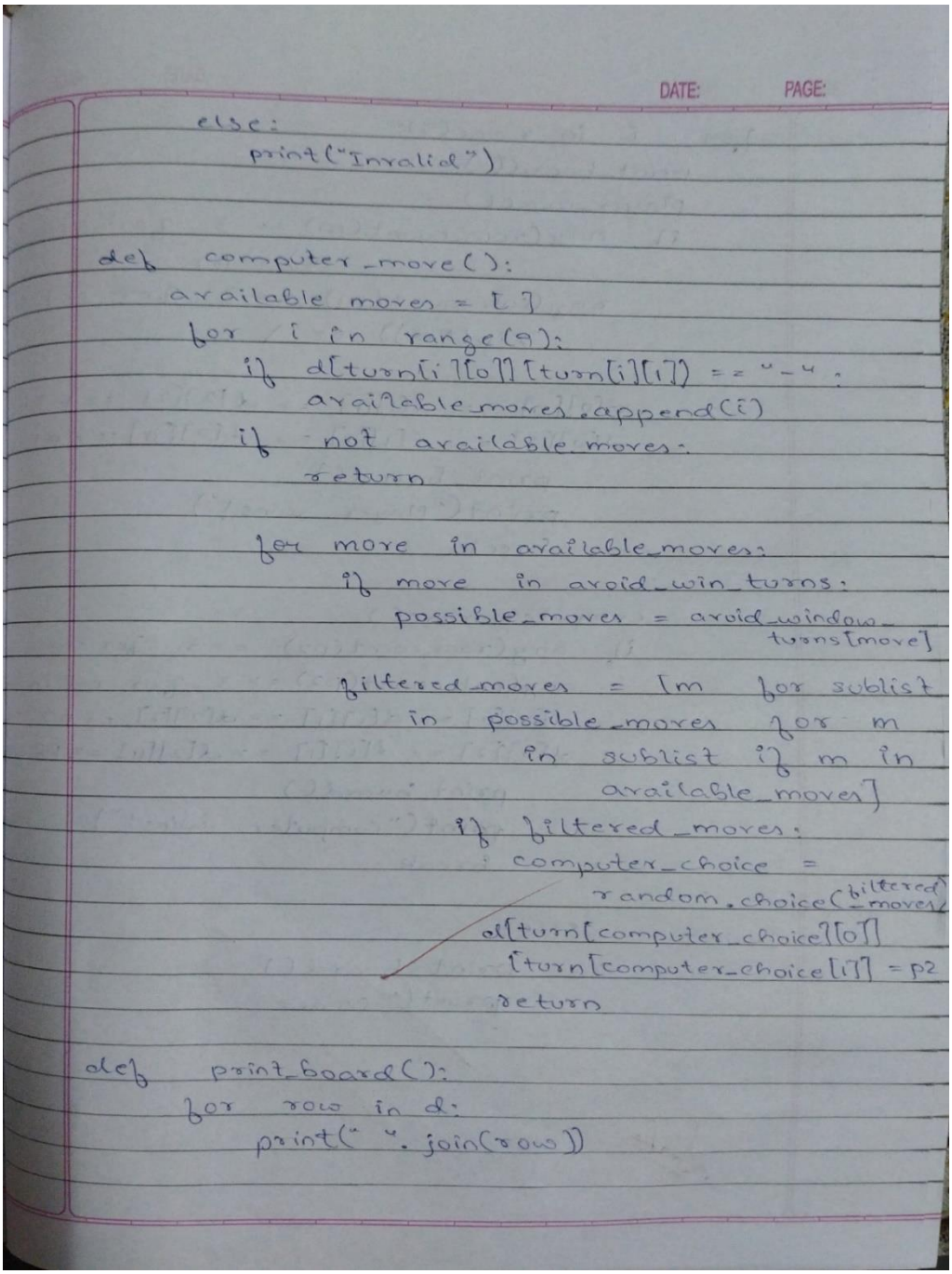
Implement Tic-Tac-Toe Game.

Algorithm:

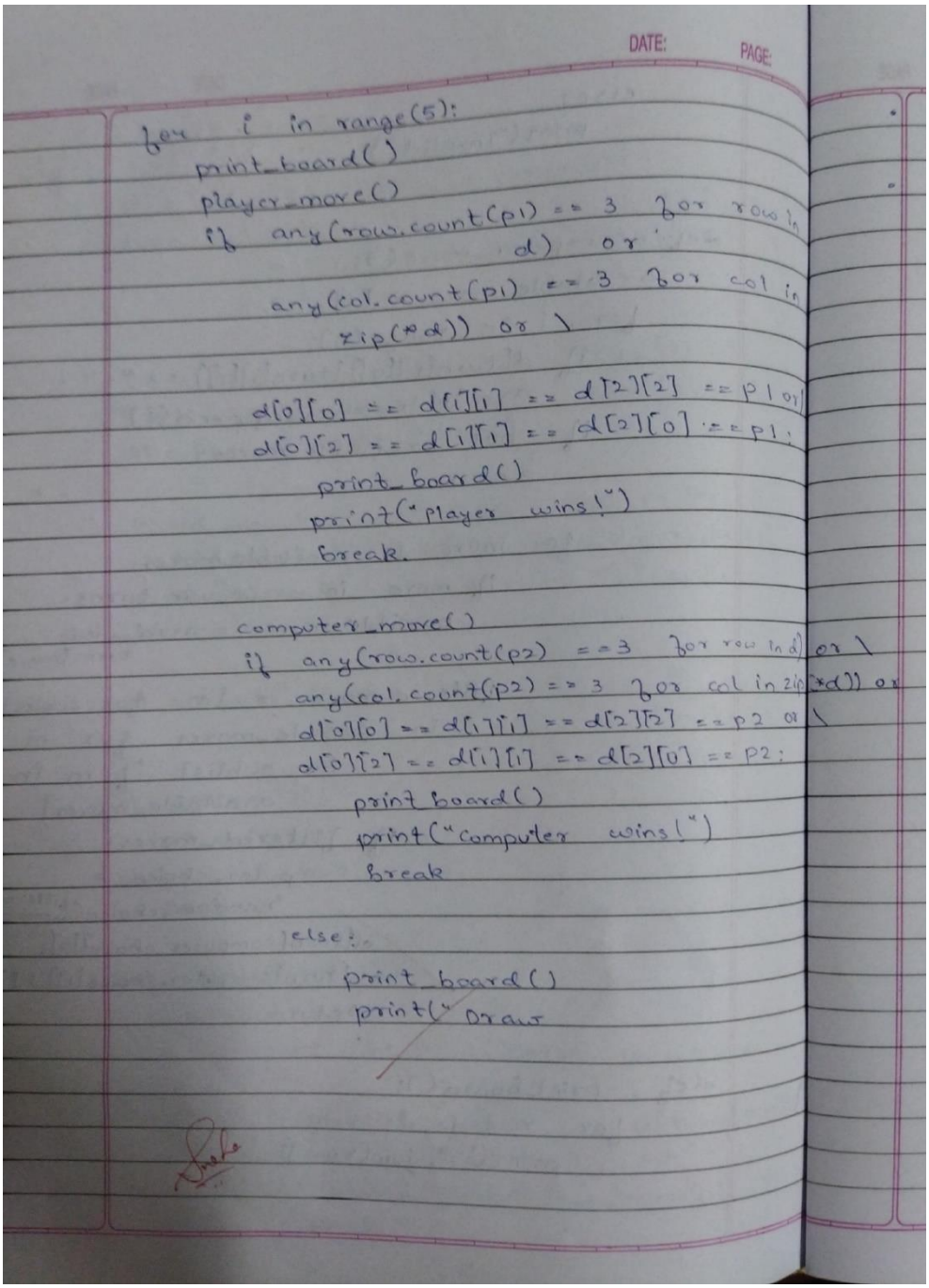




```
import random

d = [["-"]*3 for _ in range(3)]
turn = {0:[0,0], 1:[0,1], 2:[0,2], 3:[1,0],
        4:[1,1], 5:[1,2], 6:[2,0], 7:[2,1],
        8:[2,2]}

r0 = [0,1,2]; r1 = [3,4,5]; r2 = [6,7,8],
c0 = [0,3,6]; c1 = [1,4,7]; c2 = [2,5,8],
d1 = [0,4,8]; d2 = [2,4,6]

avoid_win_turns = {0:[r0,r1,r0], 1:[r0,c1],
  2:[r0,c2,d2], 3:[r0,c1], 4:[d1,d2,c1,r1],
  5:[r2,c1], 6:[c0,r2,d2], 7:[c2,r1],
  8:[c2,r2,d1]}

p1 = input("player symbol (o or x)").strip().
                                    lower()

if p1 == "x":
    p2 = "o"
else:
    p2 = "x"

def player_move():
    while True:
        move = int(input("Enter your move
                          (0-8):"))
        if move in turn and d[turn[move][0]
                              [turn[move][1]] == "-":
            d[turn[move][0]][turn[move][1]] = p1
            break
```



```
DATE:        PAGE:

    else:
        print("Invalid")


def computer_move():
    available_moves = []
    for i in range(9):
        if d[turn[i][0]][turn[i][1]] == "-":
            available_moves.append(i)
    if not available_moves:
        return

    for move in available_moves:
        if move in avoid_win_turns:
            possible_moves = avoid_window_
                                turns[move]

    filtered_moves = [m for sublist
        in possible_moves for m
            in sublist if m in
                available_moves]
            if filtered_moves:
                computer_choice =
                    random.choice(filtered_moves)
                d[turn[computer_choice][0]]
                [turn[computer_choice[1]] = p2
                return


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



```
for i in range(5):
    print_board()
    player_move()
    if any(row.count(p1) == 3 for row in
                                d) or
       any(col.count(p1) == 3 for col in
           zip(*d)) or \
       d[0][0] == d[1][1] == d[2][2] == p1 or\
       d[0][2] == d[1][1] == d[2][0] == p1:
        print_board()
        print("Player wins!")
        break.

    computer_move()
    if any(row.count(p2) == 3 for row in d) or \
       any(col.count(p2) == 3 for col in zip(*d)) or
       d[0][0] == d[1][1] == d[2][2] == p2 or \
       d[0][2] == d[1][1] == d[2][0] == p2:
        print_board()
        print("computer wins!")
        break

    else:
        print_board()
        print("Draw
```

Code:

```
import random


# Initialize the game board
d = [["-"] * 3 for _ in range(3)]
turn = {0: [0, 0], 1: [0, 1], 2: [0, 2], 3: [1, 0], 4: [1, 1], 5: [1, 2], 6: [2, 0], 7: [2, 1], 8: [2, 2]}


r0 = [0, 1, 2]
r1 = [3, 4, 5]
r2 = [6, 7, 8]
c0 = [0, 3, 6]
c1 = [1, 4, 7]
c2 = [2, 5, 8]
d1 = [0, 4, 8]  # Diagonal \
d2 = [2, 4, 6]  # Diagonal /


avoid_win_turns = {
    0: [r0, d1, c0],
    1: [r0, c1],
    2: [r0, c2, d2],
    3: [r1, c0],
    4: [d1, d2, c1, r1],
    5: [r1, c2],
    6: [r2, c0, d2],
    7: [r2, c1],
    8: [r2, c2, d1]
}
```

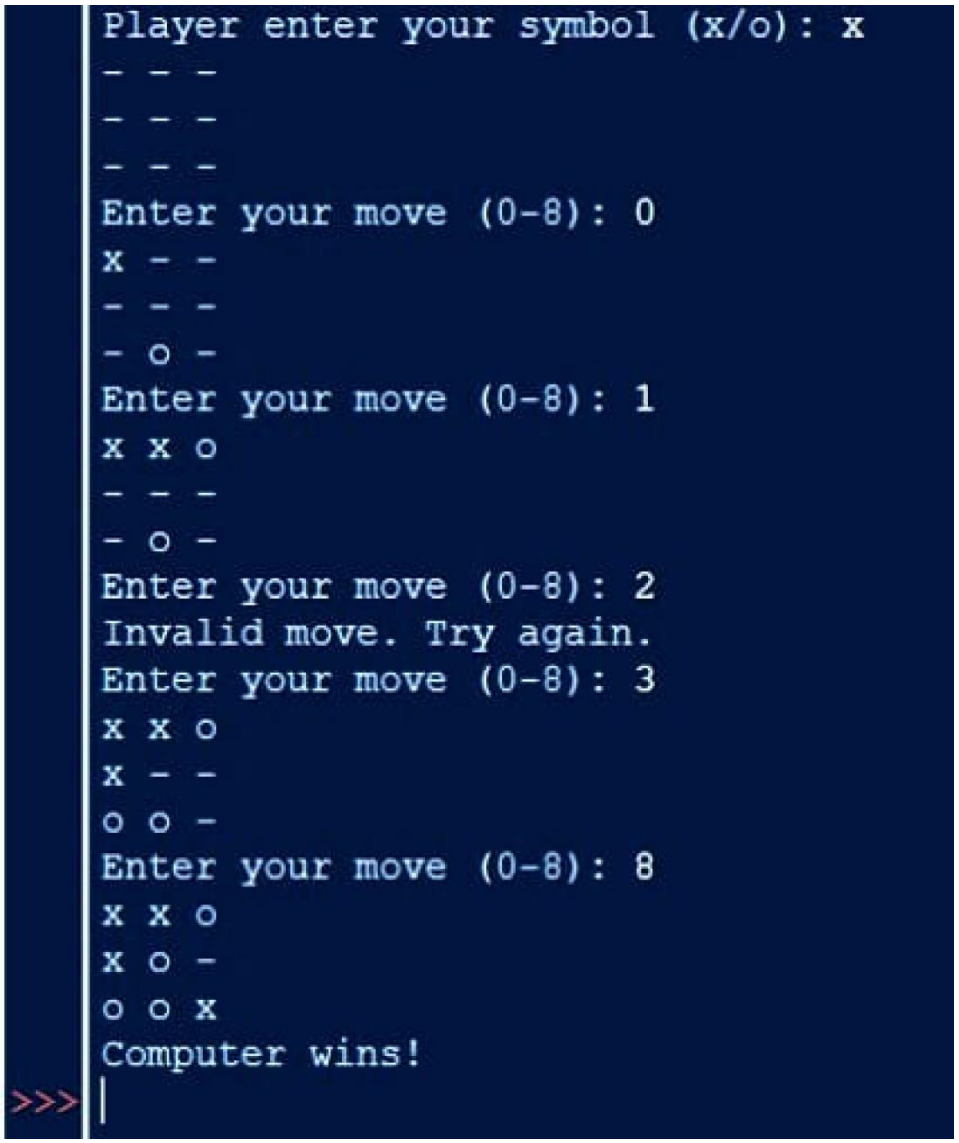
```
# Get player symbol
p1 = input("Player enter your symbol (x/o): ").strip().lower()

if p1 == "x":
    p2 = "o"
else:
    p2 = "x"

# Check if a player is about to win
def check_winning_move(symbol):
    for move, lines in avoid_win_turns.items():
        if d[turn[move][0]][turn[move][1]] == "-":  # Only consider available spots
            for line in lines:
                if sum(1 for m in line if d[turn[m][0]][turn[m][1]] == symbol) == 2:
                    return move
    return None

# Player's move
def player_move():
    while True:
        move = int(input("Enter your move (0-8): "))
        if move in turn and d[turn[move][0]][turn[move][1]] == "-":
            d[turn[move][0]][turn[move][1]] = p1
            break
        else:
            print("Invalid move. Try again.")
```

```
# Computer's move
def computer_move():

    ''' getting index of empty rows '''
    available_moves = [i for i in range(9) if d[turn[i][0]][turn[i][1]] == "-"]

    if not available_moves:
        return  # No moves available

    # Check if the computer can win
    winning_move = check_winning_move(p2)
    if winning_move is not None:
        d[turn[winning_move][0]][turn[winning_move][1]] = p2
        return

    # Check if the player is about to win and block them
    block_move = check_winning_move(p1)
    if block_move is not None:
        d[turn[block_move][0]][turn[block_move][1]] = p2
        return

    # Otherwise, make a move from the available moves based on avoid_win_turns
    for move in available_moves:
        if move in avoid_win_turns:
            possible_moves = avoid_win_turns[move]
            # Flatten the possible moves and filter by available moves
```

```
            filtered_moves = [m for sublist in possible_moves for m in sublist if m in available_moves]
            if filtered_moves:
                computer_choice = random.choice(filtered_moves)
                d[turn[computer_choice][0]][turn[computer_choice][1]] = p2
                return

    # If no strategic move, pick randomly
    random_move = random.choice(available_moves)
    d[turn[random_move][0]][turn[random_move][1]] = p2

# Print the board
def print_board():
    for row in d:
        print(" ".join(row))

# Check if a player has won
def check_winner(symbol):
    return any(row.count(symbol) == 3 for row in d) or \
           any(col.count(symbol) == 3 for col in zip(*d)) or \
           (d[0][0] == d[1][1] == d[2][2] == symbol) or \
           (d[0][2] == d[1][1] == d[2][0] == symbol)

# Game loop
for _ in range(5):  # Maximum of 5 turns (5 moves each)
    print_board()
    player_move()
    if check_winner(p1):
```

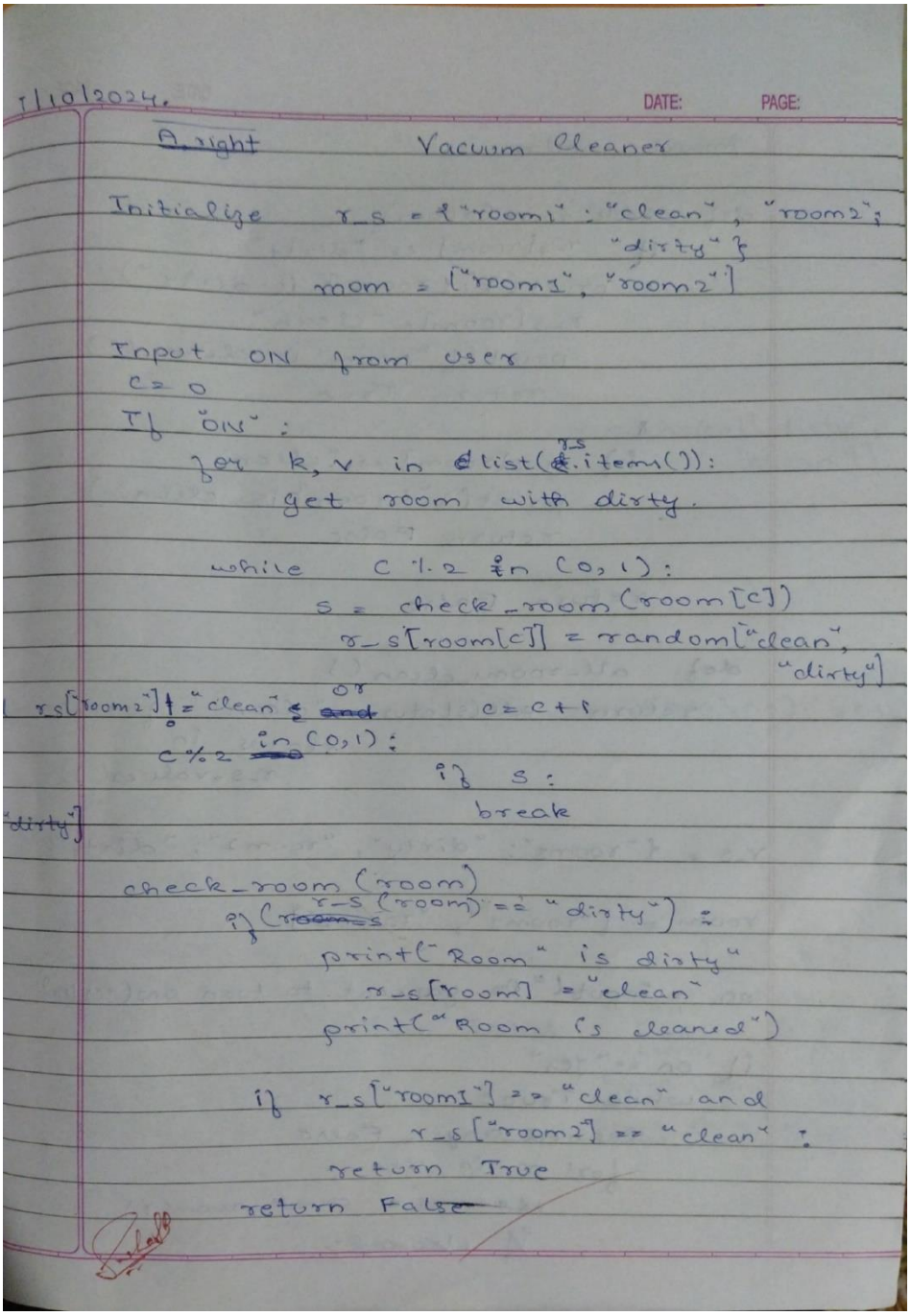
```
    print_board()
    print("Player wins!")
    break

if all(d[turn[i][0]][turn[i][1]] != "-" for i in range(9)):
    print_board()
    print("It's a draw!")
    break

computer_move()
if check_winner(p2):
    print_board()
    print("Computer wins!")
    break

''' if all items are filled with x or o '''
if all(d[turn[i][0]][turn[i][1]] != "-" for i in range(9)):
    print_board()
    print("It's a draw!")
    break
```

Output:


Player enter your symbol (x/o): x
- - -
- - -
- - -
Enter your move (0-8): 0
x - -
- - -
- o -
Enter your move (0-8): 1
x x o
- - -
- o -
Enter your move (0-8): 2
Invalid move. Try again.
Enter your move (0-8): 3
x x o
x - -
o o -
Enter your move (0-8): 8
x x o
x o -
o o x
Computer wins!
>>>

Implement Vacuum Cleaner Agent.

Algorithm:



7/10/2024. DATE: PAGE:

~~A right~~ Vacuum Cleaner

Initialize r_s = {"room1": "clean", "room2": "dirty"}

room = ["room1", "room2"]

Input ON from user

c = 0

If "ON":

for k, v in list(r_s.items()):

get room with dirty.

while c % 2 in (0, 1):

s = check_room(room[c])

r_s[room[c]] = random["clean", "dirty"]

r_s["room2"] != "clean" or c = c + 1

c % 2 in (0, 1):

if s:

break

"dirty"]

check_room(room)

if (r_s(room) == "dirty"):

print("Room" is dirty"

r_s[room] = "clean"

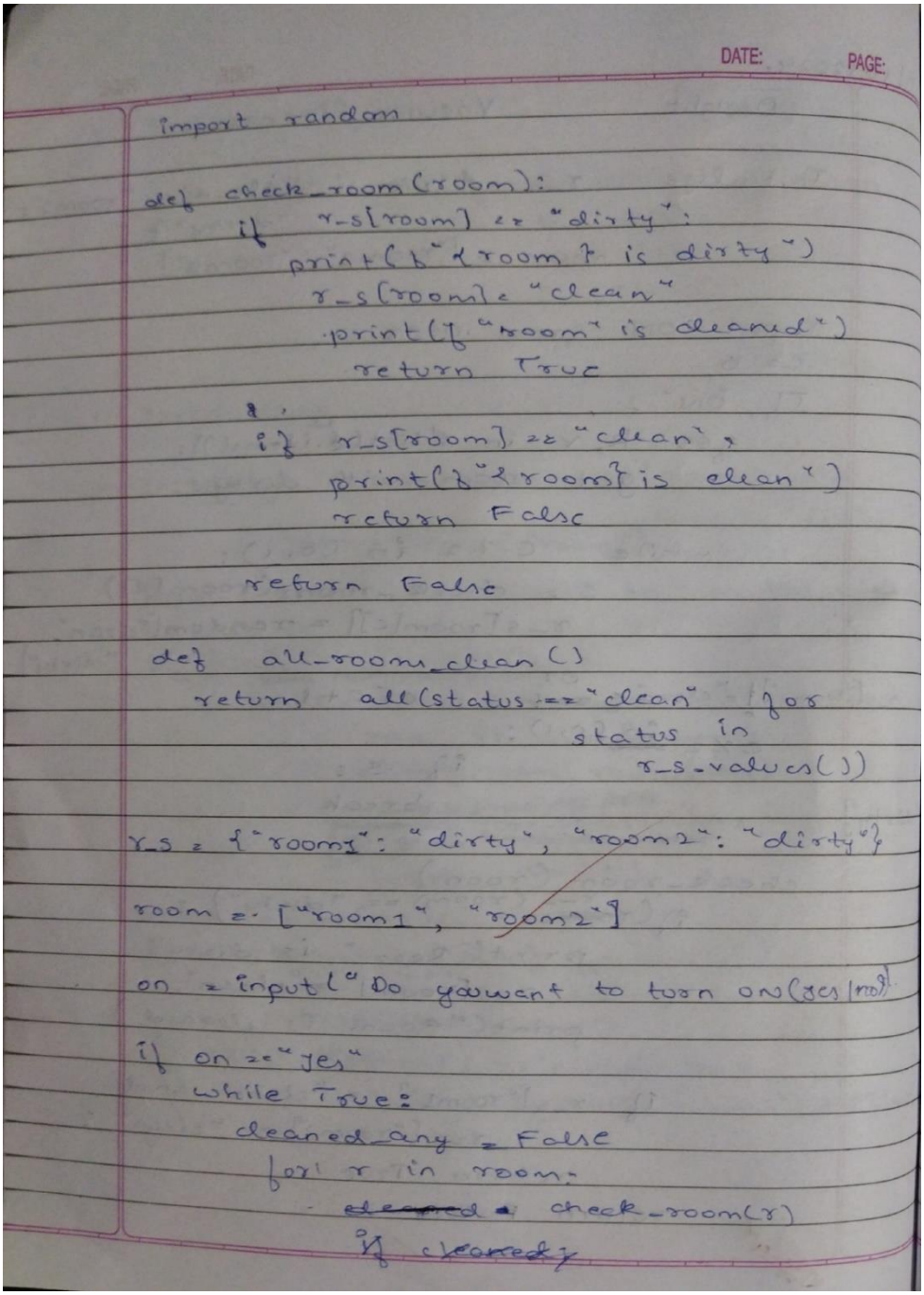
print("Room is cleaned")

if r_s["room1"] == "clean" and

r_s["room2"] == "clean":

return True

return False



DATE: PAGE:

```
import random

def check_room(room):
    if r_s[room] == "dirty":
        print(f"{room} is dirty")
        r_s[room] = "clean"
        print(f"{room} is cleaned")
        return True
    elif r_s[room] == "clean":
        return False
    return False

def all_room_clean():
    return all(status == "clean" for
               status in
               r_s.values())

r_s = {"room1": "dirty", "room2": "dirty"}

room = ["room1", "room2"]

on = input("Do you want to turn on(yes/no)

if on == "yes"
    while True:
        cleaned_any = False
        for r in room:
            cleaned = check_room(r)
            if cleaned:
```

DATE: PAGE:

```
    cleaned_any = True

if all_rooms_clean():
    print("Both are clean")
    break

if cleaned_any:
    for r in room:
        r_s[r] = random.choice(["dirty",
                                "clean"])

else:
    print("cleaner is off")
```

Output:

Do you want to turn ON (yes/no) yes
room1 is clean
room2 is dirty
room 2 is cleaned
room1 is dirty
room1 is cleaned
room2 is clean
Both rooms are clean

[R1] → [R2]

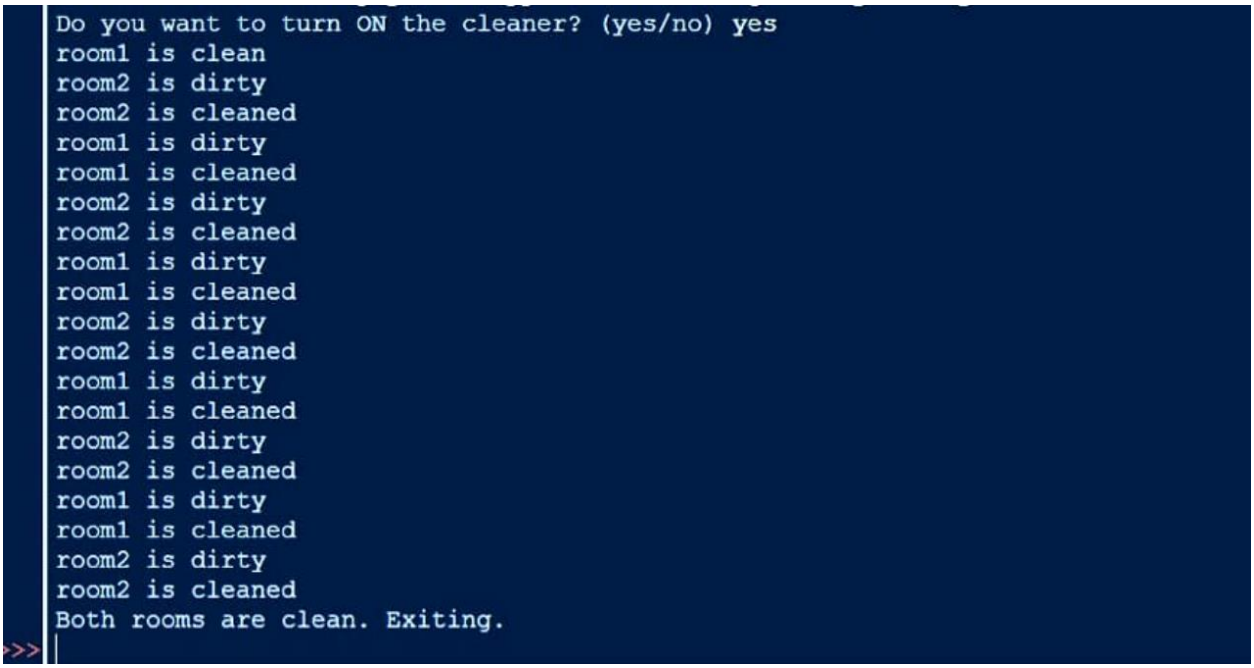
	Initial	Room1 (clean)	Action move right
	Room1 (clean)	Room2 (dirty)	clean move left
	Room1 (clean)	Room2 (dirty)	Room2 (clean)

Code:

```
import random


def check_room(room):
    if r_s[room] == "dirty":
        print(f"{room} is dirty")
        r_s[room] = "clean"  # Clean the room
        print(f"{room} is cleaned")
        return True  # Room was dirty and has now been cleaned

    if r_s[room] == "clean":
        print(f"{room} is clean")
        return False  # Room was already clean

    return False

# checking if all the rooms are clean
def all_rooms_clean():
    return all(status == "clean" for status in r_s.values())




# Start point
r_s = {"room1": "clean", "room2": "dirty"}
room = ["room1", "room2"]

on_status = input("Do you want to turn ON the cleaner? (yes/no) ").lower().strip()
```

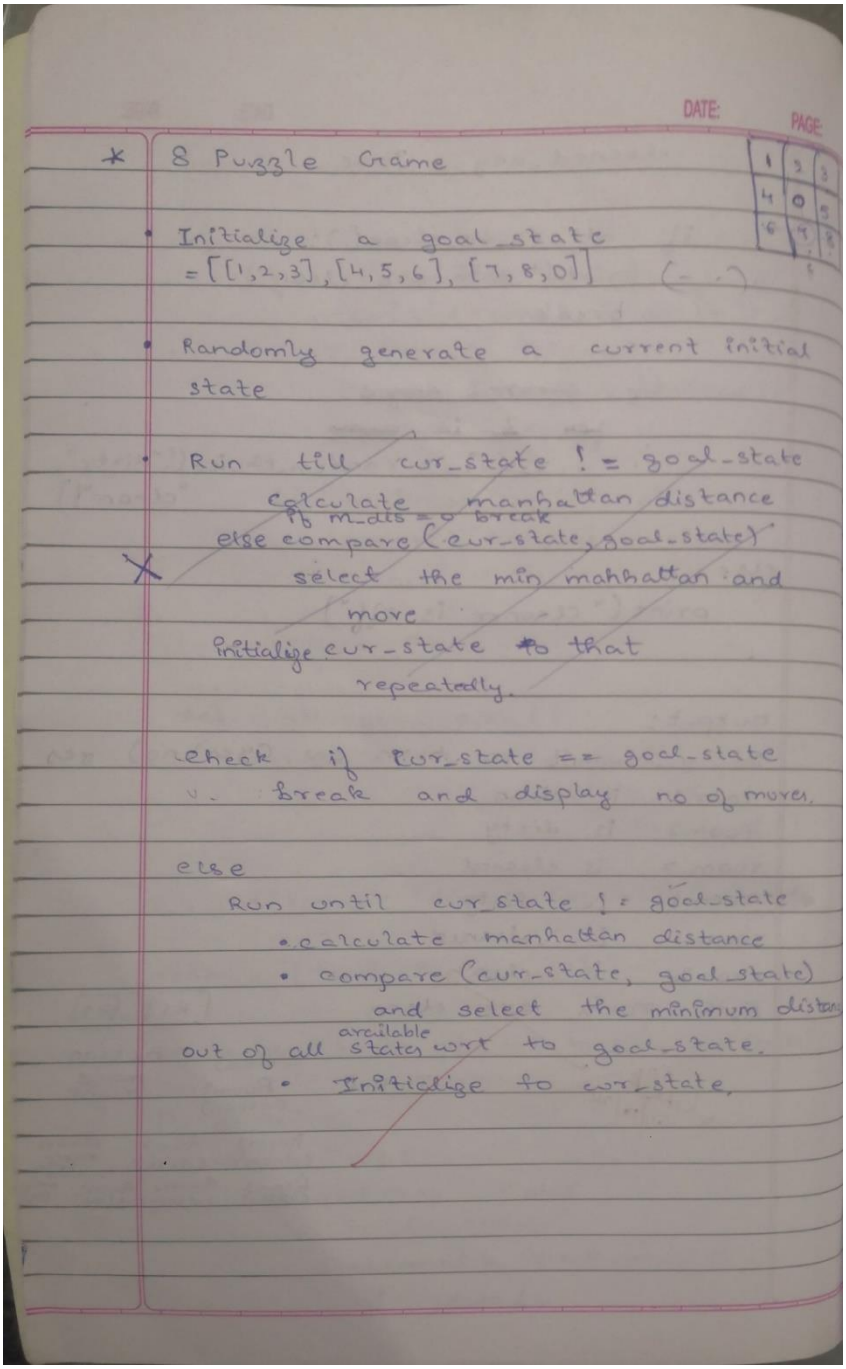
```
if on_status == "yes":
    while True:  # Continue until both rooms are clean
        for r in room:
            check_room(r)  # Check and clean the current room


            # After checking, randomly assign the status for the room
            r_s[r] = random.choice(["dirty", "clean"])


        # Check if all rooms are clean after processing both rooms
        if all_rooms_clean():
            print("Both rooms are clean. Exiting.")
            break


else:
    print("Cleaner is off.")
```

Output:


Do you want to turn ON the cleaner? (yes/no) yes
room1 is clean
room2 is dirty
room2 is cleaned
room1 is dirty
room1 is cleaned
room2 is dirty
room2 is cleaned
room1 is dirty
room1 is cleaned
room2 is dirty
room2 is cleaned
room1 is dirty
room1 is cleaned
room2 is dirty
room2 is cleaned
room1 is dirty
room1 is cleaned
room2 is dirty
room2 is cleaned
Both rooms are clean. Exiting.
>>>

Implement 8 puzzle problems using Depth First Search (DFS).

Algorithm:

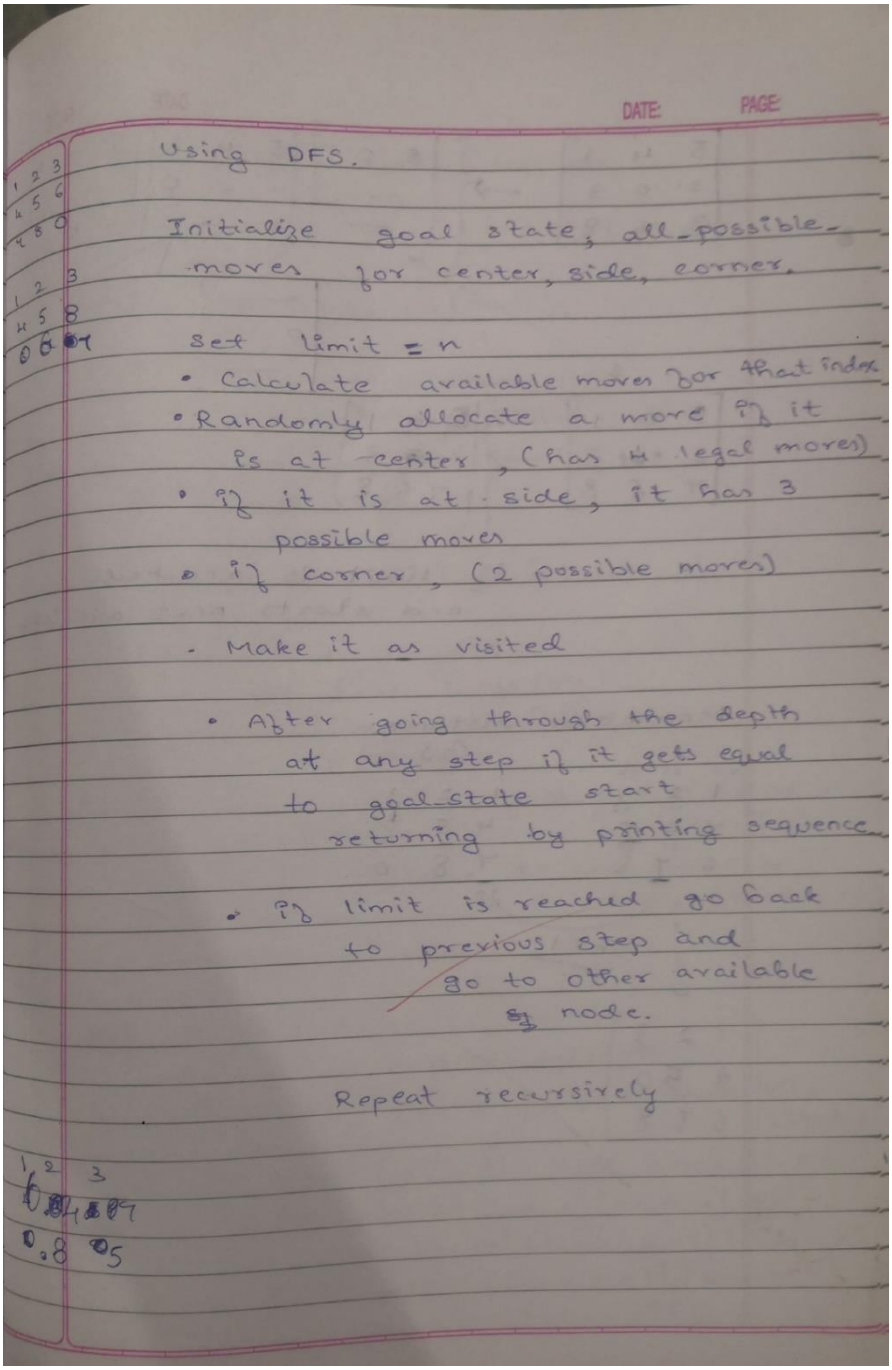


* 8 Puzzle Game

- Initialize a goal_state = [[1,2,3],[4,5,6],[7,8,0]]
- Randomly generate a current initial state
- Run till cur_state != goal_state
  calculate manhattan distance
  If m_dis = 0 break
  else compare (cur_state, goal_state)
  select the min manhattan and move
  Initialize cur_state to that
  repeatedly.

check if cur_state == goal_state
break and display no of moves.

else
Run until cur state != goal_state
- calculate manhattan distance
- compare (cur_state, goal_state) and select the minimum distance out of all available states wrt to goal_state.
- Initialize to cur_state.



DATE: PAGE:

Using DFS.

Initialize goal state, all_possible_movers for center, side, corner.

Set limit = n

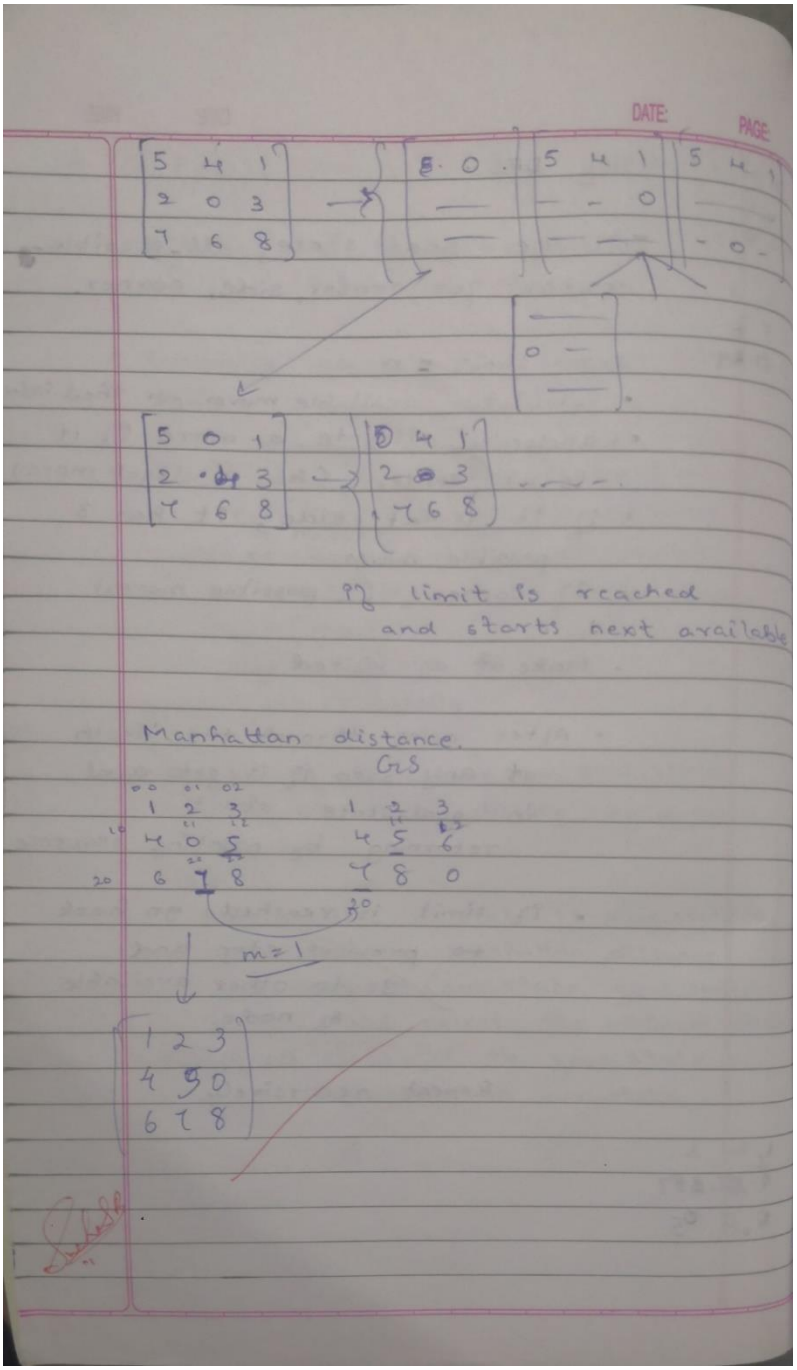
- Calculate available movers for that index
- Randomly allocate a move if it is at center, (has 4 legal moves)
- if it is at side, it has 3 possible moves
- if corner, (2 possible moves)

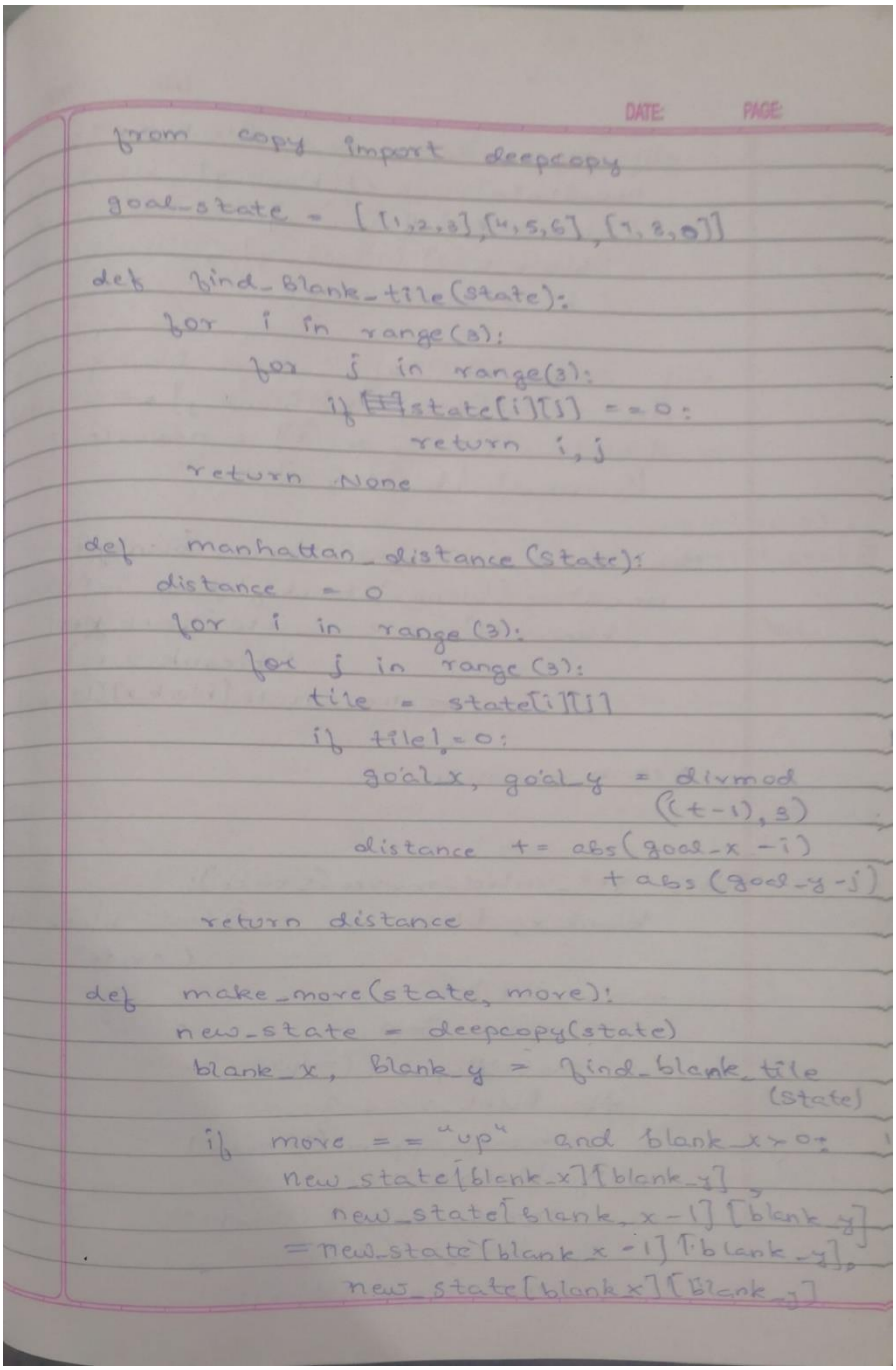
- Make it as visited

- After going through the depth at any step if it gets equal to goal_state start returning by printing sequence

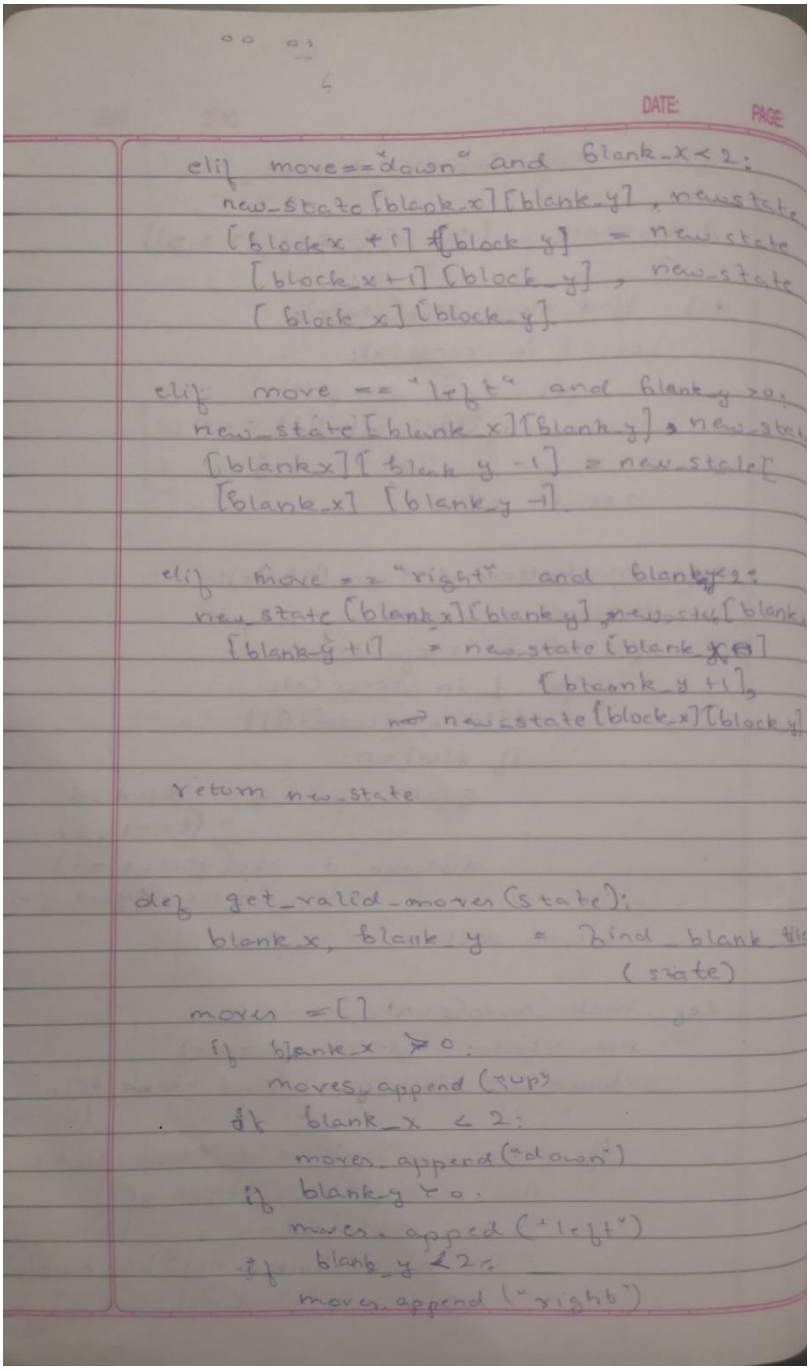
- if limit is reached go back to previous step and go to other available if node.

Repeat recursively

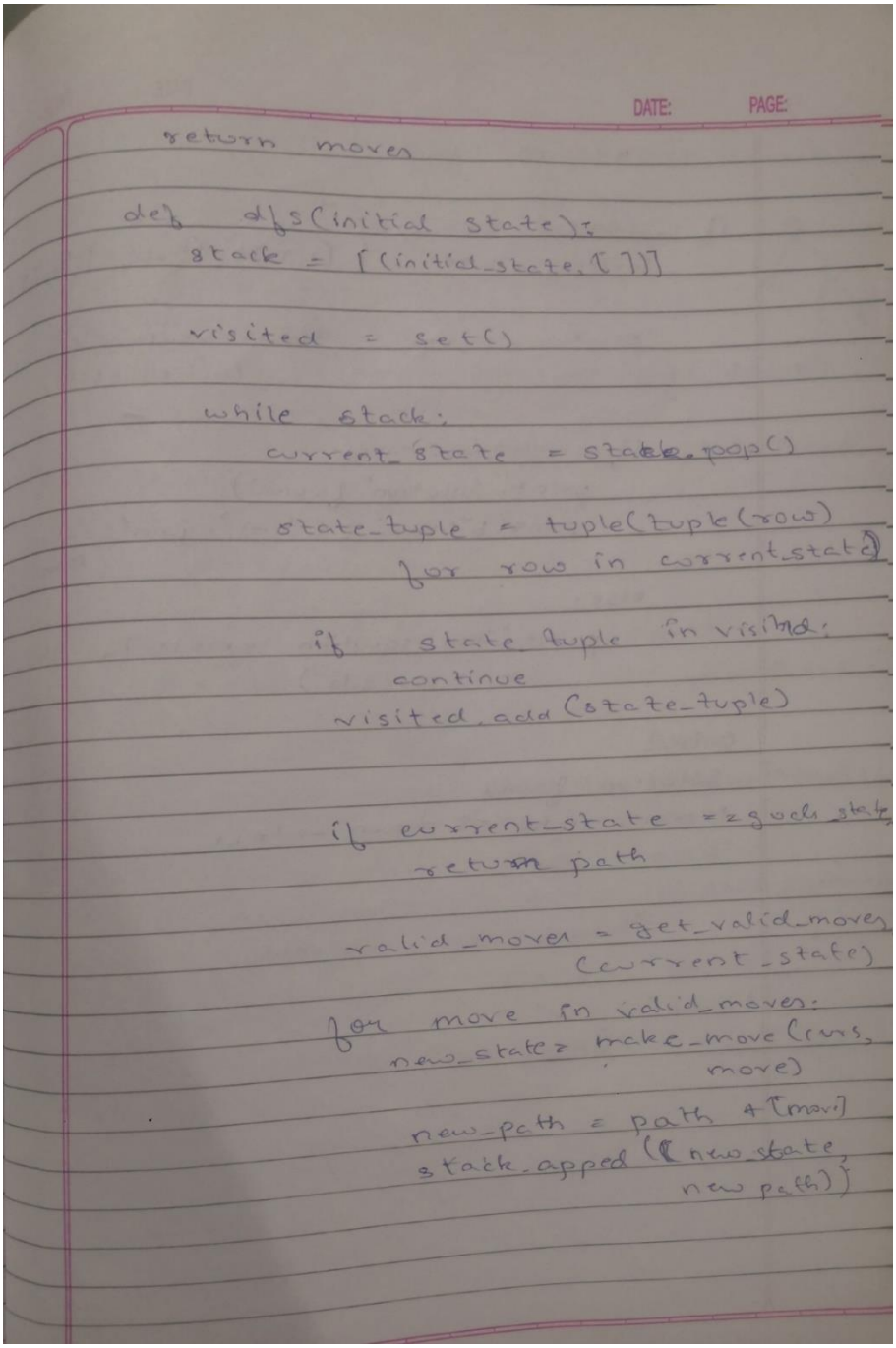

DATE:
PAGE:
limit is reached
and starts next available
Manhattan distance.
GrS
m=1



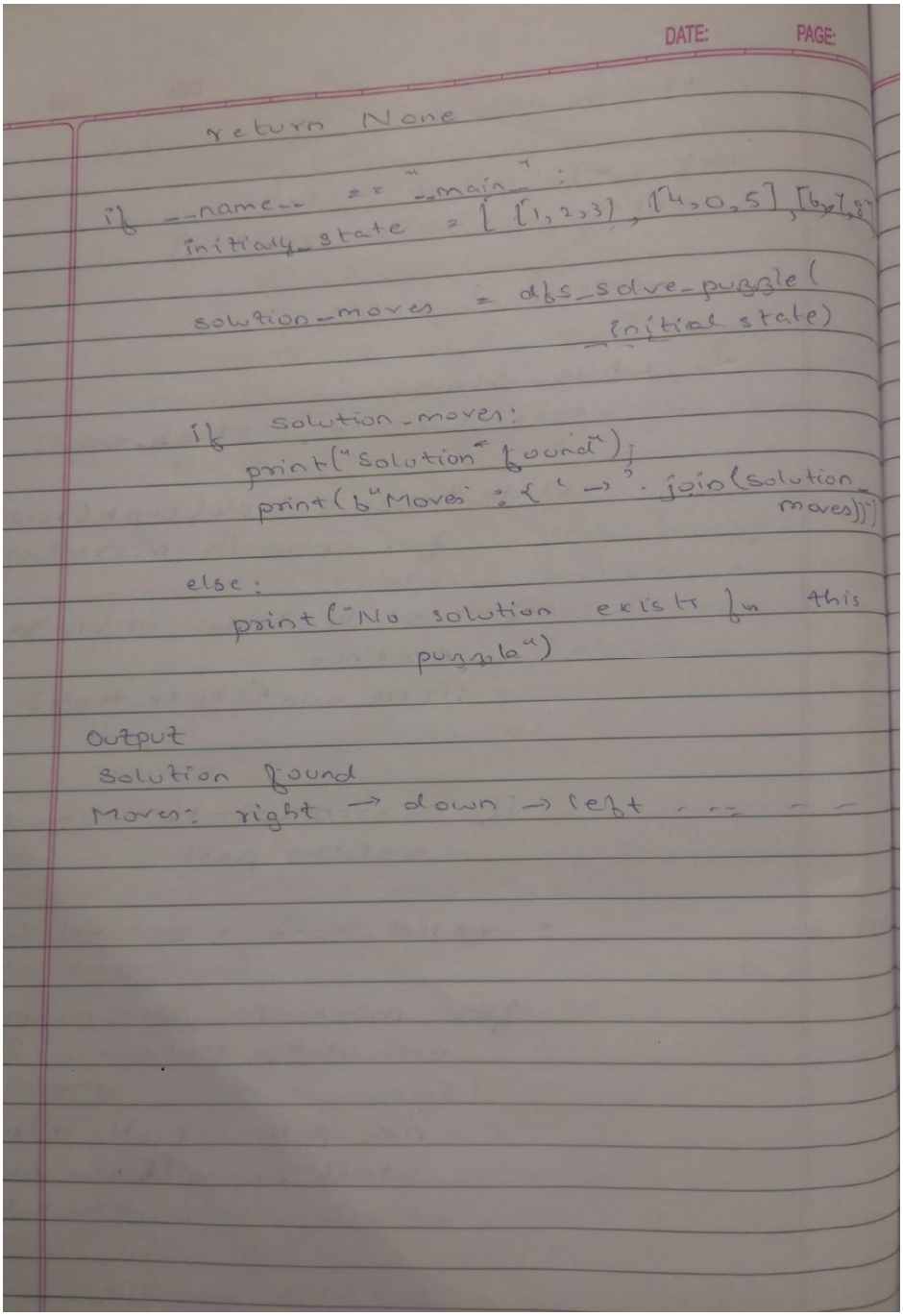
```
from copy import deepcopy

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

def find_blank_tile(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return None

def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            tile = state[i][j]
            if tile != 0:
                goal_x, goal_y = divmod
                                 ((t-1), 3)
                distance += abs(goal_x - i)
                            + abs(goal_y - j)
    return distance

def make_move(state, move):
    new_state = deepcopy(state)
    blank_x, blank_y = find_blank_tile
                       (state)
    if move == "up" and blank_x > 0:
        new_state[blank_x][blank_y],
        new_state[blank_x-1][blank_y]
        = new_state[blank_x-1][blank_y],
        new_state[blank_x][blank_y]
```



```
elif move == "down" and blank_x < 2:
    new_state[blank_x][blank_y], new_state
    [blank_x + 1][blank_y] = new_state
    [blank_x+1][blank_y], new_state
    [blank_x][blank_y]

elif move == "left" and blank_y > 0:
    new_state[blank_x][blank_y], new_sta
    [blank_x][blank_y - 1] = new_state[
    [blank_x][blank_y]

elif move == "right" and blank_y < 2:
    new_state[blank_x][blank_y], new_sta[blank_
    [blank_y + 1] = new_state[blank_x
                    [blank_y + 1],
                new_state[block_x][block_y]

return new_state

def get_valid_moves(state):
    blank_x, blank_y = find_blank_
                        (state)
    moves = []
    if blank_x > 0:
        moves.append("up")
    if blank_x < 2:
        moves.append("down")
    if blank_y > 0:
        moves.append("left")
    if blank_y < 2:
        moves.append("right")
```



```
DATE:  PAGE:
return moves

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

    visited = set()

    while stack:
        current_state = stack.pop()

        state_tuple = tuple(tuple(row)
                for row in current_state)

        if state_tuple in visited:
            continue
        visited.add(state_tuple)


        if current_state == goal_state:
            return path

        valid_moves = get_valid_moves
                (current_state)

        for move in valid_moves:
            new_state = make_move(curr,
                    move)

            new_path = path + [move]
            stack.append((new_state,
                    new_path))
```



```
DATE:  PAGE:

    return None

if __name__ == "__main__":
    initial_state = [[1,2,3],[4,0,5],[6,7,8]]

    solution_moves = dfs_solve_puzzle(
                        initial_state)

    if solution_moves:
        print("Solution found")
        print("Moves: " + " → ".join(solution_
                                      moves))
    else:
        print("No solution exists for this
                puzzle")
```

Output

Solution found

Moves: right → down → left - -- - -

Code:

```
from copy import deepcopy

# Define the goal state
goal_state = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]

# Function to find the position of the blank tile (0)
def find_blank_tile(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return None

# Function to calculate the Manhattan distance
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            tile = state[i][j]
            if tile != 0:  # Ignore the blank tile
                goal_x, goal_y = divmod(tile - 1, 3)
                distance += abs(goal_x - i) + abs(goal_y - j)
```

```
    return distance


# Function to make a move by sliding the blank tile (0)
def make_move(state, move):
    new_state = deepcopy(state)
    blank_x, blank_y = find_blank_tile(state)


    if move == "up" and blank_x > 0:
        new_state[blank_x][blank_y], new_state[blank_x - 1][blank_y] = new_state[blank_x - 1][blank_y], new_state[blank_x][blank_y]
    elif move == "down" and blank_x < 2:
        new_state[blank_x][blank_y], new_state[blank_x + 1][blank_y] = new_state[blank_x + 1][blank_y], new_state[blank_x][blank_y]
    elif move == "left" and blank_y > 0:
        new_state[blank_x][blank_y], new_state[blank_x][blank_y - 1] = new_state[blank_x][blank_y - 1], new_state[blank_x][blank_y]
    elif move == "right" and blank_y < 2:
        new_state[blank_x][blank_y], new_state[blank_x][blank_y + 1] = new_state[blank_x][blank_y + 1], new_state[blank_x][blank_y]


    return new_state


# Function to get valid moves for the blank tile (0)
def get_valid_moves(state):
    blank_x, blank_y = find_blank_tile(state)
    moves = []
    if blank_x > 0:
        moves.append("up")
    if blank_x < 2:
```

```
        moves.append("down")
    if blank_y > 0:
        moves.append("left")
    if blank_y < 2:
        moves.append("right")
    return moves

# DFS search function using a stack
def dfs_solve_puzzle(initial_state):
    # Stack for DFS (each entry contains the current state and path of moves)
    stack = [(initial_state, [])]
    # Keep track of visited states
    visited = set()
    while stack:
        current_state, path = stack.pop()
        # Mark the current state as visited (convert the state to a tuple so it's hashable)
        state_tuple = tuple(tuple(row) for row in current_state)
        if state_tuple in visited:
            continue
        visited.add(state_tuple)
        # If the current state is the goal state, return the path of moves
        if current_state == goal_state:
            return path

        # Get all valid moves and generate new states
        valid_moves = get_valid_moves(current_state)
        for move in valid_moves:
```

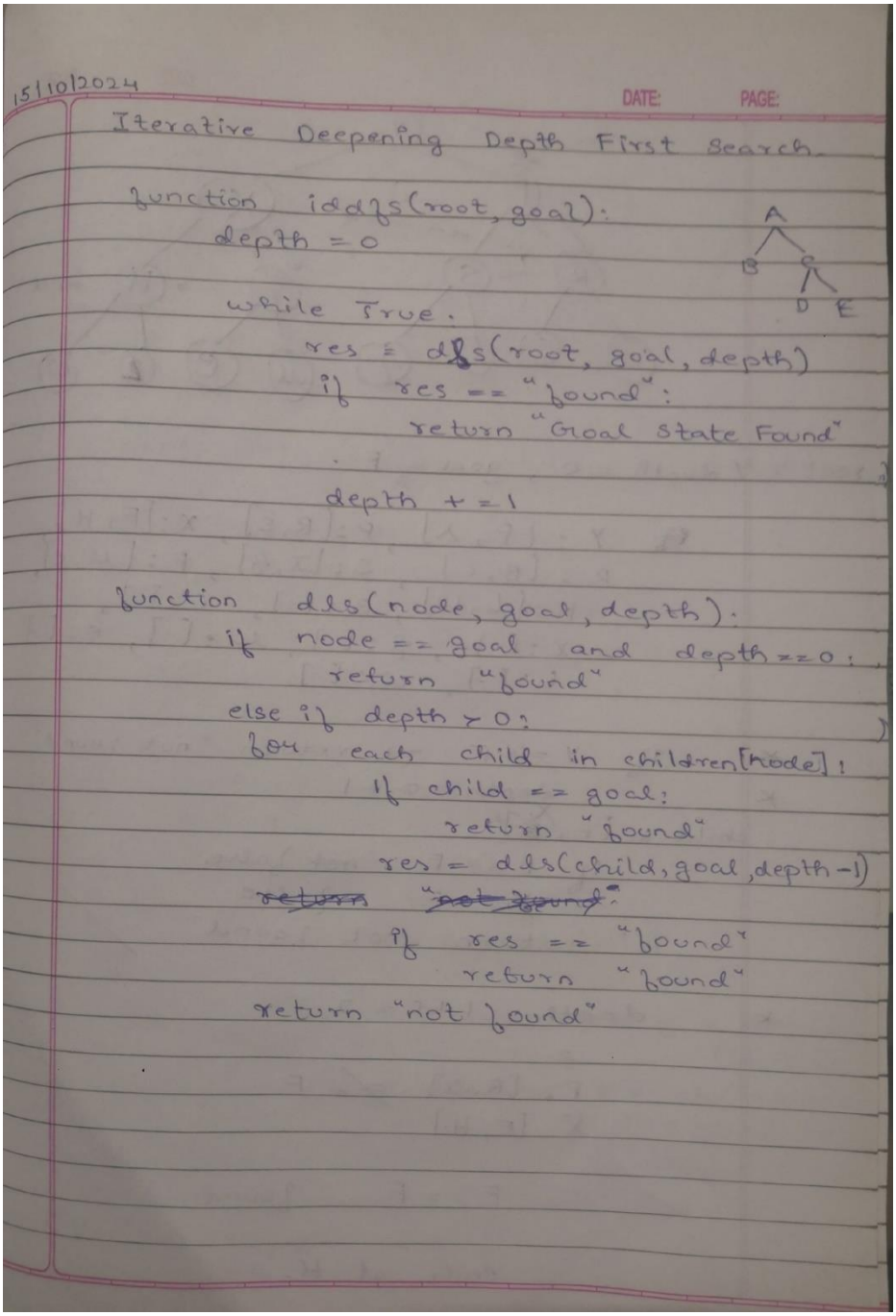
```
            new_state = make_move(current_state, move)
            new_path = path + [move]
            stack.append((new_state, new_path))
    return None  # No solution found
# Main function to start the puzzle game
if __name__ == "__main__":
    # Define the initial state
    initial_state = [
        [1, 2, 3],
        [4, 0, 5],
        [6, 7, 8]
    ]
    # Run DFS to solve the puzzle
    solution_moves = dfs_solve_puzzle(initial_state)
    if solution_moves:
        print("Solution found!")
        print(f"Moves: {' -> '.join(solution_moves)}")
    else:
        print("No solution exists for this puzzle.")
```

Output:

```
Solution found!
Moves: right -> down -> left -> left -> up -> right -> down -> right -> up -> left -> left -> down -> right -> right
```

Implement Iterative deepening search algorithm.

Algorithm:



Code:

```
# Define a simple Node class for the tree structure
class Node:
    def __init__(self, value):
        self.value = value
        self.children = []
# Iterative Deepening Depth-First Search (IDDFS)
def iddfs(root, goal):
    depth = 0
    while True:
        print(f"Searching at depth {depth}...")
        result = dls(root, goal, depth)
        if result == "found":
            return "Goal State Found"
        depth += 1
# Depth-Limited Search (DLS)
def dls(node, goal, depth):
    if depth == 0:
        if node.value == goal:
            return "found"
        else:
            return "not found"
    elif depth > 0:
        for child in node.children:
            result = dls(child, goal, depth - 1)
            if result == "found":
                return "found"
    return "not found"
# Build the tree (graph)
root = Node('Y')
p = Node('P')
X = Node('X')
r = Node('R')
S = Node('S')
F = Node('F')
H = Node('H')
```

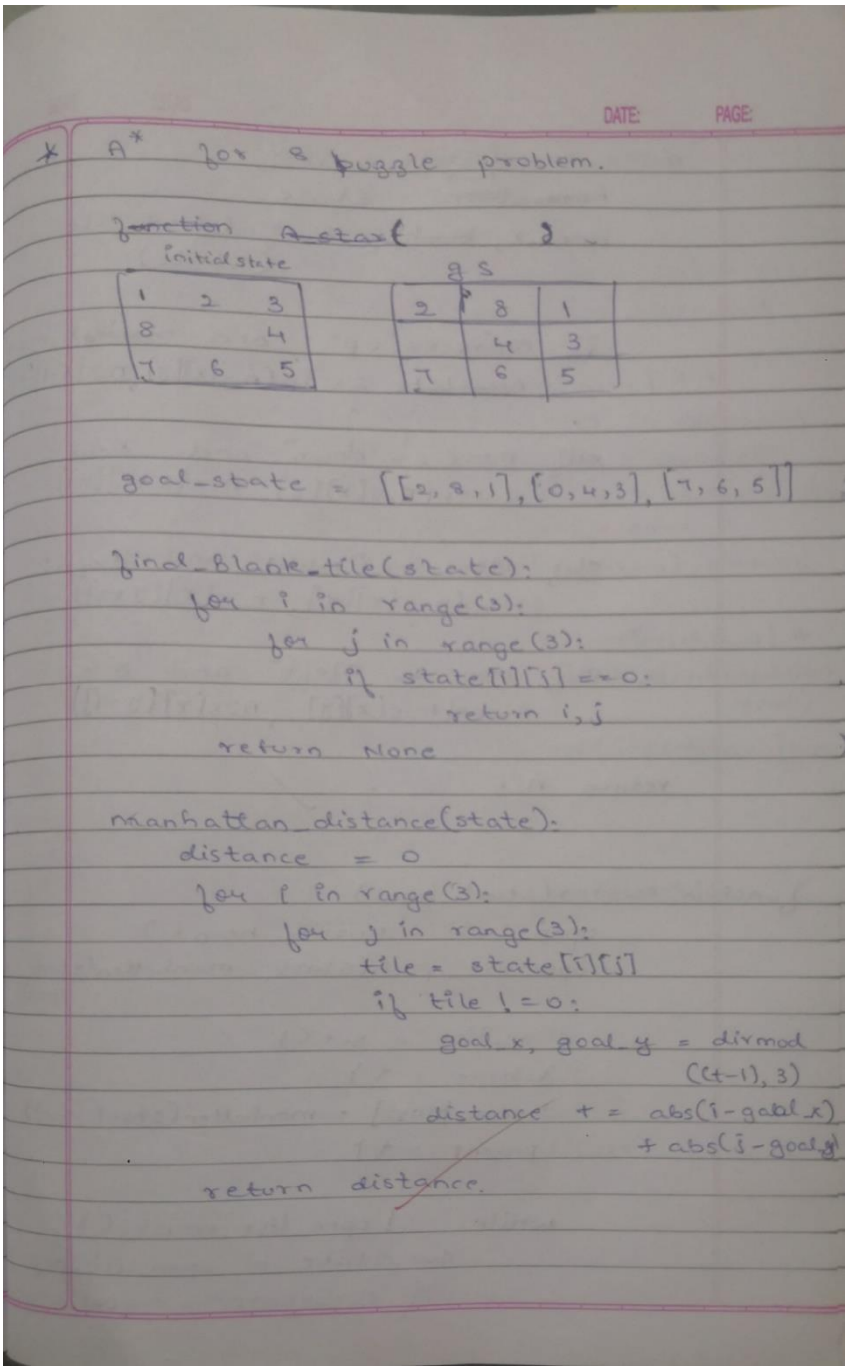
```
B = Node('B')
C = Node('C')
Z = Node('Z')
U = Node('U')
E = Node('E')
L = Node('L')
W = Node('W')
x = Node('x')
# Connect the nodes (edges)
root.children = [p, X]
p.children = [r, S]
X.children = [F, H]
r.children = [B, C]

S.children = [x, Z]
p.children = [r, S]
X.children = [F, H]
F.children = [U, E]
H.children = [L, W]


# Define the goal state
goal = 'F'

# Run IDDFS
result = iddfs(root, goal)
print(result)
```

Output:

```
Searching at depth 0...
Searching at depth 1...
Searching at depth 2...
Goal State Found
```

Implement A* search algorithm.

Algorithm:



A* for 8 puzzle problem.

function A-star( )

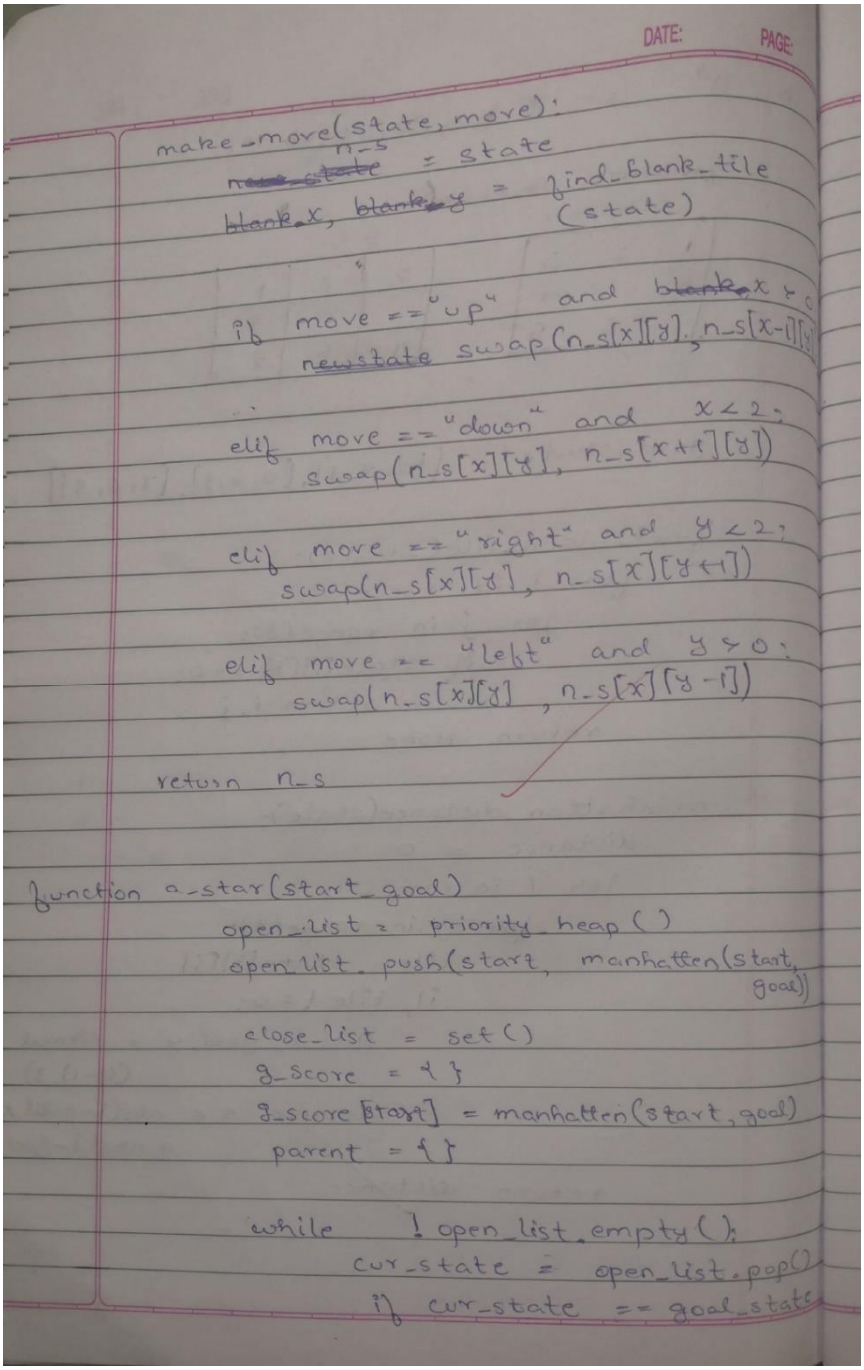
initial state

1	2	3
8		4
7	6	5

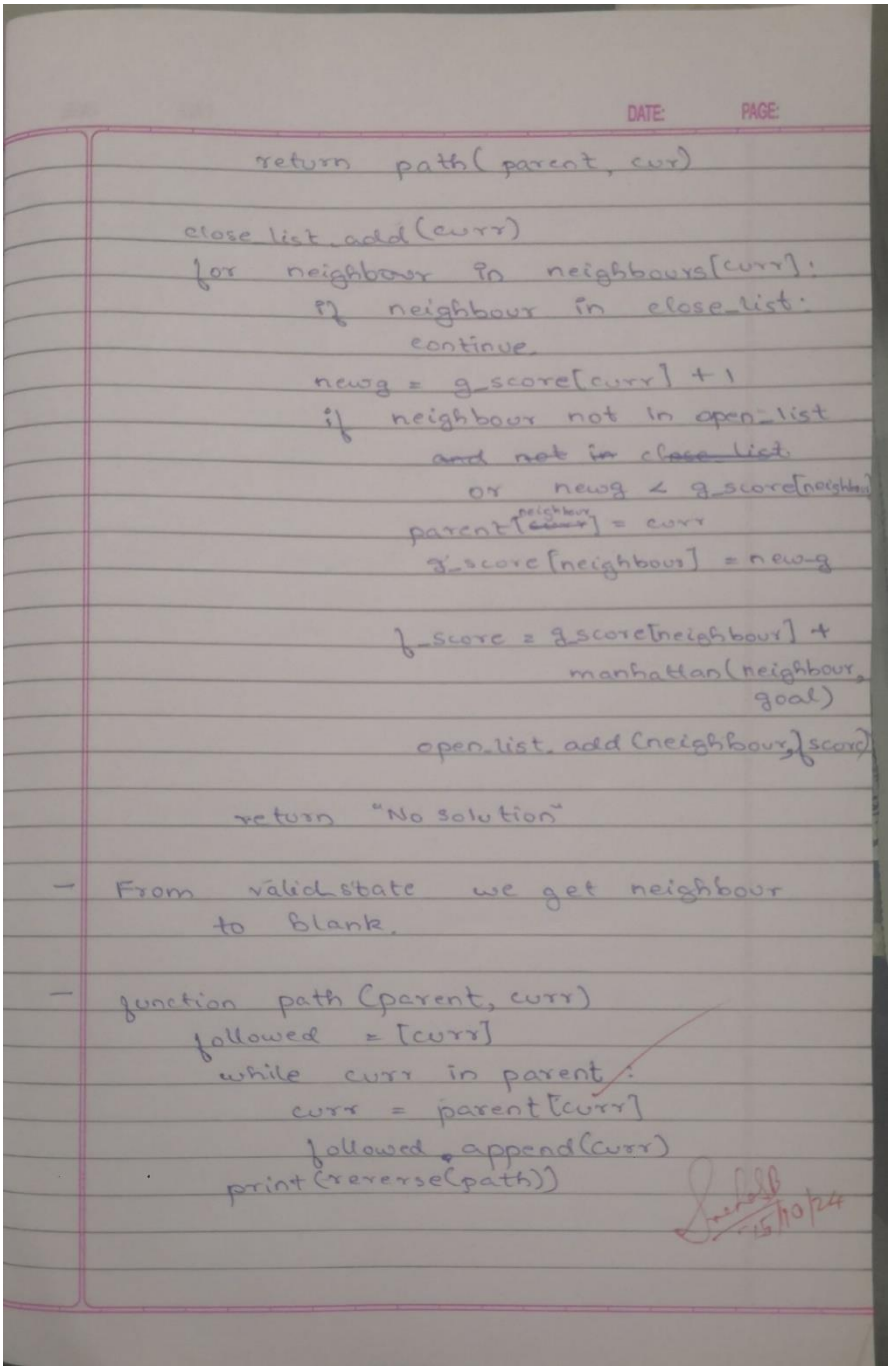
g s

2	8	1
	4	3
7	6	5

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

find_Blank_tile(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return None

manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            tile = state[i][j]
            if tile != 0:
                goal_x, goal_y = divmod
                                 ((t-1), 3)
                distance += abs(i-goal_x)
                          + abs(j-goal_y)
    return distance.
```



```
DATE:  PAGE:

make_move(state, move):
    n_s = state
    blank_x, blank_y = find_blank_tile(state)

    if move == "up" and blank_x > 0:
        newstate swap(n_s[x][y], n_s[x-1][y])

    elif move == "down" and x < 2:
        swap(n_s[x][y], n_s[x+1][y])

    elif move == "right" and y < 2:
        swap(n_s[x][y], n_s[x][y+1])

    elif move == "left" and y > 0:
        swap(n_s[x][y], n_s[x][y-1])

    return n_s

function a_star(start_goal)
    open_list = priority_heap()
    open_list.push(start, manhatten(start, goal))

    close_list = set()
    g_score = {}
    g_score[start] = manhatten(start, goal)
    parent = {}

    while ! open_list.empty():
        cur_state = open_list.pop()
        if cur_state == goal_state:
```



DATE: PAGE:

```
return path(parent, curr)

close_list.add(curr)
for neighbour in neighbours[curr]:
    if neighbour in close_list:
        continue
    newg = g_score[curr] + 1
    if neighbour not in open_list
        and not in close_list:
        or newg < g_score[neighbour]
    parent[neighbour] = curr
    g_score[neighbour] = newg

    f_score = g_score[neighbour] +
              manhattan(neighbour,
                        goal)

    open_list.add(neighbour, fscore)

return "No solution"
```

- From valid state we get neighbour to blank.

- function path(parent, curr)
  followed = [curr]
  while curr in parent:
  curr = parent[curr]
  followed.append(curr)
  print(reverse(path))

15/10/24

Code:

```
import heapq
from copy import deepcopy

# Define the goal state
goal_state = [
    [2, 8, 1],
    [0, 4, 3],
    [7, 6, 5]
]

# Function to find the position of the blank tile (0)
def find_blank_tile(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return None

# Function to calculate the Manhattan distance
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            tile = state[i][j]
            if tile != 0:  # Ignore the blank tile
                goal_x, goal_y = divmod(tile - 1, 3)
                distance += abs(goal_x - i) + abs(goal_y - j)
    return distance

# Function to make a move by sliding the blank tile (0)
def make_move(state, move):
    new_state = deepcopy(state)
    blank_x, blank_y = find_blank_tile(state)

    if move == "up" and blank_x > 0:
        new_state[blank_x][blank_y], new_state[blank_x - 1][blank_y] = new_state[blank_x - 1][blank_y], new_state[blank_x][blank_y]
    elif move == "down" and blank_x < 2:
        new_state[blank_x][blank_y], new_state[blank_x + 1][blank_y] = new_state[blank_x + 1][blank_y], new_state[blank_x][blank_y]
    elif move == "left" and blank_y > 0:
        new_state[blank_x][blank_y], new_state[blank_x][blank_y - 1] = new_state[blank_x][blank_y - 1], new_state[blank_x][blank_y]
    elif move == "right" and blank_y < 2:
```

```
        new_state[blank_x][blank_y], new_state[blank_x][blank_y + 1] = new_state[blank_x][blank_y + 1], new_state[blank_x][blank_y]

    return new_state

# Function to get valid moves for the blank tile (0)
def get_valid_moves(state):
    blank_x, blank_y = find_blank_tile(state)
    moves = []
    if blank_x > 0:
        moves.append("up")
    if blank_x < 2:
        moves.append("down")
    if blank_y > 0:
        moves.append("left")
    if blank_y < 2:
        moves.append("right")
    return moves


def astar_solve_puzzle(initial_state):
    """Solves the 8-puzzle using A* search with Manhattan distance heuristic."""

    open_list = [(manhattan_distance(initial_state), initial_state, [])]  # (f, state, path)
    closed_list = set()

    while open_list:

        # Extracting last tuple and destructuring into three variables
        _, current_state, path = heapq.heappop(open_list)

        # tuple of tuples
        state_tuple = tuple(map(tuple, current_state))

        # checking
        if current_state == goal_state:
            return path


        # set of tuples
        if state_tuple in closed_list:
            continue
        closed_list.add(state_tuple)

        for move in get_valid_moves(current_state):
```

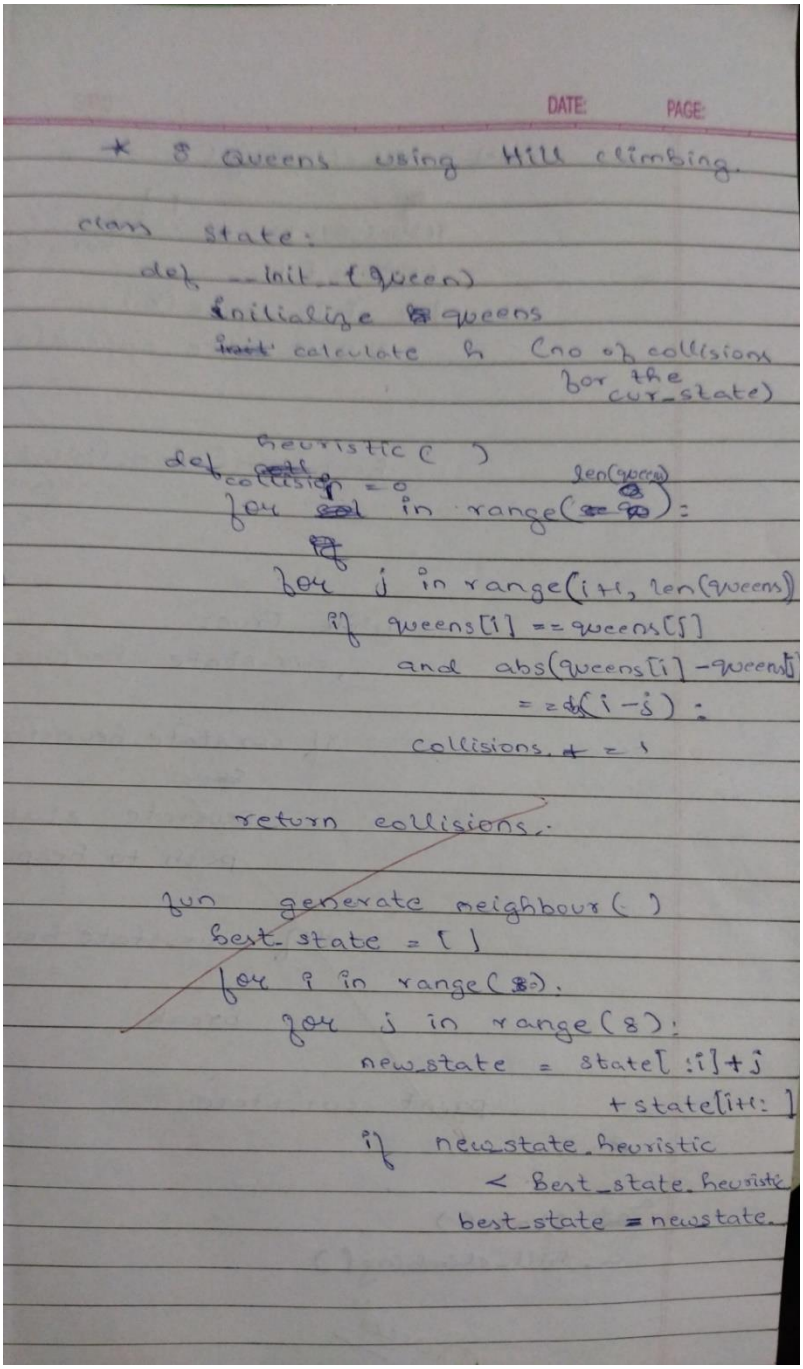
```
            # making moves to all possible places
            new_state = make_move(current_state, move)

            # appending to new path to list
            new_path = path + [move]

            # calculating new distance
            f_value = manhattan_distance(new_state) + len(new_path)  # f = g + h

            # by doing this we get min distance state,  which will be used for next state.
            heapq.heappush(open_list, (f_value, new_state, new_path))

    return None  # No solution found



# Main function
if __name__ == "__main__":
    # ... (your initial state definition) ...
    # Define the initial state
    initial_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]

    # Run A* search using Manhattan distance
    solution_moves = astar_solve_puzzle(initial_state)

    if solution_moves:
        print("Solution found!")
        print(f"Moves: {' -> '.join(solution_moves)}")
    else:
        # if len(solution_moves) == 0:
        #   print("ALready Won!")
        # else:
          print("No solution exists for this puzzle.")
```

Output:

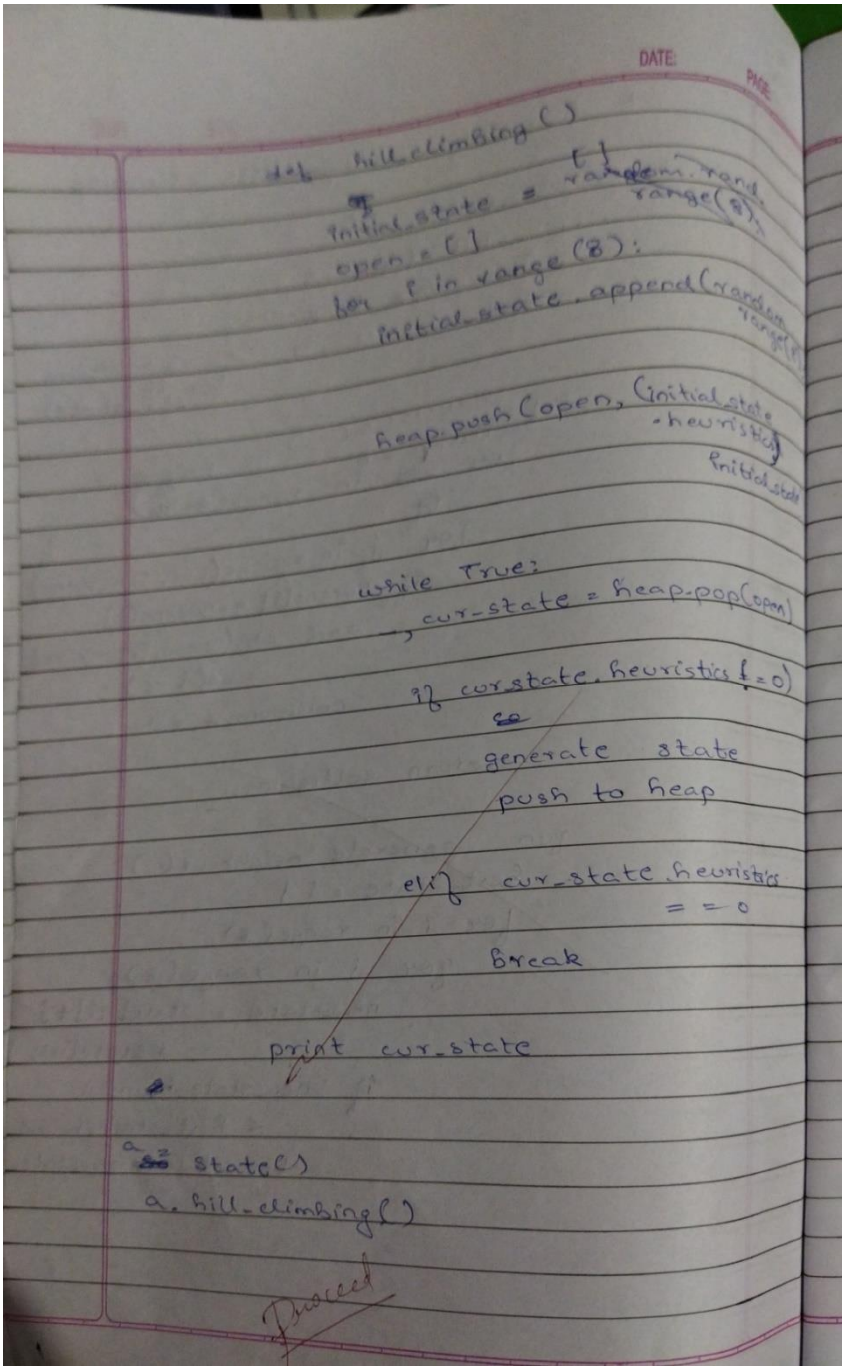
```
Solution found!
Moves: up -> left -> down -> right -> right -> up -> left -> left -> down
```

Implement Hill Climbing search algorithm to solve N-Queens problem.

Algorithm:

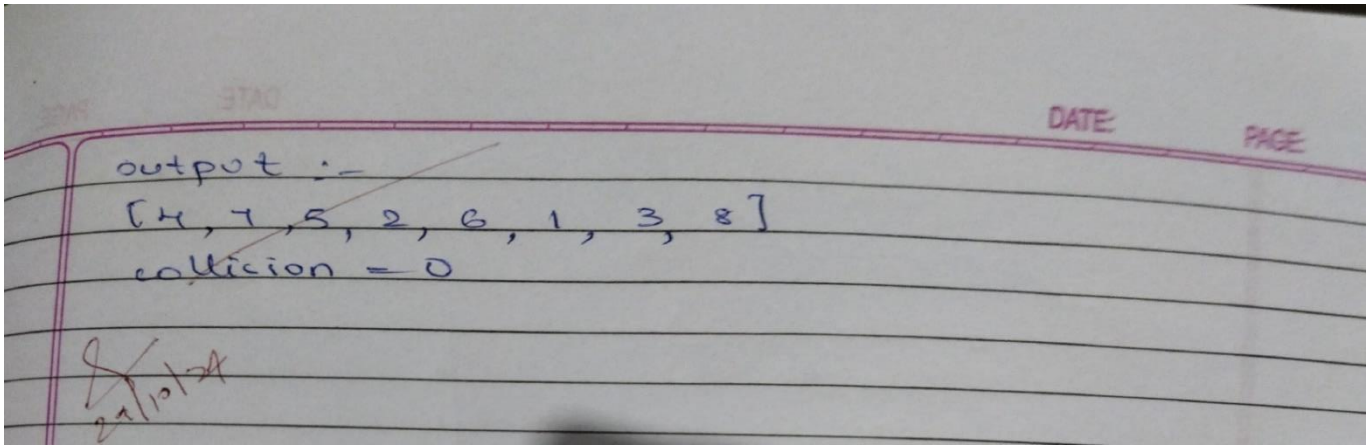


```
DATE:  PAGE:
* 8 Queens using Hill climbing.

class State:
    def __init__(queen)
        initialize 8 queens
        calculate h (no of collisions for the cur_state)

    def heuristic( ):
        collision = 0
        for i in range(len(queens)):
            for j in range(i+1, len(queens)):
                if queens[i] == queens[j]
                    and abs(queens[i] - queens[j])
                        == abs(i - j):
                    collisions += 1

        return collisions.

fun generate neighbour( )
    best_state = [ ]
    for i in range(8):
        for j in range(8):
            new_state = state[:i] + j
                + state[i+1:]
            if new_state.heuristic
                < best_state.heuristic
                best_state = newstate.
```



```
def hill_climbing():
    initial_state = [random.randint(0, 7) for _ in range(8)]
    open = []
    for i in range(8):
        initial_state.append(random.randint(0, 7))

    heap_push(open, (initial_state.heuristics, initial_state))

    while True:
        cur_state = heap_pop(open)

        if cur_state.heuristics != 0:
            generate state
            push to heap

        elif cur_state.heuristics == 0:
            break

    print cur_state

state()
hill_climbing()
```

Proceed

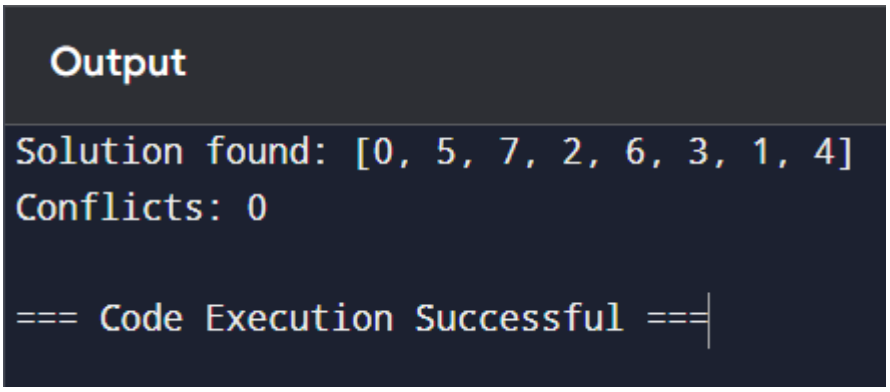

DATE:
PAGE:
output :-
[4, 7, 5, 2, 6, 1, 3, 8]
collision = 0
29/10/24

Code:

```
## hill climbing
import random

# Define the board state with queens' positions
class State:
    def __init__(self, queens):
        self.queens = queens
        self.h = self.heuristic()  # Calculate initial conflicts (heuristic)

    def heuristic(self):
        """Calculate the number of conflicts."""
        conflicts = 0
        for i in range(len(self.queens)):
            for j in range(i + 1, len(self.queens)):
                # Check for conflicts in same column or diagonals
                if self.queens[i] == self.queens[j] or abs(self.queens[i] - self.queens[j]) == abs(i - j):
                    conflicts += 1
        return conflicts

    def generate_neighbors(self):
        """Create new board arrangements by moving each queen in its row."""
        neighbors = []
        for row in range(len(self.queens)):
            for col in range(8):  # Try each column for each queen
                if col != self.queens[row]:  # Move only if it's a new position
                    new_queens = self.queens[:]
                    new_queens[row] = col
                    neighbors.append(State(new_queens))
        return neighbors

# Hill climbing to find a solution
def hill_climbing(initial_state):
    current_state = initial_state

    while True:
        neighbors = current_state.generate_neighbors()  # All possible next steps
        next_state = min(neighbors, key=lambda s: s.h)  # Pick neighbor with least conflicts

        if next_state.h >= current_state.h:  # If no better neighbor, stop
            break
        current_state = next_state  # Move to better neighbor

    return current_state  # Return the found state

# Run the hill-climbing algorithm
```

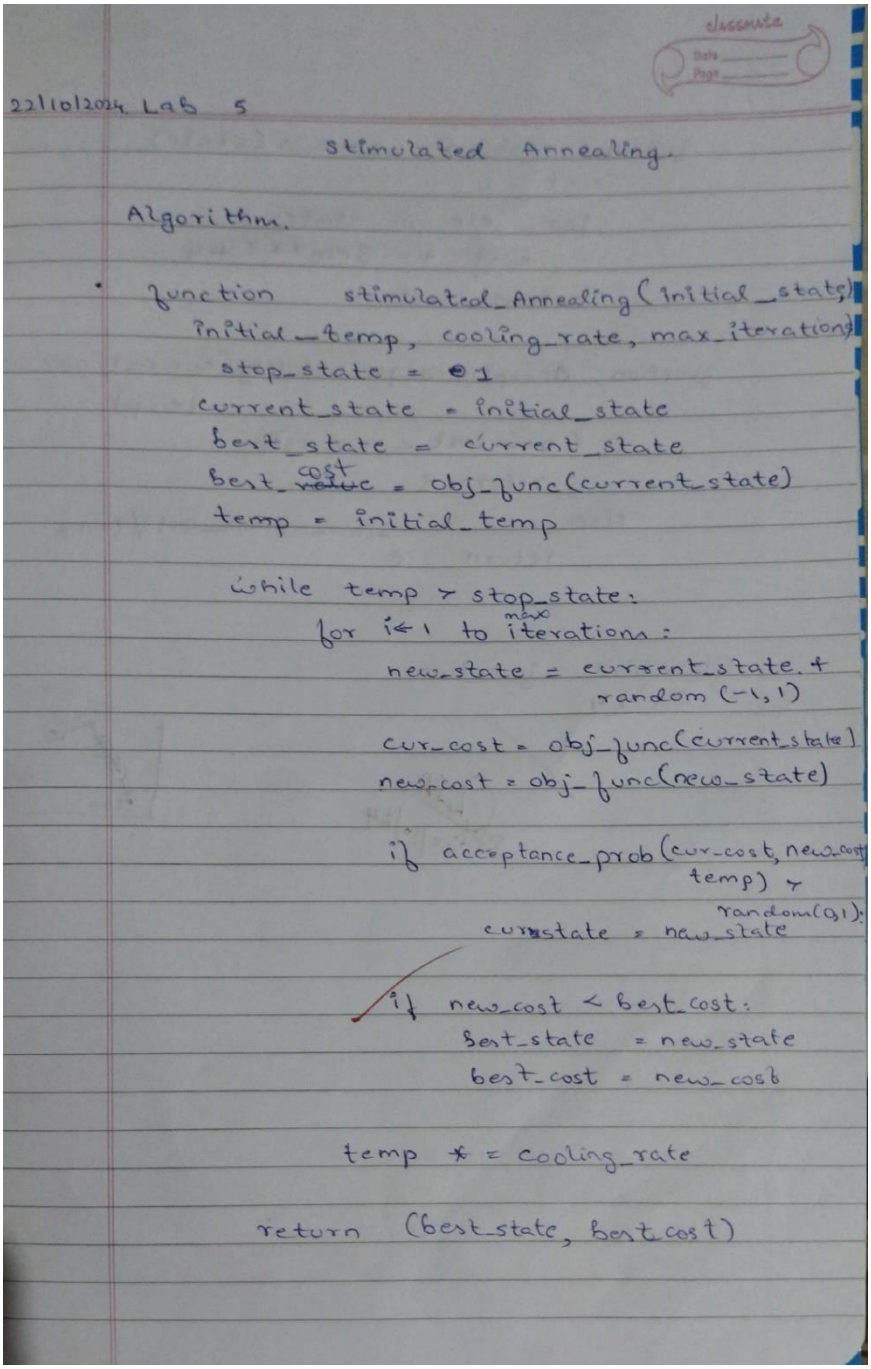
```
initial_queens = [random.randint(0, 7) for _ in range(8)]  # Random starting position
initial_state = State(initial_queens)
solution = hill_climbing(initial_state)

# Output the solution
print("Solution found:", solution.queens)
print("Conflicts:", solution.h)
```

Output:



Implement Simulated Annealing:

Algorithm:



22/10/2024 Lab 5

Stimulated Annealing.

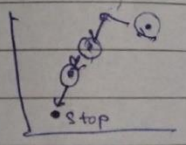
Algorithm.

```
function stimulated_Annealing(initial_state,
    initial_temp, cooling_rate, max_iterations)
    stop_state = 1
    current_state = initial_state
    best_state = current_state
    best_cost = obj_func(current_state)
    temp = initial_temp

    while temp > stop_state:
        for i<1 to max iterations:
            new_state = current_state +
                        random(-1,1)
            cur_cost = obj_func(current_state)
            new_cost = obj_func(new_state)

            if acceptance_prob(cur_cost, new_cost,
                               temp) >
                               random(0,1):
                current state = new_state

            if new_cost < best_cost:
                best_state = new_state
                best_cost = new_cost

        temp *= cooling_rate

    return (best_state, best_cost)
```

```
function objective_function(state)
    cost = 0
    for ele in state:
        cost += 3*ele**3 + 4
    return cost.

function Acceptance_probability(cur_cost, new_cost,
                                temp)
    if (new_cost < cur_cost):
        return 1
    else
        return e^(-(cur_cost - new_cost)/temp)
```

Objective function

$3x^3 + 4$



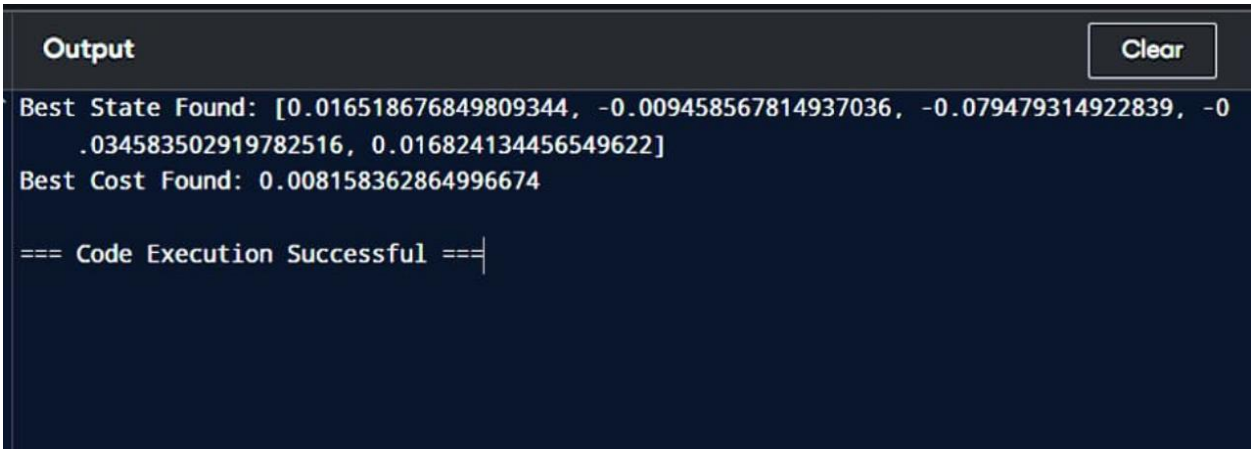
22/10/24

Code:

```
import math
import random

# Simulated Annealing function
def SA(initial_state, initial_temp, cooling_rate, max_iterations):
    stop_rate = 1  # Stop when temperature is low enough
    current_state = initial_state  # Initialize the current state
    current_cost = objective_function(current_state)  # Cost of the current state

    best_state = current_state  # Track the best state found
    best_cost = current_cost  # Track the best cost found

    temp = initial_temp  # Start with the initial temperature

    # Main loop for Simulated Annealing
    while temp > stop_rate:  # Stop when temperature is lower than the stop rate
        for i in range(max_iterations):  # Iterate for a number of steps at each temperature level
            # Generate a new state by perturbing the current state slightly
            new_state = [x + random.uniform(-1, 1) for x in current_state]
            new_cost = objective_function(new_state)  # Calculate cost of the new state

            # Acceptance probability calculation
            if acceptance_probablity(current_cost, new_cost, temp) > random.random():
                current_state = new_state  # Move to the new state
                current_cost = new_cost  # Update the current cost

            # Update the best state found so far
            if new_cost < best_cost:
                best_state = new_state
                best_cost = new_cost

        # Cool down the temperature
        temp *= cooling_rate  # Reduce the temperature by the cooling rate

    return best_state, best_cost

# Objective function to minimize
def objective_function(state):
    cost = 0
    for x in state:
        cost += x**2   # Cubic function for each element in the state
    return cost

# Acceptance probability calculation
def acceptance_probablity(current_cost, new_cost, temp):
```

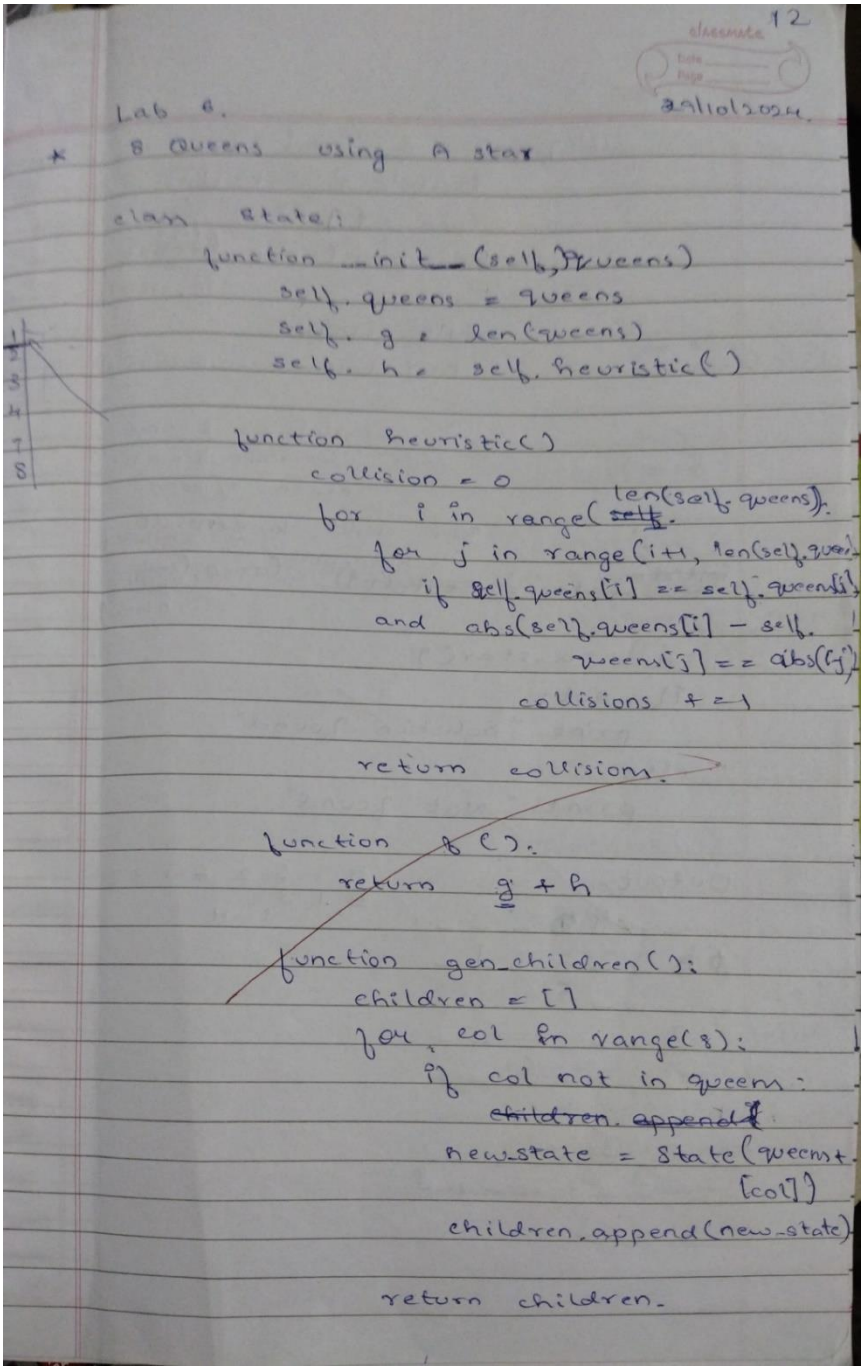
```
    if new_cost < current_cost:
        return 1  # Always accept if the new cost is better (lower)
    else:
        # Accept worse solutions with a probability depending on temperature
        return math.exp((current_cost - new_cost) / temp)

# Example execution with initial state and parameters
initial_state = [1,2,3, 4, 5]  # Example list of values as the initial state
initial_temp = 1000  # High starting temperature
cooling_rate = 0.99  # Cooling rate
max_iterations = 100  # Maximum iterations at each temperature level

# Run the Simulated Annealing algorithm
best_state, best_cost = SA(initial_state, initial_temp, cooling_rate, max_iterations)

# Output the results
print("Best State Found:", best_state)
print("Best Cost Found:", best_cost)
```

Output:



Implement A star algorithm to solve N-Queens problem.

Algorithm:

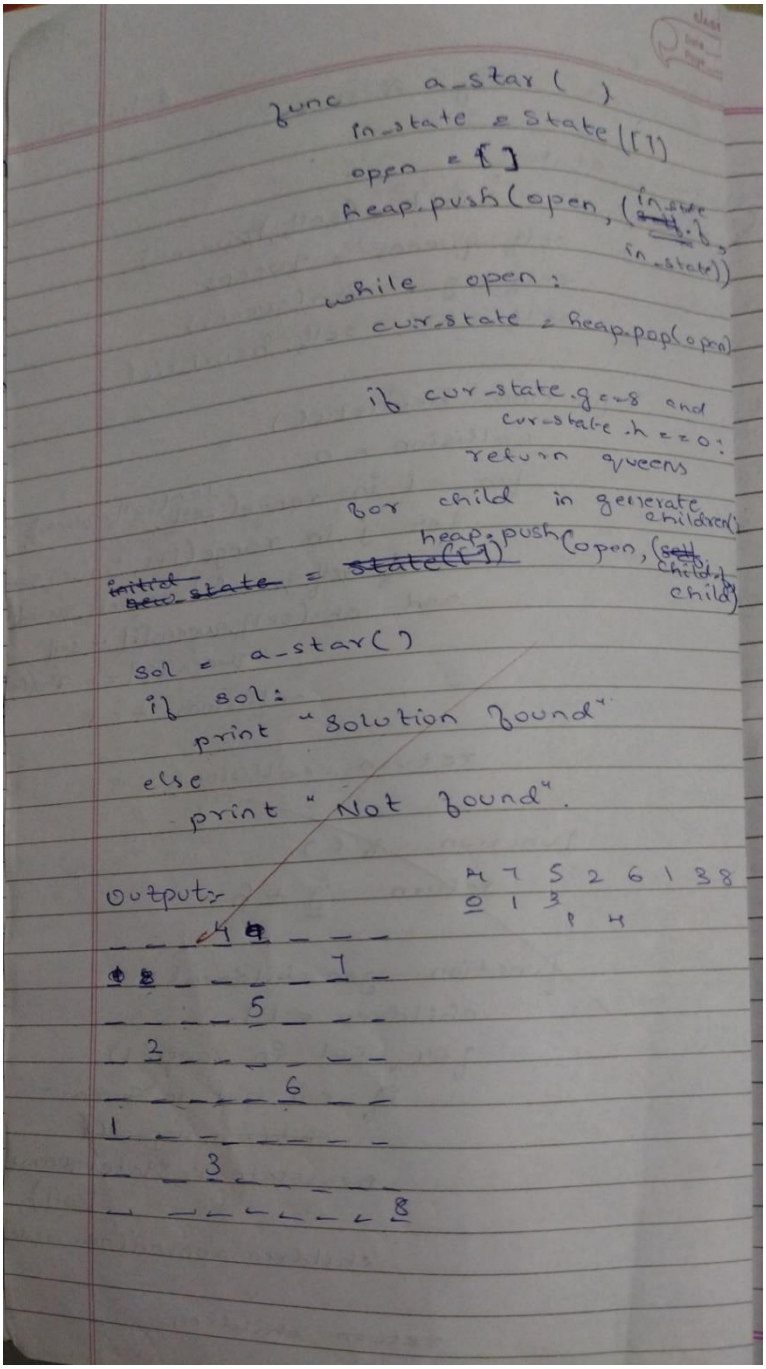


Lab 8.

29/10/2024.

* 8 Queens using A star

```
class State:
    function __init__(self, queens)
        self.queens = queens
        self.g = len(queens)
        self.h = self.heuristic()

    function heuristic()
        collision = 0
        for i in range(len(self.queens)):
            for j in range(i+1, len(self.queens)
                if self.queens[i] == self.queens[j]
                and abs(self.queens[i] - self.
                        queens[j] == abs(i-j)
                    collisions += 1

        return collisions.

    function f():
        return g + h

    function gen_children():
        children = []
        for col in range(8):
            if col not in queens:
                children.append()
            new_state = State(queens+
                              [col])
            children.append(new_state)

        return children.
```



Code:

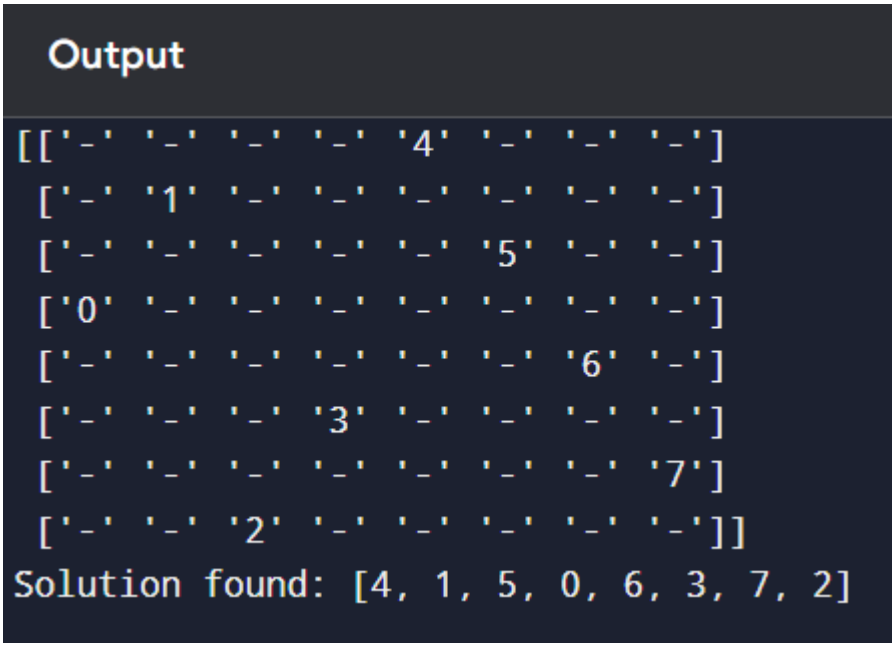
```
import heapq
import numpy as np
# Define the state as the current board configuration
```

```
class State:
    def __init__(self, queens):
        self.queens = queens   # List of queen positions in each row
        self.g = len(queens)   # Number of queens placed (cost)
        self.h = self.heuristic()  # Estimated conflicts (heuristic)

    def heuristic(self):
        conflicts = 0
        for i in range(len(self.queens)):
            for j in range(i + 1, len(self.queens)):
                # Check for conflicts (same column or diagonal)
                if self.queens[i] == self.queens[j] or abs(self.queens[i] - self.queens[j]) == abs(i - j):
                    conflicts += 1
        return conflicts

    def f(self):
        return self.g + self.h   # A* cost function f(n) = g(n) + h(n)

    # Define the less-than method for priority queue comparison
    def __lt__(self, other):
        return self.f() < other.f()

    def generate_children(self):
        children = []
        row = len(self.queens)
        for col in range(8):   # Try placing queen in each column
            if col not in self.queens:  # Avoid placing in the same column
```

```
            new_state = State(self.queens + [col])  # Add queen
            children.append(new_state)
        return children

# A* search for the 8-queens solution
def a_star_search():
    initial_state = State([])  # Start with no queens placed
    open_set = []
    heapq.heappush(open_set, (initial_state.f(), initial_state))

    while open_set:
        _, current_state = heapq.heappop(open_set)

        # Check if the goal is reached
        if current_state.g == 8 and current_state.h == 0:
            return current_state.queens  # Solution found

        # Generate children and add to open set
        for child in current_state.generate_children():
            heapq.heappush(open_set, (child.f(), child))

    return None  # No solution found

# Run the algorithm
solution = a_star_search()
if solution:
    a = [["-"] * 8 for i in range(8)]
```

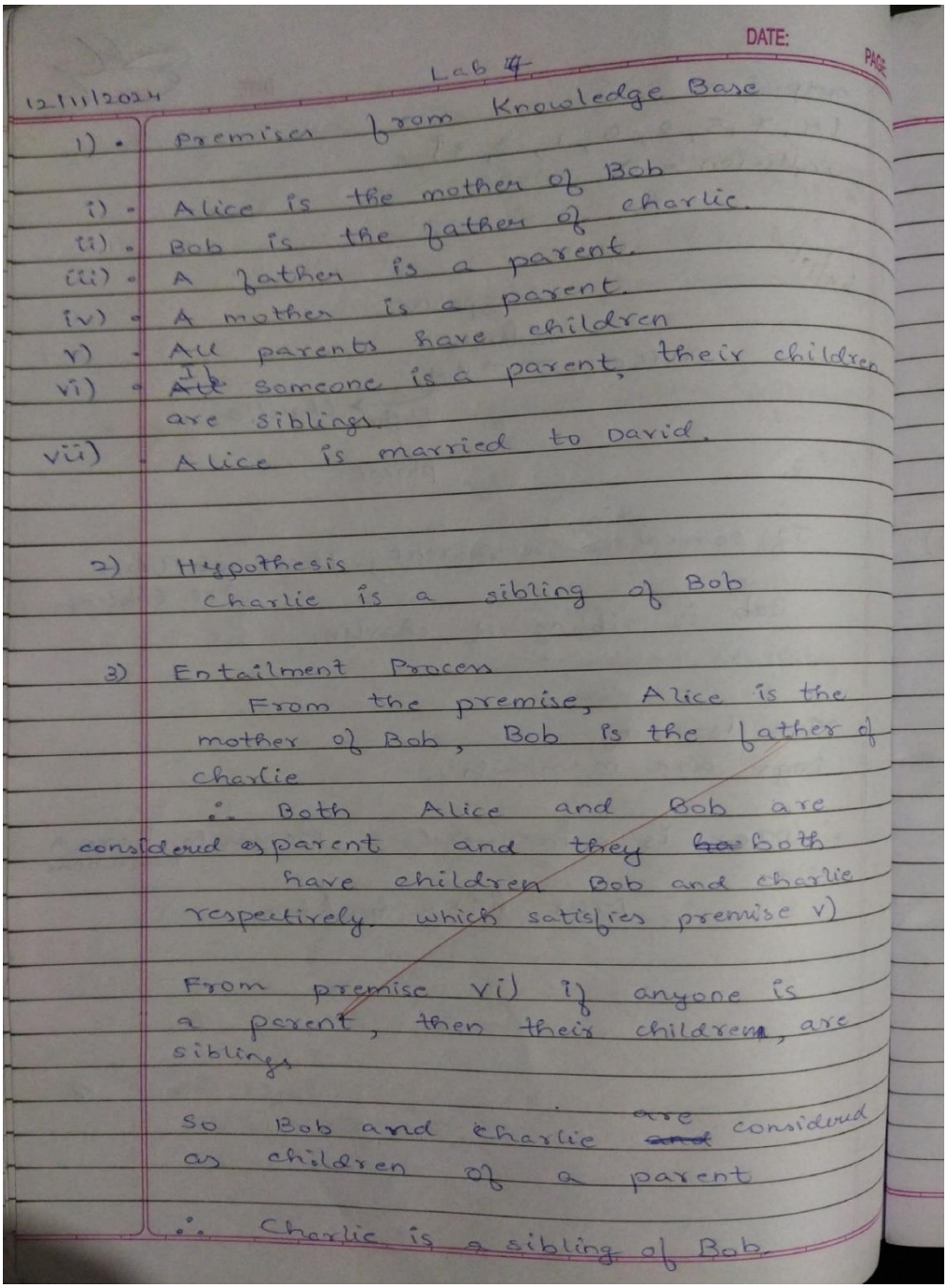
```
    for r in range(8):
        for c in range(8):
            if c == solution[r]:
                a[r][c] = solution[r]
    print(np.array(a))
    print("Solution found:", solution)
else:
    print("No solution found")
```

Output:



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

Algorithm:



Lab 4

DATE:

12/11/2024

1) Premises from Knowledge Base

i) Alice is the mother of Bob

ii) Bob is the father of Charlie.

iii) A father is a parent.

iv) A mother is a parent.

v) All parents have children

vi) If someone is a parent, their children are siblings.

vii) Alice is married to David.

2) Hypothesis

Charlie is a sibling of Bob

3) Entailment Process

From the premise, Alice is the mother of Bob, Bob is the father of Charlie

∴ Both Alice and Bob are considered as parent and they both have children Bob and Charlie respectively. which satisfies premise v)

From premise vi) if anyone is a parent, then their children, are siblings

So Bob and Charlie are considered as children of a parent

∴ Charlie is a sibling of Bob.

Code:

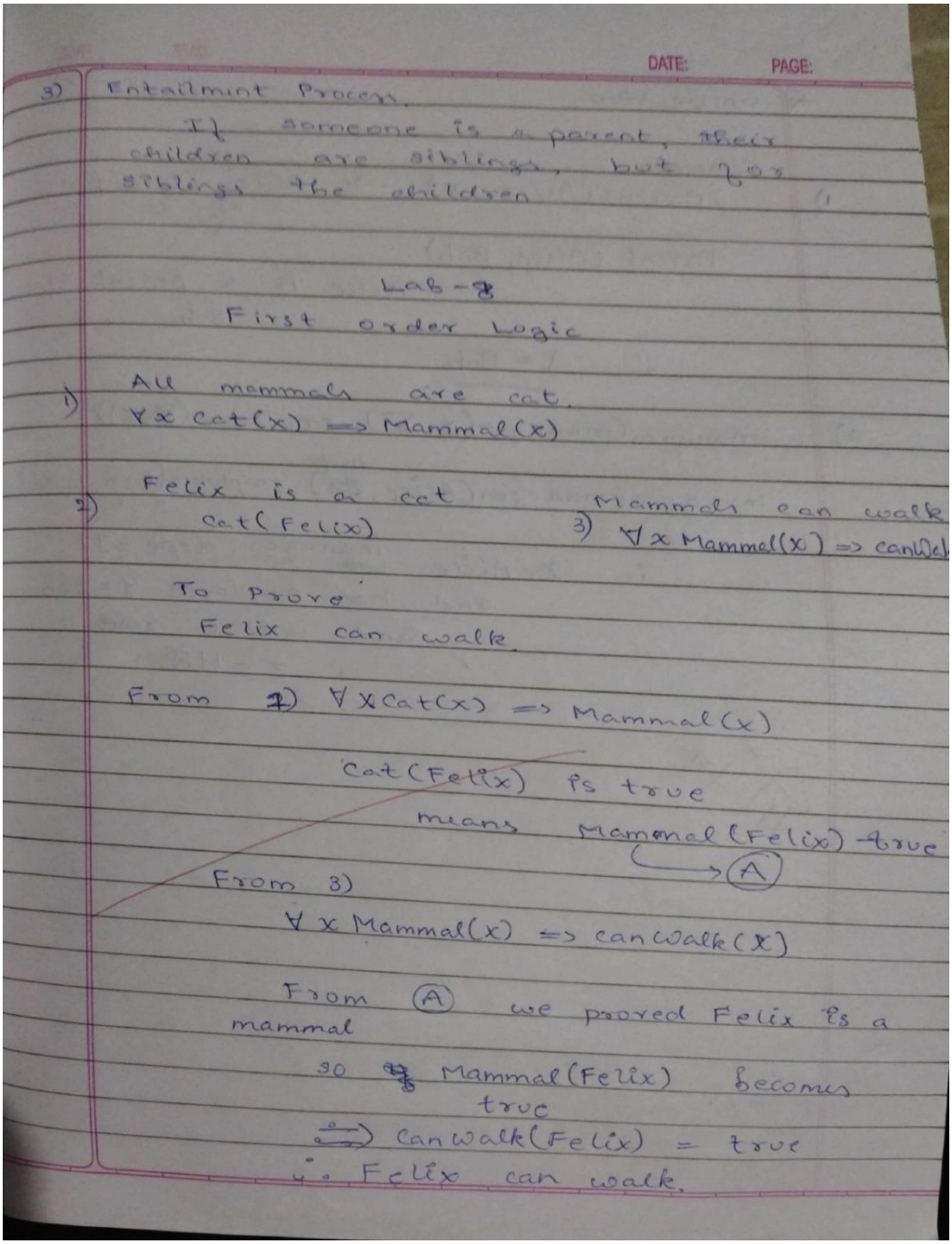
```
from sympy.logic.boolalg import Or, And, Not
from sympy.abc import A, B, C, D, E, F
from sympy import simplify_logic

def is_entailment(kb, query):

    # Negate the query
    negated_query = Not(query)

    # Add negated query to the knowledge base
    kb_with_negated_query = And(*kb, negated_query)

    # Simplify the combined KB to CNF
    simplified_kb = simplify_logic(kb_with_negated_query, form="cnf")

    # If the simplified KB evaluates to False, the query is entailed
    return simplified_kb == False


# Define a larger Knowledge Base
kb = [
    Or(A, B),       # A ∨ B
    Or(Not(A), C),   # ¬A ∨ C
    Or(Not(B), D),   # ¬B ∨ D
    Or(Not(D), E),   # ¬D ∨ E
    Or(Not(E), F),   # ¬E ∨ F
    F           # F
]
# Query to check
query = Or(C, F) # C ∨ F

# Check entailment
result = is_entailment(kb, query)
print(f"Is the query '{query}' entailed by the knowledge base? {'Yes' if result else 'No'}")
```

Output:

```
Is the query 'C | F' entailed by the knowledge base? Yes
```

Implement unification in first order logic.

Algorithm:



Entailment Process.

If someone is a parent, their children are siblings, but for siblings the children

Lab - 7

First order Logic

1) All mammals are cat.
$\forall x \; Cat(x) \Rightarrow Mammal(x)$

2) Felix is a cat
Cat(Felix)

Mammals can walk
3) $\forall x \; Mammal(x) \Rightarrow canWalk$

To Prove
Felix can walk.

From 1) $\forall x \; Cat(x) \Rightarrow Mammal(x)$

Cat(Felix) is true
means Mammal(Felix) true → (A)

From 3)
$\forall x \; Mammal(x) \Rightarrow canWalk(x)$

From (A) we proved Felix is a mammal

so Mammal(Felix) becomes true
$\Rightarrow$ canWalk(Felix) = true
∴ Felix can walk.

DATE: PAGE:

* Unification

Consider

1) Parent(Alice, x) → Alice is a parent of x

Parent (Alice, Bob)

↳ Alice is a parent of Bob

∴ x = Bob

2) manages(manager(x, IT), employee(y, z))

manages(manager(Alice, dept), employee(Bob, HR))

∴ x = Alice manages dept = IT and has employee y = Bob and whose z = HR.

Code:

```
import re

def occurs_check(var, x):
    if var == x:
        return True
    elif isinstance(x, list):
        return any(occurs_check(var, xi) for xi in x)
    return False

def unify_var(var, x, subst):
    if var in subst:
        return unify(subst[var], x, subst)
    elif isinstance(x, (list, tuple)) and tuple(x) in subst:
        return unify(var, subst[tuple(x)], subst)
    elif occurs_check(var, x):
        return "FAILURE"
    else:
        subst[var] = tuple(x) if isinstance(x, list) else x
        return subst

def unify(x, y, subst=None):
    if subst is None:
        subst = {}
    if x == y:
        return subst
    elif isinstance(x, str) and x.islower():
        return unify_var(x, y, subst)
    elif isinstance(y, str) and y.islower():
        return unify_var(y, x, subst)
    elif isinstance(x, list) and isinstance(y, list):
        if len(x) != len(y):
            return "FAILURE"
        if x[0] != y[0]:
            return "FAILURE"
        for xi, yi in zip(x[1:], y[1:]):
            subst = unify(xi, yi, subst)
            if subst == "FAILURE":
                return "FAILURE"
        return subst
    else:
        return "FAILURE"

def unify_and_check(expr1, expr2):
    result = unify(expr1, expr2)
    if result == "FAILURE":
```

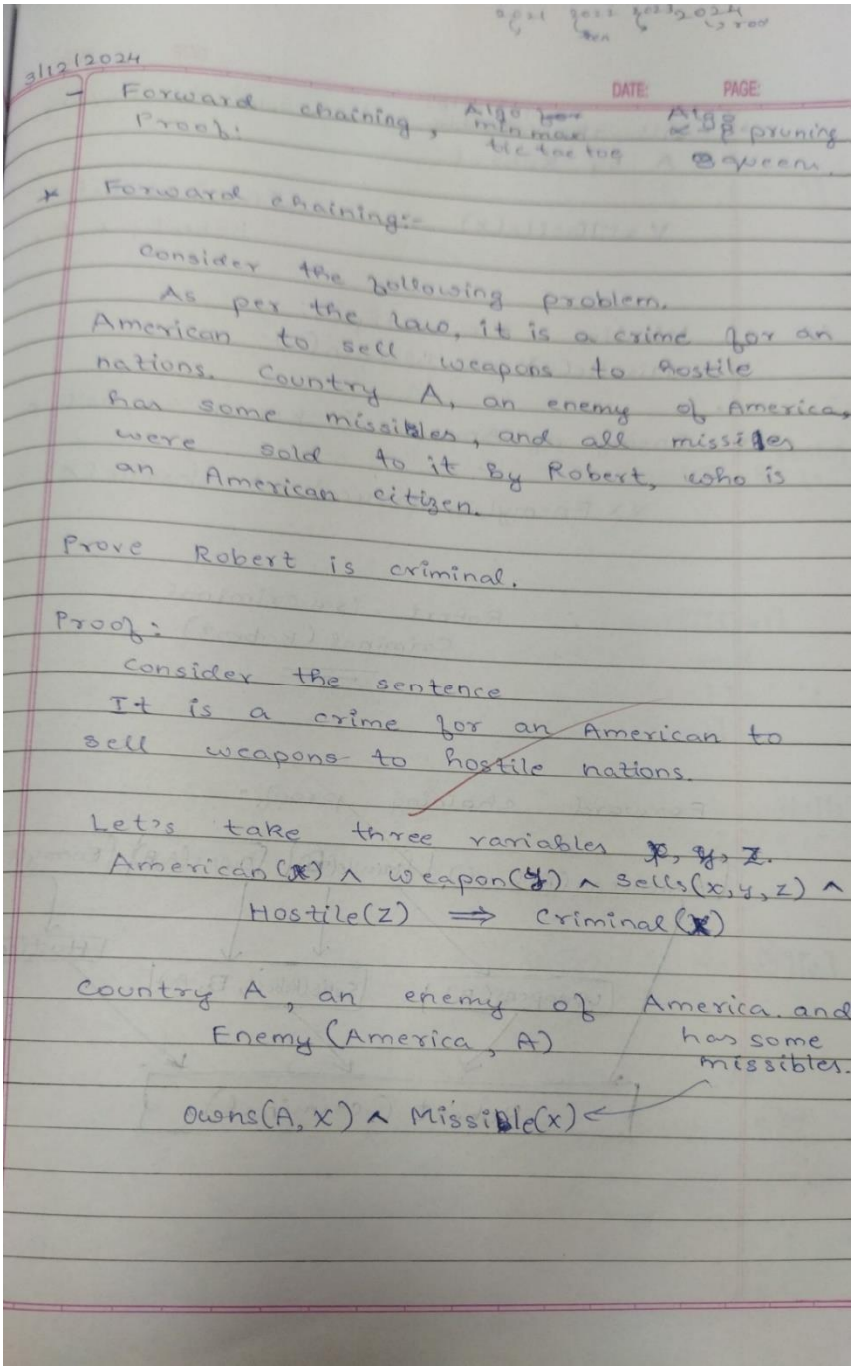
```
        return False, None
    return True, result

def display_result(expr1, expr2, is_unified, subst):
    print("Expression 1:", expr1)
    print("Expression 2:", expr2)
    if not is_unified:
        print("Result: Unification Failed")
    else:
        print("Result: Unification Successful")
        print("Substitutions:", {k: list(v) if isinstance(v, tuple) else v for k, v in subst.items()})

def parse_input(input_str):
    input_str = input_str.replace(" ", "")
    def parse_term(term):
        if '(' in term:
            match = re.match(r'([a-zA-Z0-9_]+)(.*)', term)
            if match:
                predicate = match.group(1)
                arguments_str = match.group(2)
                arguments = [parse_term(arg.strip()) for arg in arguments_str.split(',')]
                return [predicate] + arguments
        return term
    return parse_term(input_str)

def main():
    while True:
        expr1_input = input("Enter the first expression (e.g., p(x, f(y))): ")
        expr2_input = input("Enter the second expression (e.g., p(a, f(z))): ")
        expr1 = parse_input(expr1_input)
        expr2 = parse_input(expr2_input)
        is_unified, result = unify_and_check(expr1, expr2)
        display_result(expr1, expr2, is_unified, result)
        another_test = input("Do you want to test another pair of expressions? (yes/no): ").strip().lower()
        if another_test != 'yes':
            break

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

Output:

```
Enter the first expression (e.g., p(x, f(y))): Knows(f(Alice,Bob),g(z))
Enter the second expression (e.g., p(a, f(z))): Knows(f(x,y),g(x))
Expression 1: ['Knows', '(f(Alice', 'Bob)', ['g', '(z))']]
Expression 2: ['Knows', '(f(x', 'y)', ['g', '(x))']]
Result: Unification Successful
Substitutions: {'(f(x': '(f(Alice', 'y)': 'Bob)', '(z))': '(x))'}
Do you want to test another pair of expressions? (yes/no): yes
Output: 1BM22CS200
Enter the first expression (e.g., p(x, f(y))): A(x,y)
Enter the second expression (e.g., p(a, f(z))): A(Bob,Jack)
Expression 1: ['A', '(x', 'y)']
Expression 2: ['A', '(Bob', 'Jack)']
Result: Unification Successful
Substitutions: {'(x': '(Bob', 'y)': 'Jack)'}
Do you want to test another pair of expressions? (yes/no):
```

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

Algorithm:



9/12/2024

DATE: PAGE:

→ Forward chaining, Algo for min max tic tac toe Algo α-β pruning queen.

Proof:

* Forward chaining:-

Consider the following problem.

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 missibles, and all missibles were sold to it by Robert, who is an American citizen.

Prove Robert is criminal.

Proof:

Consider the sentence

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

Let's take three variables x, y, z.

American(x) ∧ weapon(y) ∧ sells(x,y,z) ∧ Hostile(z) ⟹ Criminal(x)

Country A, an enemy of America. and has some missibles.

Enemy(America, A)

Owns(A, x) ∧ Missible(x)

DATE: PAGE:

- All the ~~missible~~ missiles were sold to country A by Robert

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

- Missiles are weapons

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

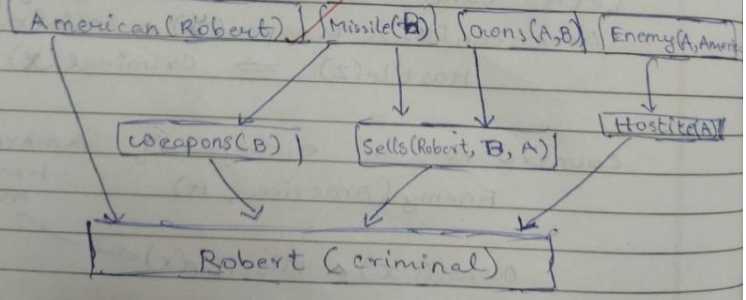
- Enemy of America is known as Hostile

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

$\therefore$ Robert is a criminal

Criminal(Robert)

Forward chaining proof:



Code:

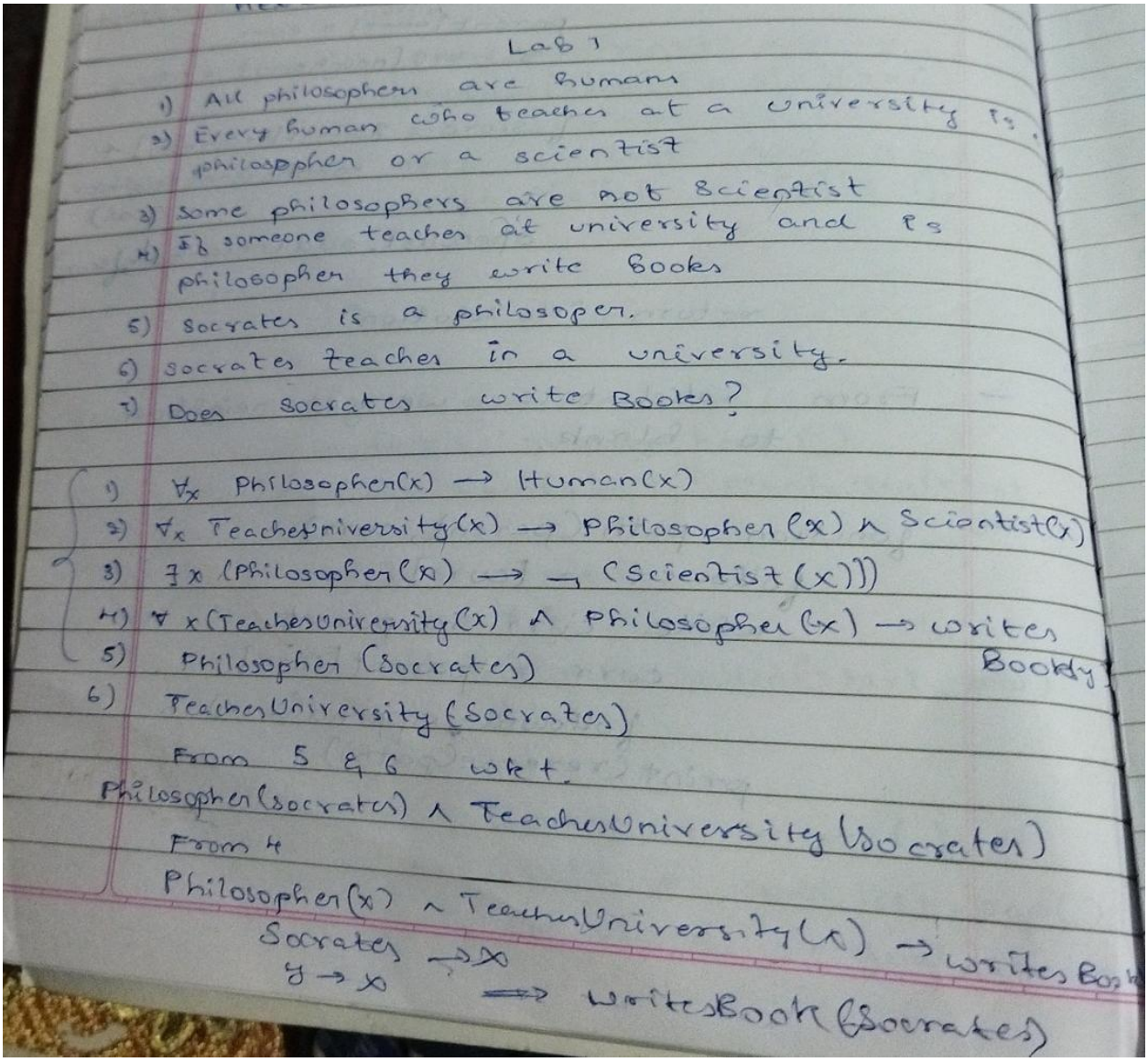
```
KB = set()

KB.add('American(Robert)')
KB.add('Enemy(America, A)')
KB.add('Missile(T1)')
KB.add('Owns(A, T1)')

def modus_ponens(fact1, fact2, conclusion):
    if fact1 in KB and fact2 in KB:
        KB.add(conclusion)
        print(f"Inferred: {conclusion}")

def forward_chaining():
    if 'Missile(T1)' in KB:
        KB.add('Weapon(T1)')
        print(f"Inferred: 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)")

    if 'Enemy(America, A)' in KB:
        KB.add('Hostile(A)')
        print(f"Inferred: Hostile(A)")

    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)")

    if 'Criminal(Robert)' in KB:
        print("Robert is a criminal!")
    else:
        print("No more inferences can be made.")

forward_chaining()
```

Output:

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

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

Algorithm:



Code:

```
# Define the knowledge base (KB)
KB = {
    # Rules and facts
    "philosopher(X)": "human(X)",  # Rule 1: All philosophers are humans
    "human(Socrates)": True,  # Socrates is human (deduced from philosopher)
    "teachesAtUniversity(X)": "philosopher(X) or scientist(X)",  # Rule 2
    "some(philosopher, not scientist)": True,  # Rule 3: Some philosophers are not scientists
    "writesBooks(X)": "teachesAtUniversity(X) and philosopher(X)",  # Rule 4
```

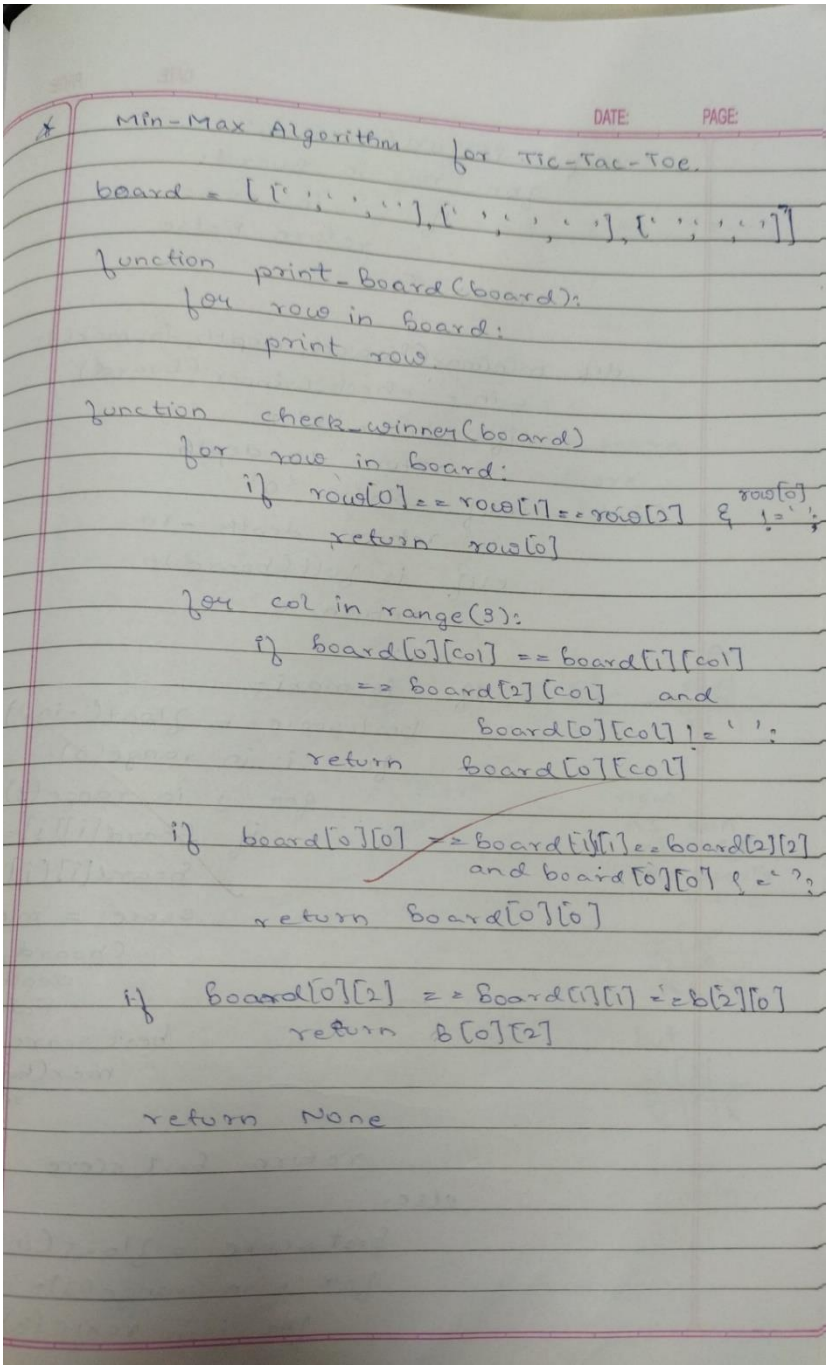
```
    "philosopher(Socrates)": True,  # Fact: Socrates is a philosopher
    "teachesAtUniversity(Socrates)": True,  # Fact: Socrates teaches at university
}

# Function to evaluate a predicate based on the KB
def resolve(predicate):
    # If it's a direct fact in KB
    if predicate in KB and isinstance(KB[predicate], bool):
        return KB[predicate]

    # If it's a derived rule
    if predicate in KB:
        rule = KB[predicate]

        if " and " in rule:  # Handle conjunction
            sub_preds = rule.split(" and ")
            return all(resolve(sub.strip()) for sub in sub_preds)
        elif " or " in rule:  # Handle disjunction
            sub_preds = rule.split(" or ")
            return any(resolve(sub.strip()) for sub in sub_preds)
        elif "not " in rule:  # Handle negation
            sub_pred = rule[4:]  # Remove "not "
            return not resolve(sub_pred.strip())
        else:  # Handle single predicate
            return resolve(rule.strip())

    # If the predicate contains variables
    if "(" in predicate:
        func, args = predicate.split("(")
        args = args.strip(")").split(", ")
        # Handle philosopher and human link
        if func == "philosopher":
            return resolve(f"human({args[0]})")
        # Handle writesBooks rule explicitly
        if func == "writesBooks":
            return resolve(f"teachesAtUniversity({args[0]})") and resolve(f"philosopher({args[0]})")

    # Default to False if no rule or fact applies
    return False

# Query to check if Socrates writes books
query = "writesBooks(Socrates)"
result = resolve(query)

# Print the result
print("Output: 1BM22CS189")
```

print(f"Does Socrates write books? {'Yes' if result else 'No'}")

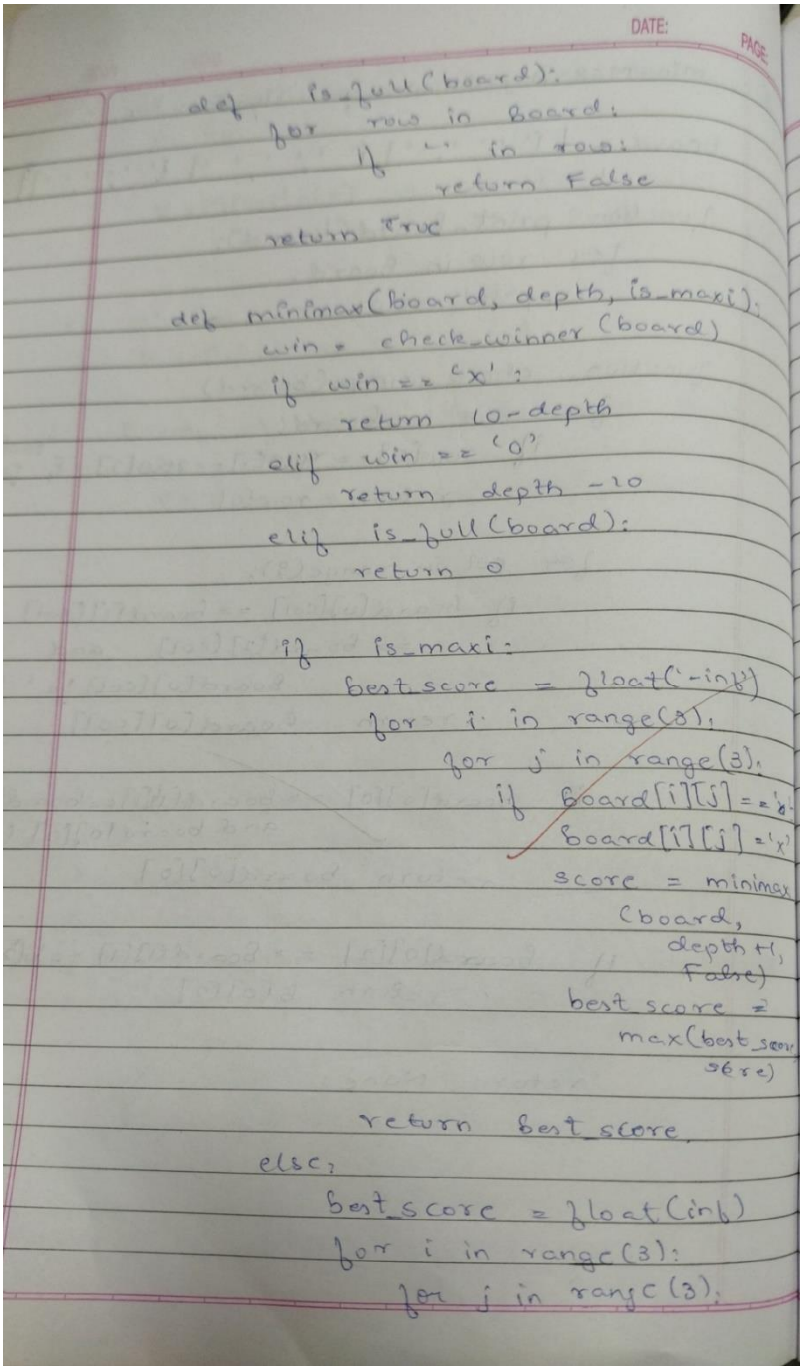
Output:

Implement MinMax Algorithm for TicTacToe.

Algorithm:



```
Min-Max Algorithm for Tic-Tac-Toe.

board = [[' ',' ',' '],[' ',' ',' '],[' ',' ',' ']]

function print_board(board):
    for row in board:
        print row

function check_winner(board)
    for row in board:
        if row[0] == row[1] == row[2] & row[0] != ' ':
            return row[0]

    for col in range(3):
        if board[0][col] == board[1][col]
            == board[2][col] and
                board[0][col] != ' ':
            return board[0][col]

    if board[0][0] == board[1][1] == board[2][2]
                and board[0][0] != ' ':
        return board[0][0]

    if board[0][2] == board[1][1] == b[2][0]
        return b[0][2]

    return None
```



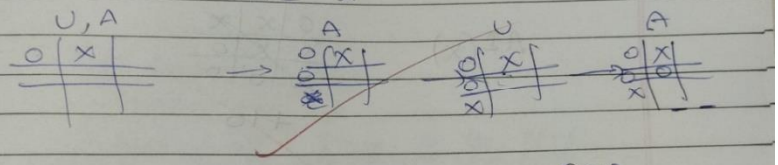
```
def is_full(board):
    for row in board:
        if ' ' in row:
            return False
    return True

def minimax(board, depth, is_maxi):
    win = check_winner(board)
    if win == 'X':
        return 10-depth
    elif win == 'O':
        return depth -10
    elif is_full(board):
        return 0

    if is_maxi:
        best_score = float('-inf')
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = 'X'
                    score = minimax
                        (board,
                        depth+1,
                        False)
                    best_score =
                        max(best_score,
                        score)
        return best_score
    else:
        best_score = float(inf)
        for i in range(3):
            for j in range(3):
```

```
if board[i][j] == '':
    board[i][j] = 'O'
    score = minimax(b, d+1, True)

return best_score
```

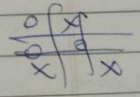
```
find best_move():
    for each empty cell
        if score > best_score
            best_score = score
            best_move = (i, j)
```

Output

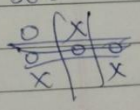
user - O AI - X



now AI can minimize the user's score



but



user win

Code:

```
import math
# Function to print the board
def print_board(board):
    for row in board:
        print("|".join(row))
    print()
# Check for a winner
def check_winner(board):
    # Check rows, columns, and diagonals
    for row in board:
        if row[0] == row[1] == row[2] and row[0] != ' ':
            return row[0]
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] and board[0][col] != ' ':
            return board[0][col]
    if board[0][0] == board[1][1] == board[2][2] and board[0][0] != ' ':
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] and board[0][2] != ' ':
        return board[0][2]
    return None
# Check if the board is full
def is_full(board):
    return all(cell != ' ' for row in board for cell in row)
# Minimax algorithm
def minimax(board, depth, is_maximizing):
    winner = check_winner(board)
```

```
    if winner == 'X':  # Maximizer wins
        return 10 - depth
    elif winner == 'O':  # Minimizer wins
        return depth - 10
    elif is_full(board):  # Tie
        return 0
    if is_maximizing:
        best_score = -math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = 'X'
                    score = minimax(board, depth + 1, False)
                    board[i][j] = ' '
                    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] == ' ':
                    board[i][j] = 'O'
                    score = minimax(board, depth + 1, True)
                    board[i][j] = ' '
                    best_score = min(best_score, score)
        return best_score
# Find the best move
```

```
def find_best_move(board):
    best_score = -math.inf
    move = None
    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':
                board[i][j] = 'X'
                score = minimax(board, 0, False)
                board[i][j] = ' '
                if score > best_score:
                    best_score = score
                    move = (i, j)
    return move
# Main function to play Tic Tac Toe
def main():
    board = [[' ' for _ in range(3)] for _ in range(3)]
    print("Tic Tac Toe: X (AI) vs O (Player)")
    print_board(board)

    while True:
        # Player's move
        row, col = map(int, input("Enter your move (row and column: 0 1 2): ").split())
        if board[row][col] != ' ':
            print("Cell is already taken. Try again.")
            continue
        board[row][col] = 'O'
        # Check for end of game
```

```
            if check_winner(board) or is_full(board):
                break
            # AI's move
            print("AI's turn...")
            move = find_best_move(board)
            if move:
                board[move[0]][move[1]] = 'X'
            # Print the board
            print_board(board)
            # Check for end of game
            if check_winner(board) or is_full(board):
                break
    # Determine the result
    winner = check_winner(board)
    if winner:
        print(f"{winner} wins!")
    else:
        print("It's a tie!")
    print_board(board)
if __name__ == "__main__":
    main()
```

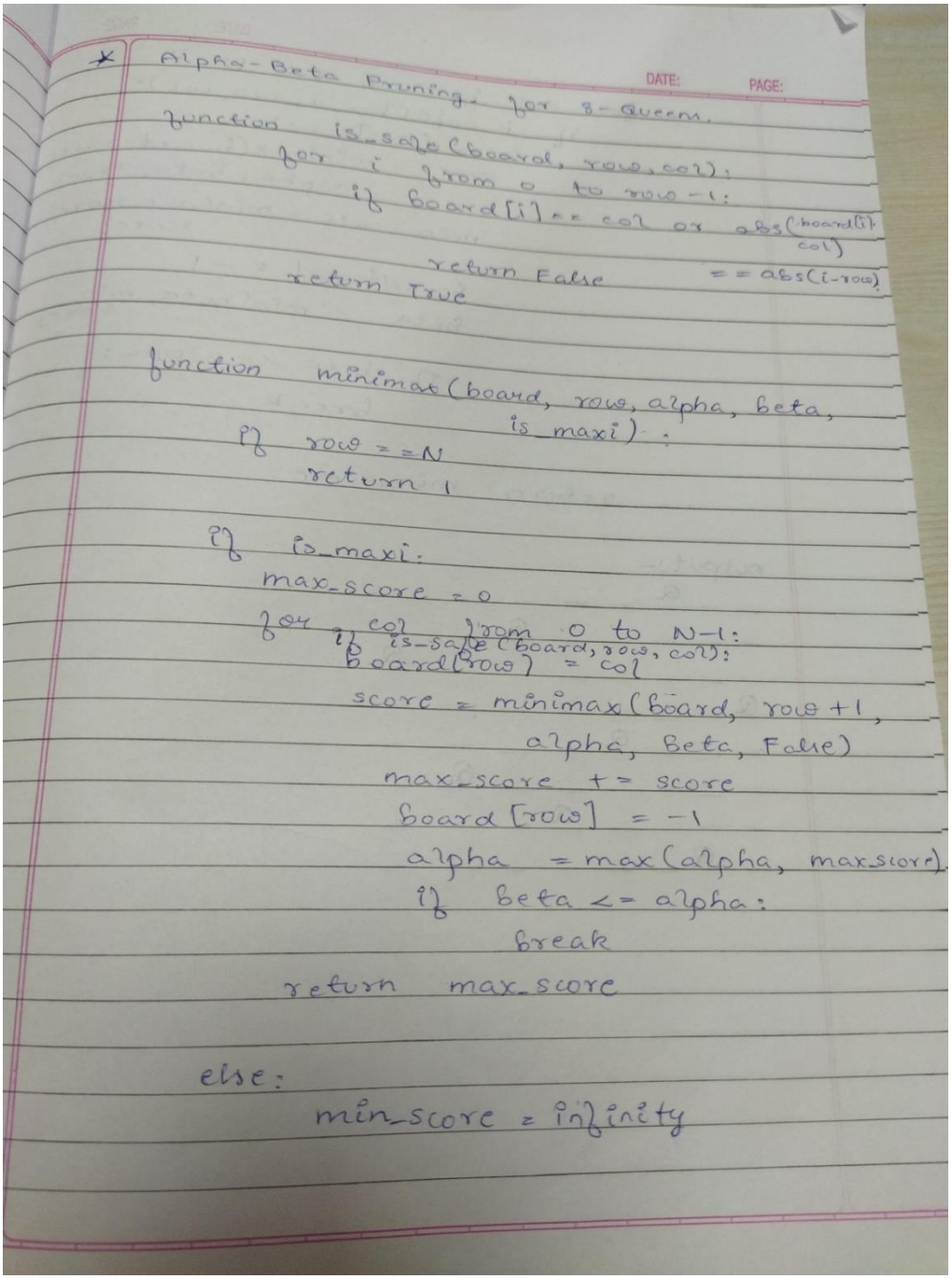
Output:

```
Tic Tac Toe!
You are 'O'. The AI is 'X'.
  |   |
---------
  |   |
---------
  |   |
---------
Enter your move (row and column: 0, 1, or 2): 2 2
Your move:
  |   |
---------
  |   |
---------
  |   | O
---------
AI is making its move...
AI's move:
  |   |
---------
  | X |
---------
  |   | O
---------
Enter your move (row and column: 0, 1, or 2): 0 0
Your move:
O |   |
---------
  | X |
---------
  |   | O
---------
AI is making its move...
AI's move:
O | X |
---------
  | X |
---------
  |   | O
---------
Enter your move (row and column: 0, 1, or 2): 2 1
Your move:
O | X |
---------
  | X |
---------
  | O | O
---------
AI is making its move...
```

```
AI is making its move...
AI's move:
O | X |
---------
  | X |
---------
X | O | O
---------
Enter your move (row and column: 0, 1, or 2): 0 2
Your move:
O | X | O
---------
  | X |
---------
X | O | O
---------
AI is making its move...
AI's move:
O | X | O
---------
  | X | X
---------
X | O | O
---------
Enter your move (row and column: 0, 1, or 2): 1 0
Your move:
O | X | O
---------
O | X | X
---------
X | O | O
---------
It's a draw!
```

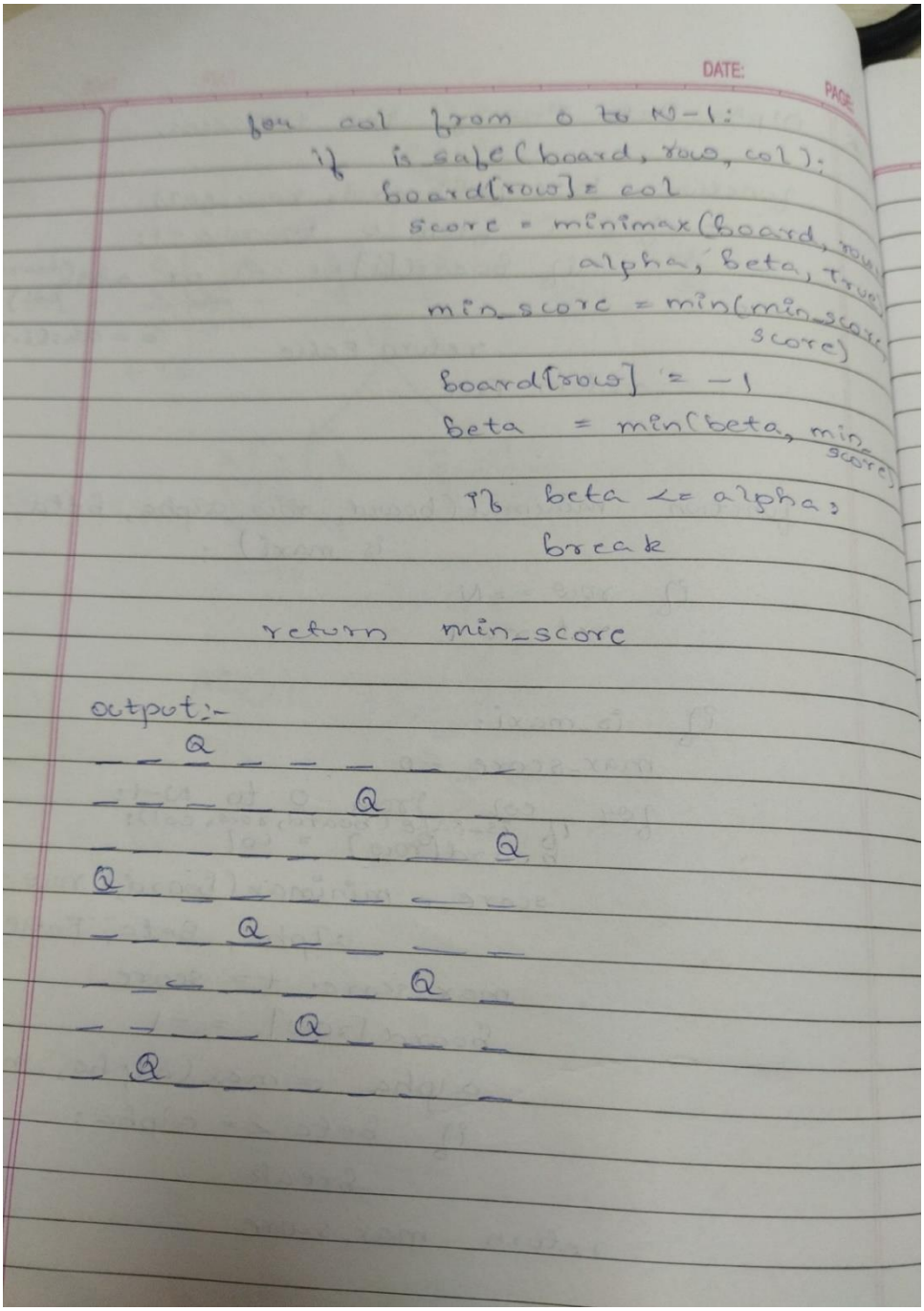
Implement Alpha-Beta Pruning for 8Queens.

Algorithm:



Alpha-Beta Pruning for 8-Queens.

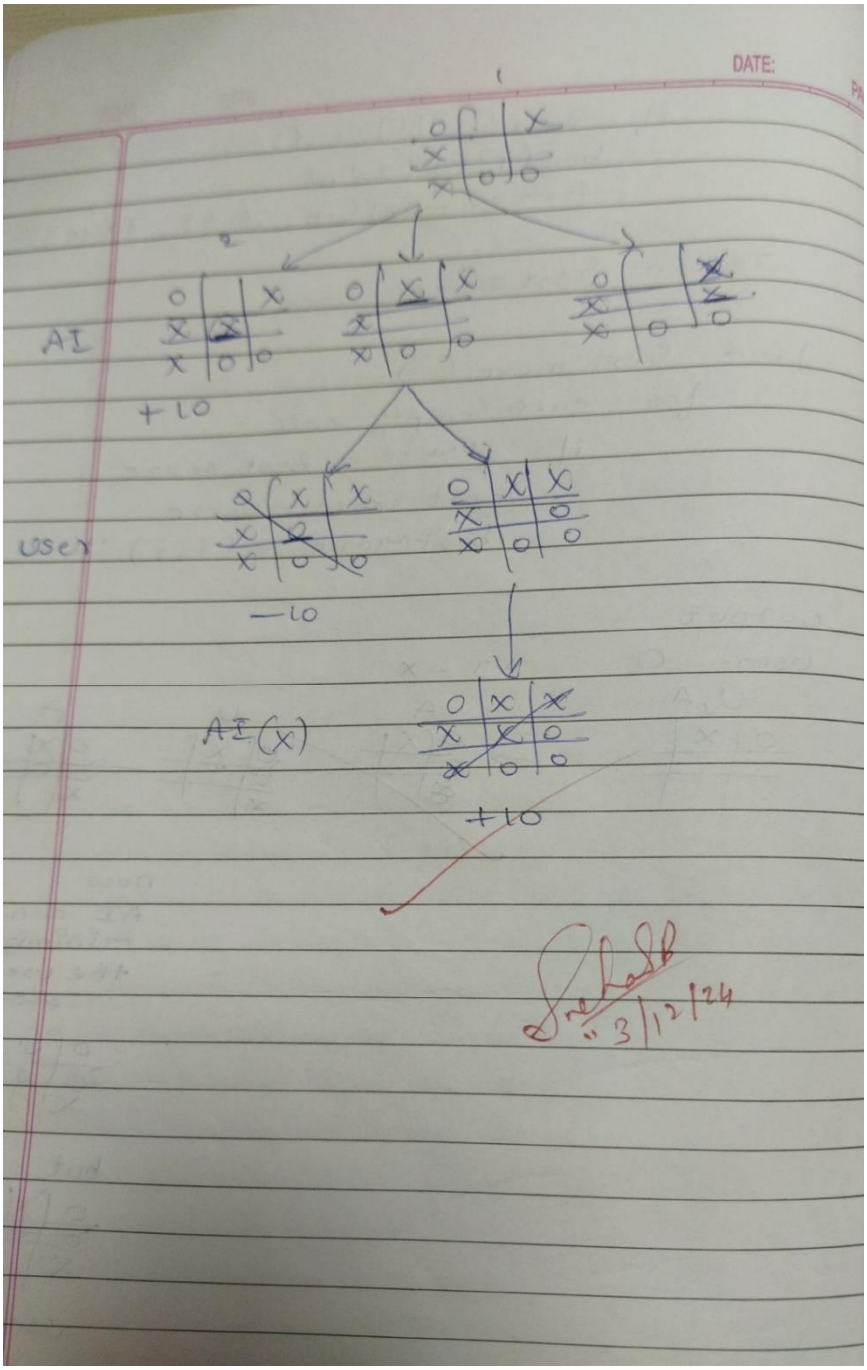
```
function is_safe(board, row, col):
    for i from 0 to row-1:
        if board[i] == col or abs(board[i] - col) == abs(i-row):
            return False
    return True

function minimax(board, row, alpha, beta, is_maxi):
    if row == N
        return 1

    if is_maxi:
        max_score = 0
        for col from 0 to N-1:
            if is_safe(board, row, col):
                board[row] = col
                score = minimax(board, row+1, alpha, beta, False)
                max_score += score
                board[row] = -1
                alpha = max(alpha, max_score)
                if beta <= alpha:
                    break
        return max_score

    else:
        min_score = infinity
```



```
for col from 0 to N-1:
    if is_safe(board, row, col):
        board[row] = col
        score = minimax(board, row+1,
                alpha, beta, True)
        min_score = min(min_score,
                score)
        board[row] = -1
        beta = min(beta, min_score)
        if beta <= alpha:
            break

return min_score
```

output:-

```
  Q
      Q
          Q
Q
    Q
        Q
   Q
 Q
```

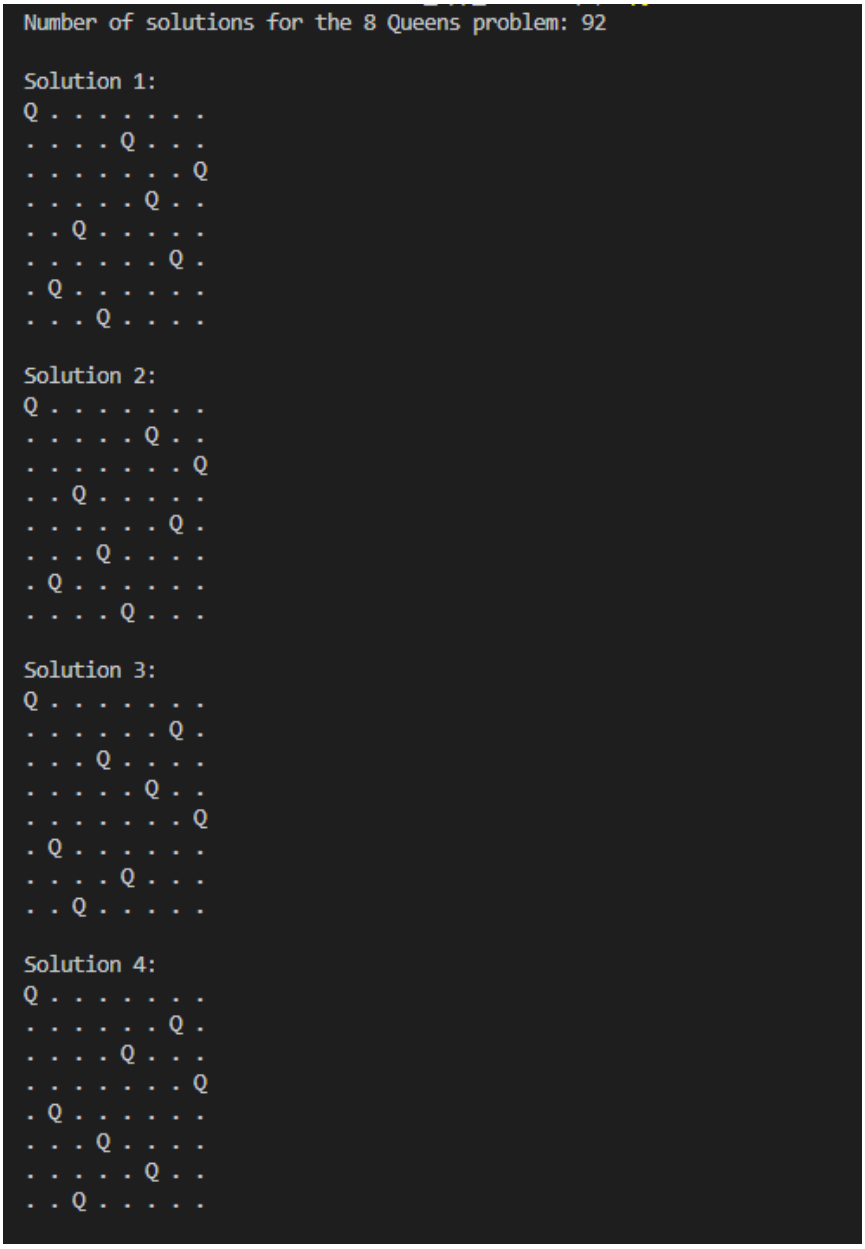

DATE:
AI
+10
User
−10
AI(X)
+10
3/12/24

Code:

```
def is_valid(board, row, col):


    for i in range(row):
        if board[i] == col or \
           abs(board[i] - col) == abs(i - row):
            return False
    return True


def alpha_beta(board, row, alpha, beta, isMaximizing):


    if row == len(board):
        return 1


    if isMaximizing:
        max_score = 0
        for col in range(len(board)):
            if is_valid(board, row, col):
                board[row] = col
                max_score += alpha_beta(board, row + 1, alpha, beta, False)
                board[row] = -1
                alpha = max(alpha, max_score)
                if beta <= alpha:
                    break
        return max_score
    else:
        min_score = float('inf')
        for col in range(len(board)):
            if is_valid(board, row, col):
```

```
            board[row] = col
            min_score = min(min_score, alpha_beta(board, row + 1, alpha, beta, True))
            board[row] = -1
            beta = min(beta, min_score)
            if beta <= alpha:
                break
        return min_score

def solve_8_queens():

    board = [-1] * 8
    alpha = -float('inf')
    beta = float('inf')
    return alpha_beta(board, 0, alpha, beta, True)

solutions = solve_8_queens()
print(f"Number of solutions for the 8 Queens problem: {solutions}")
```

Output:


Number of solutions for the 8 Queens problem: 92
Solution 1:
Q . . . . . .
. . . . Q . . .
. . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
Solution 2:
Q . . . . . .
. . . . . Q . .
. . . . Q
. . Q . . . . .
. . . . . . Q .
. . . Q . . . .
. Q . . . . . .
. . . . Q . . .
Solution 3:
Q . . . . . .
. . . . . . Q .
. . . Q . . . .
. . . . . Q . .
. . . . Q
. Q . . . . . .
. . . . Q . . .
. . Q . . . . .
Solution 4:
Q . . . . . .
. . . . . . Q .
. . . . Q . . .
. . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . Q . . . . .