VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Raghavendra N (1BM22CS213)

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 outby **Raghavendra N (1BM22CS213),** 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.

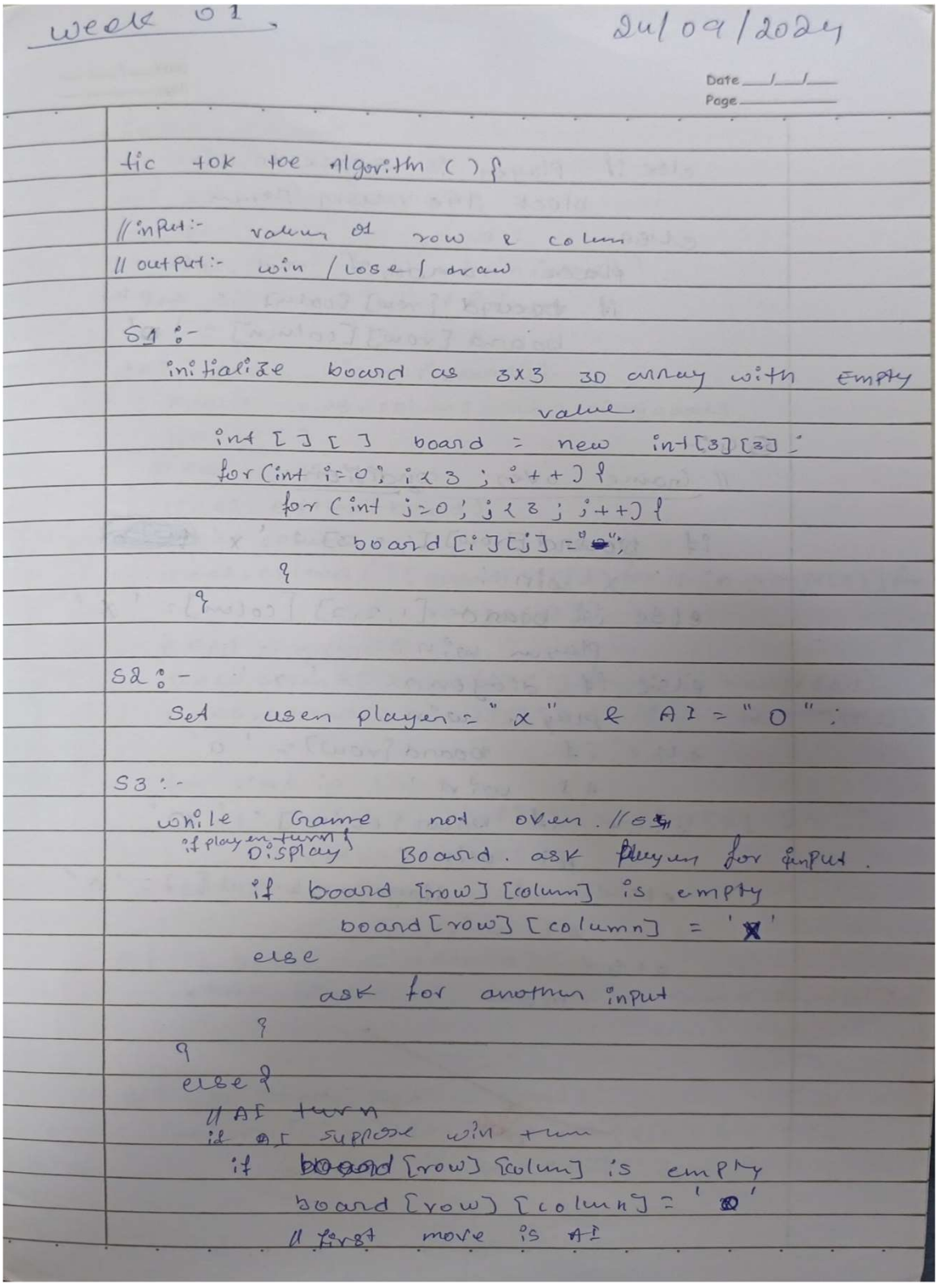
Prof. Radhika A D	Dr. Kavitha Sooda
Assistant Professor	Professor & HOD
Department of CSE, BMSCE	Department of CSE, BMSCE

Index

Github Link:
https://github.com/RaghavendraN-cs213/AI-LAb

Implement Tic-Tac-Toe Game.

Algorithm:



week 01 — 24/09/2024

Date __/__/__
Page _____

tic tok toe algorithm ( ) {

// input :- value of row & column
// output :- win / lose / draw

S1 :-
initialize board as 3x3 2D array with Empty value.

```
int [ ] [ ] board = new int[3][3];
for(int i=0; i<3 ; i++) {
    for (int j=0; j<3 ; j++) {
        board [i][j] = " ";
    }
}
```

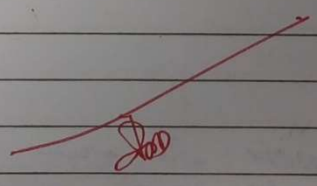
S2 :-
Set user player = "X" & AI = "O";

S3 :-
while Game not over. // if player turn { Display Board. ask player for input.
    if board [row] [column] is empty
        board[row] [column] = 'X'
    else
        ask for another input
}
else {
// AI turn
// I suppose win turn
    if board [row] [column] is empty
        board [row] [column] = 'O'
// first move is AI

Date __/__/__
Page _____

else if Player is suppose to win
block the user/Player.
else
place ramdom of row, colm
if board [row] [colm] is empty
board [row] [column] = 'O'

Ans:-
// Game over condition
if board [row] [1,2,3] = 'X'
X win;
else if board [1,2,3] [colm] = 'X'
Player win;
else if diagonal = 'X'
Player win;
else if board [row] = 'O'
AI win;
else if board [colm] = 'O'
AI win;
else if diagonal. board [ ] = 'O'
AI win.
else
draw.



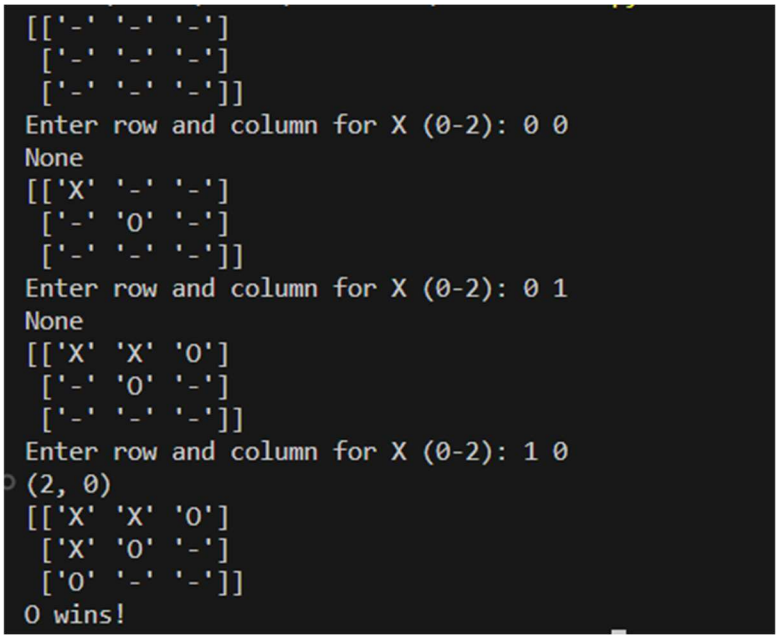
Code:

```
import random
import numpy as np

board = [["-"] * 3 for _ in range(3)]

def check_win():
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != "-":
            return True
        if board[0][i] == board[1][i] == board[2][i] != "-":
            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 full():
    return all(cell != "-" for row in board for cell in row)

def can_win(m):
    for i in range(3):
        row = board[i]
        if row.count(m) == 2 and row.count("-") == 1:
```

```
            return (i, row.index("-"))

    for i in range(3):
        col = [board[j][i] for j in range(3)]
        if col.count(m) == 2 and col.count("-") == 1:
            return (col.index("-"), i)

    diag1 = [board[i][i] for i in range(3)]
    if diag1.count(m) == 2 and diag1.count("-") == 1:
        return (diag1.index("-"), diag1.index("-"))

    diag2 = [board[i][2 - i] for i in range(3)]
    if diag2.count(m) == 2 and diag2.count("-") == 1:
        return (diag2.index("-"), 2 - diag2.index("-"))

    return None

def display():
    print(np.array(board))

while True:
    display()
    u = tuple(map(int, input("Enter row and column for X (0-2): ").strip().split()))
    if board[u[0]][u[1]] != "-":
        print("Invalid move, try again.")
        continue
```

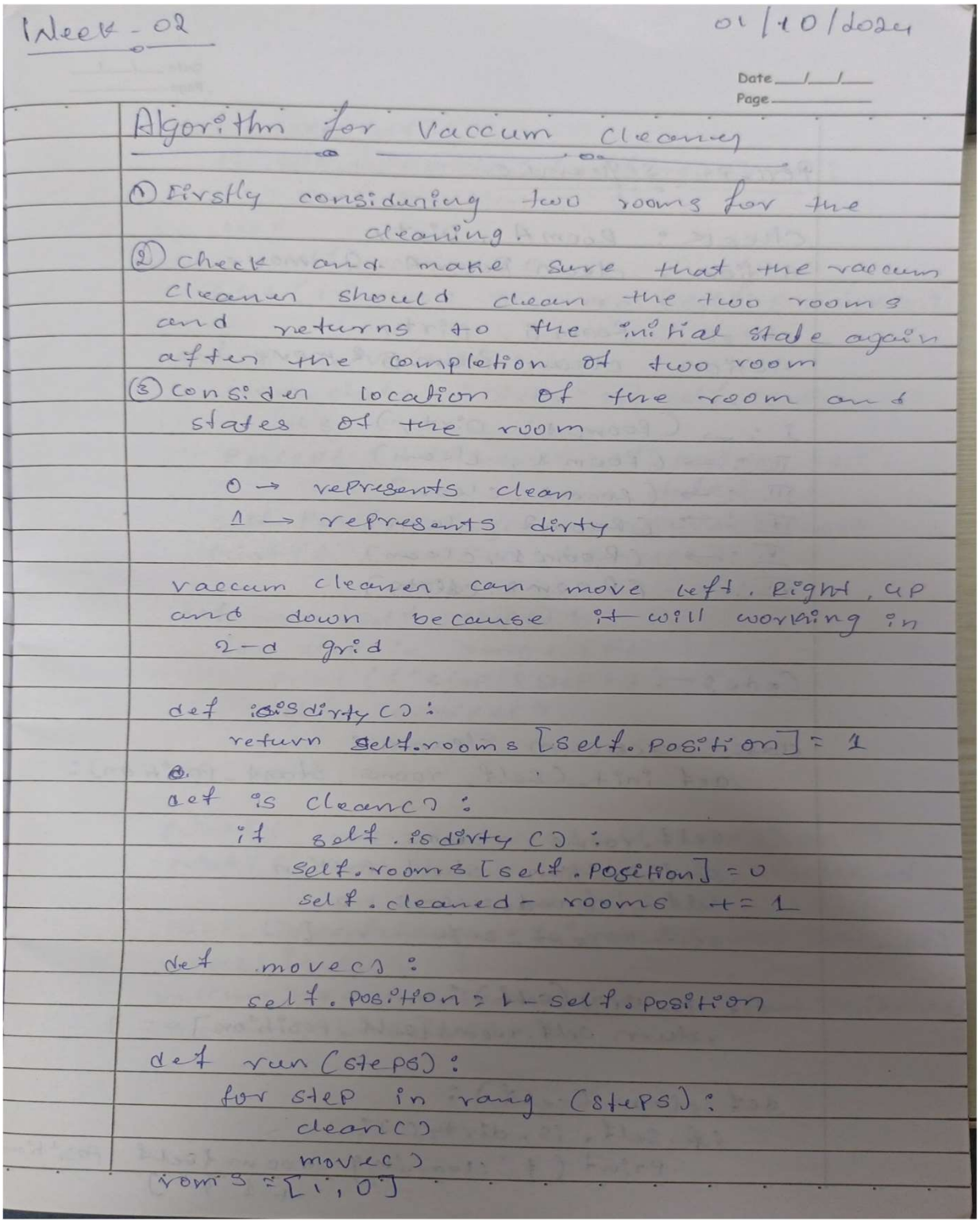
```
board[u[0]][u[1]] = "X"

if check_win():
    display()
    print("X wins!")
    break

if full():
    display()
    print("It's a tie!")
    break

move = can_win("O")
print(move)
if move is None:
    move = can_win("X")
    if move is None:
        empty = [(i, j) for i in range(3) for j in range(3) if board[i][j] == "-"]
        move = random.choice(empty)
if board[1][1]=="-":
    move=(1 ,1)
board[move[0]][move[1]] = "O"

if check_win():
    display()
    print("O wins!")
    break
```

Output:


[['-' '-' '-']
['-' '-' '-']
['-' '-' '-']]
Enter row and column for X (0-2): 0 0
None
[['X' '-' '-']
['-' 'O' '-']
['-' '-' '-']]
Enter row and column for X (0-2): 0 1
None
[['X' 'X' 'O']
['-' 'O' '-']
['-' '-' '-']]
Enter row and column for X (0-2): 1 0
(2, 0)
[['X' 'X' 'O']
['X' 'O' '-']
['O' '-' '-']]
O wins!

Implement Vacuum Cleaner Agent.

Algorithm:



Week - 02 01/10/2024

Date / /
Page

Algorithm for vaccum cleaner

① Firstly considering two rooms for the cleaning.
② check and make sure that the vaccum cleaner should clean the two rooms and returns to the initial state again after the completion of two room
③ consider location of the room and states of the room

0 → represents clean
1 → represents dirty

vaccum cleaner can move left, Right, up and down because it will working in 2-d grid

```
def isdirty():
    return self.rooms[self.position] == 1
def is clean():
    if self.isdirty():
        self.rooms[self.position] = 0
        self.cleaned += 1
def move():
    self.position = 1 - self.position
def run(steps):
    for step in rang(steps):
        clean()
        move()
rooms = [1, 0]
```

Date __/__/__
Page

Percept sequence

Check : RoomA , Dirty
Action : clean RoomA & move

check : RoomB , Dirty
Action : clean RoomB & move

I :→ (Room 1, Dirty)
II :→ (Room 2, clean)
III :→ (Room 1, clean)
IV :→ (Room 2, clean)
V :→ (Room 1, clean)
VI :→ (Room 2, clean)

Code :-

```
class vaccum cleaner :
    def __init__(self, rooms, start_position):

        self.rooms = rooms
        self.position = start_position
        self.cleaned_rooms = 0
        self.Percept_sequence = [ ]

    def is_dirty(self):
        return self.rooms[self.position] == 1

    def clean(self):
        if self.is_dirty():
            print(f"cleaning room {self.position
                                    + 1}")
```

```
def move(self):
    if self.position clean(self.room)- 1 :
        self. position += 1
    else
        self.position = 0
    Print(f"moved to room {self position+1}")

def perceive(self):
    room_state = "Dirty" :if self. is dirty()
        else "clean"
    Percept = (f"Room {self. position + 1}",
                    room_state)
    self. Percept_Sequence. appends(Percept)
    Print(f"perception: {percept}")

def run(self, steps):
    for step in range(steps):
        print(f"step {step + 1}:")
        self. Perceive()
        self. clean()
        self. move()

print(f"Rooms states: {self. rooms}\n")
print(f"Total cleaned rooms: {self.cleaned-
                    room}")
Print("Percept sequence:", self. Percept_Sequence)
rooms = [1, 0, 1, 1]
vaccum = vaccumcleaner(rooms, start_position=0)
vaccum, run( steps = 8)
```

Date __/__/__
Page _____

Sample Output :-

Step 1 :
Perception : ('Room 1', 'Dirty')
Cleaning Room 1
Move to Room 2
Rooms states = [0, 0, ', ']
Step 2 :
Perception : ('Room 2', 'clean')
moved to Room 3
Rooms states = [0, 0, ', ']

Step 3 :
Perception : ('Room 3', 'Dirty')
Cleaning Room 3
moved to room 4
Rooms states = [0, 0, 0, ']

Step 4 :
Perception : ('Room 4', 'Dirty')
Cleaning room 4
move 4 to room 1
Room states = [0, 0, 0, 0]

Step 5 :
Perception : ('Room 1', 'clean')
moved to room 2
Rooms states : [0, 0, 0, 0]

Step 6 :-
Perception : ('Room 2', 'clean')
moved to room 3
Rooms states : [0, 0, 0, 0]

Date __/__/__
Page _____

step 7 :
Perception : ('Room 3', 'clean')
moved to room 4
Room status : [0, 0, 0, 0]

step 8 :
Perception : ('Room 4', 'clean')
moved to room 1
Rooms status = [0, 0, 0, 0]

Total cleaned rooms : 3
Percept sequence :
[(Room 1, Dirty)
(Room 2, clean)
(Room 3, Dirty)
(Room 4, Dirty)
(Room 1, clean)
(Room 2, clean)
(Room 3, clean)
(Room 4, clean)]

Code:

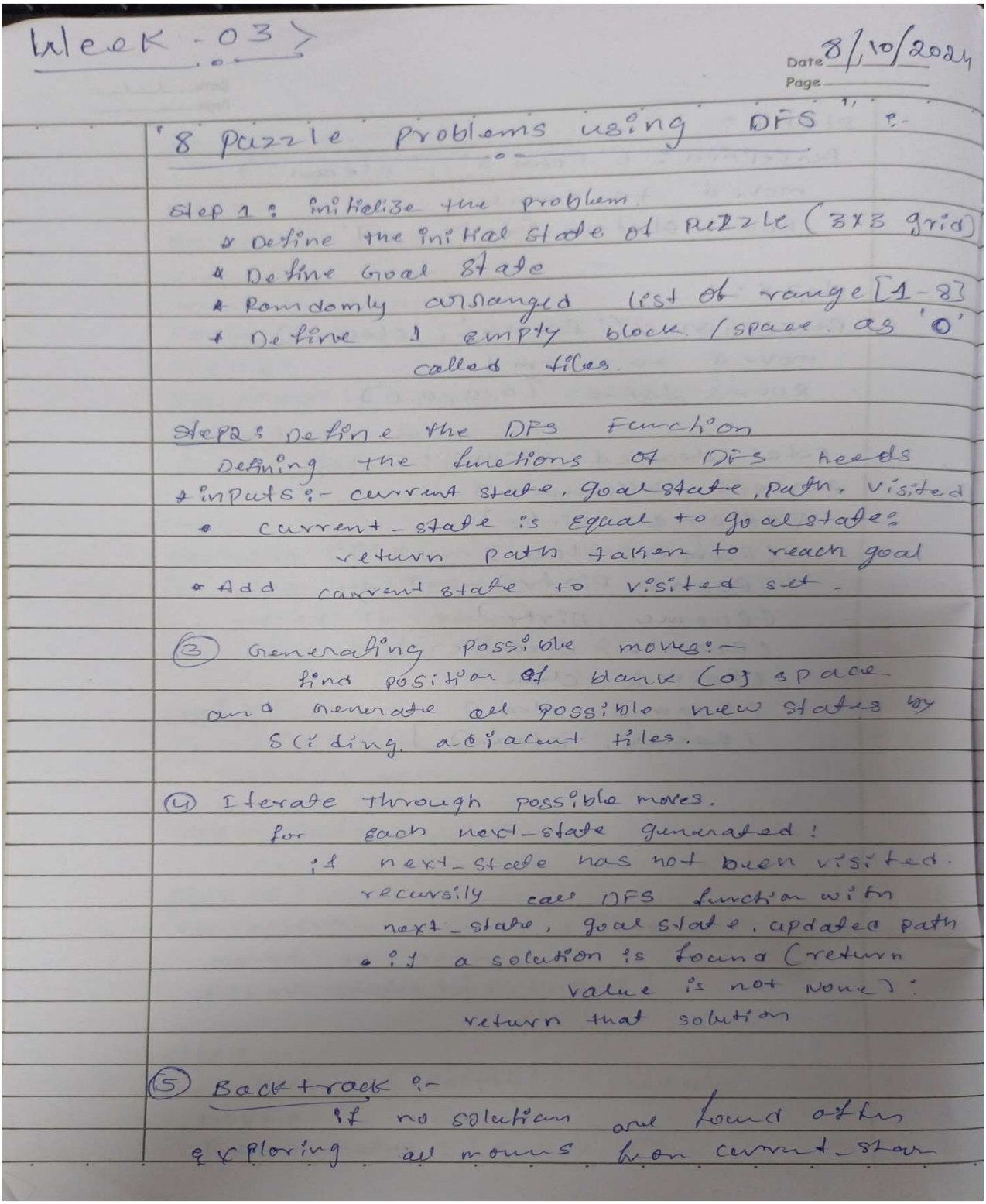
```
import random
l=[random.choice([0,1]),random.choice([0,1])]
def check(i):
    if l[i]==0:
        l[i]=1
        print(f"Cleaned Room {i}")
    print(f"Moved to Room {(i+1)%2}")
    return (i+1)%2
i=random.choice([0,1])

print(f"{i} is the start index")
print("0 is dirty and 1 is clean")
print(f"{l} is the initial state of room")

while sum(l)!=2:
    i=check(i)
    if l[(i+1)%2]==1:
        l[(i+1)%2]=random.choice([0,1])
        if l[(i+1)%2]==0:
            print(f"Room {(i+1)%2} got dirty")
    print(f"{l} is current state of rooms")
print("Rooms are clean")
```

Output:

```
1 is the start index
0 is dirty and 1 is clean
[0, 0] is the initial state of room
Cleaned Room 1
Moved to Room 0
Room 1 got dirty
[0, 0] is current state of rooms
Cleaned Room 0
Moved to Room 1
Room 0 got dirty
[0, 0] is current state of rooms
Cleaned Room 1
Moved to Room 0
Room 1 got dirty
[0, 0] is current state of rooms
Cleaned Room 0
Moved to Room 1
Room 0 got dirty
[0, 0] is current state of rooms
Cleaned Room 1
Moved to Room 0
[0, 1] is current state of rooms
Cleaned Room 0
Moved to Room 1
Room 0 got dirty
[0, 1] is current state of rooms
Moved to Room 0
Room 1 got dirty
[0, 0] is current state of rooms
Cleaned Room 0
Moved to Room 1
[1, 0] is current state of rooms
Cleaned Room 1
Moved to Room 0
[1, 1] is current state of rooms
Rooms are clean
```

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

Algorithm:



Week - 03

Date 8/10/2024
Page

'8 Puzzle problems using DFS :-

Step 1 : initialize the problem.
* Define the initial state of Puzzle (3x3 grid)
* Define Goal state
* Randomly arranged list of range [1-8]
* Define 1 empty block / space as '0' called tiles.

Step 2 : Define the DFS Function
Defining the functions of DFS needs
* inputs :- current state, goal state, path, visited
* current_state is Equal to goal state?
return path taken to reach goal
* Add current state to visited set.

(3) Generating possible moves :-
find position of blank (0) space
and generate all possible new states by
sliding adjacent tiles.

(4) Iterate through possible moves.
for each next_state generated :
if next_state has not been visited.
recursily call DFS function with
next_state, goal state, updated path
* if a solution is found (return
value is not None) :
return that solution

(5) Backtrack :-
if no solution are found after
exploring all moves from current_state

Date __/__/__
Page _____

remove current state from visited set
and backtrack
indicate no solution in this path

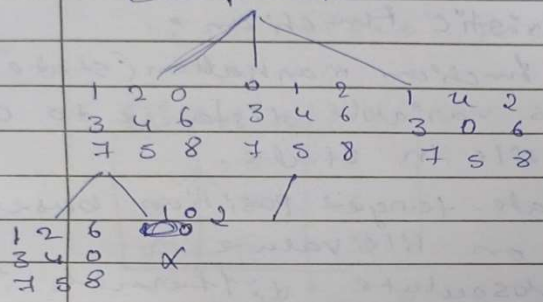
(8) ~~Start search~~

initial state

1	0	2
3	4	6
7	5	8

Our Goal

1	2	3
4	5	6
7	8	0



Date / /
Page

8-Puzzle Solving using Manhattan Distance

(1) initialize the problem

* Define the initial state of Puzzle (3x3 Grid)
* Define goal state
* Randomly arranged list of range (0-8)
* Define 1 Empty block as '0' called tile

(2) Defining Heuristic Function :-

Create a function manhattan(state) :
initializes variable distance to 0
for each tile in state.
calculate target position based on tile value.
Add absoulute differences in row and colum indices to distance and return distance.

* input :- current state, goal state, g_cost, visited, path
if current_state is equal to goal_state.
Return Path.

(3) Enumerating Possible moves

* calculate g_cost as g_cost + 1
calculate the heuristic h_cost using manhattan_distance (next_state).
calculate total cost f_cost = g_cost + h_cost
if next state not visited.
Add (f_cost, next_state, updated_path)
to Priority Queue

Date __/__/__
Page _____

③ Process of priority queue.

while priority queue not empty
POP state lowest total cost (f(n))
mark it as visited
call the A* function recursively with
state & updated costs

④ Backtrack.

if no solution found Return to
previous position.

Initial state

1	0	2
3	4	6
7	5	8

our Goal

1	2	3
4	5	6
7	8	0

	V	H
1	0	0
2	0	1
3	1	2
4	0	1
5	1	1
6	0	0
7	0	0
8	0	1

Priority queue. → POP state with lowest cost. return Path make as visited.

Proceed

Code:

```
import heapq
import numpy as np

goal = [[0,1,2], [3,4,5], [6,7,8]]
vis = set()
q = []
parent_map = {}
move_map = {}

def manhattan(curr):
    ans = 0
    pos = {goal[i][j]: (i, j) for i in range(3) for j in range(3)}
    for i in range(3):
        for j in range(3):
            x, y = pos[curr[i][j]]
            ans += abs(i - x) + abs(j - y)
    return ans

def moves(curr):
    x, y = [(i, j) for i in range(3) for j in range(3) if curr[i][j] == 0][0]
    poss = [[0, -1, 'left'], [-1, 0, 'up'], [1, 0, 'down'], [0, 1, 'right']]
    for dx, dy, direction in poss:
```

```
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            curr1 = [row[:] for row in curr]
            curr1[x][y], curr1[nx][ny] = curr1[nx][ny], curr1[x][y]
            tuple_curr1 = tuple(map(tuple, curr1))
            if tuple_curr1 not in vis:
                heapq.heappush(q, (manhattan(curr1), curr1))
                vis.add(tuple_curr1)
                parent_map[tuple(map(tuple, curr1))] = curr
                move_map[tuple(map(tuple, curr1))] = direction

def dfs(curr):
    vis.add(tuple(map(tuple, curr)))
    if curr == goal:
        return True
    moves(curr)
    if q:
        curr = heapq.heappop(q)[1]
        if dfs(curr):
            return True
    return False

def display_board(board):
    print("+---+---+---+")
    for row in board:
        print("| " + " | ".join(str(x) if x != 0 else ' ' for x in row) + " |")
        print("+---+---+---+")
```

```
c =[[] for i in range(3)]
for i in range(3):
  print(f"Enter elements of row {i+1}")
  c[i]=list(map(int,input().split()))
dfs(c)


result_path = []
directions = []
state = goal
while state:
    result_path.append(state)
    directions.append(move_map.get(tuple(map(tuple, state)), None))
    state = parent_map.get(tuple(map(tuple, state)))

for ind, (state, direction) in enumerate(reversed(list(zip(result_path, directions)))):
    print(f"Step {ind}:")
    display_board(state)
    if ind==0:
      print("Initial state")
    if direction:
        print(f" Move empty space {direction}")
    print()

print(f"Steps taken: {len(result_path) - 1}")
```

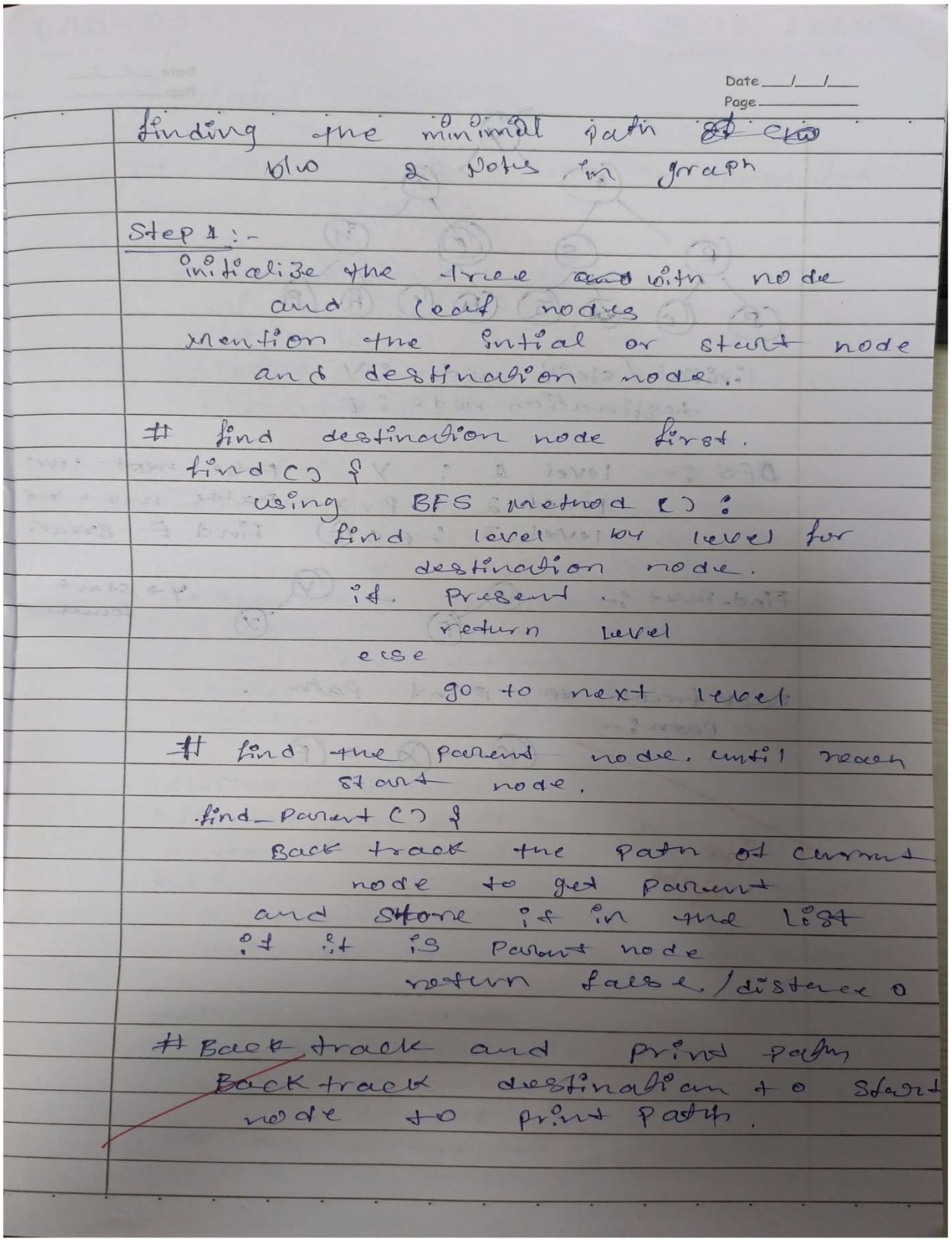
Output:

```
Enter elements of row 1
3 7 6
Enter elements of row 2
4 5 8
Enter elements of row 3
2 0 1
Step 0:
+---+---+---+
| 3 | 7 | 6 |
+---+---+---+
| 4 | 5 | 8 |
+---+---+---+
| 2 |   | 1 |
+---+---+---+
Initial state

Step 1:
+---+---+---+
| 3 | 7 | 6 |
+---+---+---+
| 4 |   | 8 |
+---+---+---+
| 2 | 5 | 1 |
+---+---+---+
 Move empty space up
```

```
Step 57:
+---+---+---+
| 3 | 1 | 2 |
+---+---+---+
| 4 |   | 5 |
+---+---+---+
| 6 | 7 | 8 |
+---+---+---+
 Move empty space down

Step 58:
+---+---+---+
| 3 | 1 | 2 |
+---+---+---+
|   | 4 | 5 |
+---+---+---+
| 6 | 7 | 8 |
+---+---+---+
 Move empty space left

Step 59:
+---+---+---+
|   | 1 | 2 |
+---+---+---+
| 3 | 4 | 5 |
+---+---+---+
| 6 | 7 | 8 |
+---+---+---+
 Move empty space up

Steps taken: 59
```

Implement Iterative deepening search algorithm.

Algorithm:



finding the minimal path ~~between~~ btw 2 nodes in graph

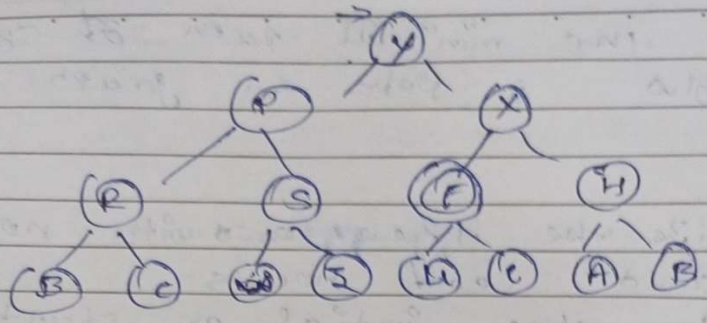
Step 1 :-
initialize the tree with node and leaf nodes
mention the intial or start node and destination node.

# find destination node first.
find() {
using BFS method() {
find level by level for destination node.
if present.
return level
else
go to next level

# find the parent node. until reach start node.
find_Parent() {
Back track the path of current node to get parent and store it in the list if it is Parent node
return false / distance 0

Back track and print path
Back track destination to start node to print path.

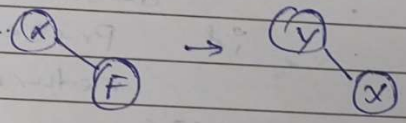
Date __/__/__
Page



initial / start node : Y
destination node : F

BFS :- level 1 : Y False next level
level 2 : P, X False next level
level 3 : R S (F) Find F Break.

Find Parent :-



Y → start found.

Back track to print path.

Path :-

(Y) → (X) → (F)

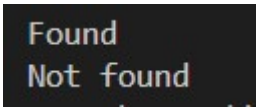
Done
16/10/24

Code:

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []  # List to hold children nodes

    def add_child(self, child_node):
        self.children.append(child_node)

def iddfs(root, goal):
    for i in range(0,100000):
        res=dls(root,goal,i)
        if res:
            print("Found")
            return
    print("Not found")

def dls(root,goal,depth):
    if depth==0:
        if root.value==goal:
            return True
        return False
    for child in root.children:
        if dls(child,goal,depth-1):
            return True
    return False

root=TreeNode("Y")
node1=TreeNode("P")
node2=TreeNode("X")
node3=TreeNode("R")
node4=TreeNode("S")
node5=TreeNode("F")
node6=TreeNode("H")
node7=TreeNode("B")
```

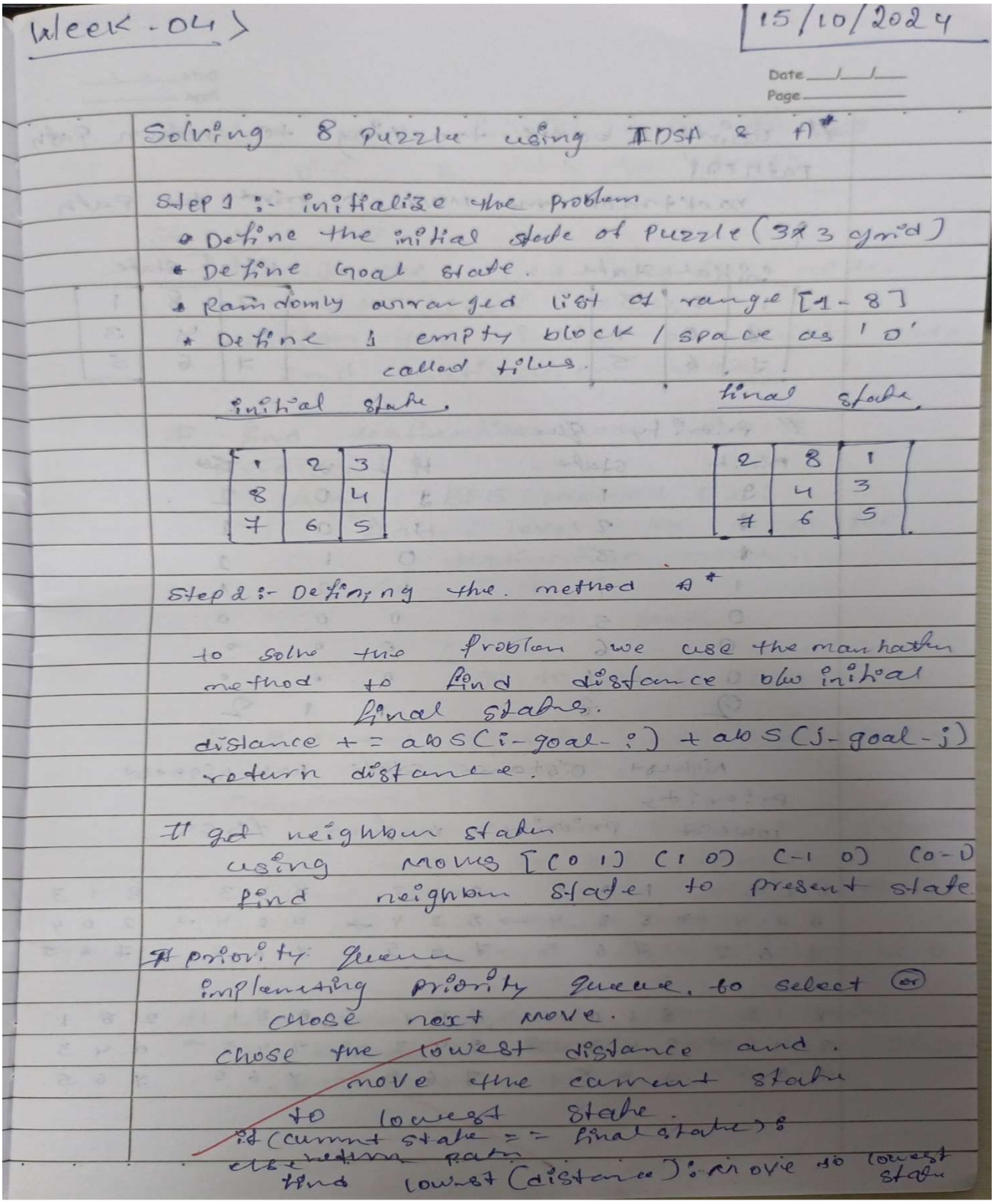
```
node8=TreeNode("C")
node9=TreeNode("S")

root.add_child(node1)
root.add_child(node2)

node1.add_child(node3)
node1.add_child(node4)

node2.add_child(node5)
node2.add_child(node6)

node3.add_child(node7)
node3.add_child(node8)

node4.add_child(node9)

iddfs(root, "F")
iddfs(root, "A")
```

Output:



Implement A* search algorithm.

Algorithm:



Week - 04 >

15/10/2024

Date __/__/__
Page _____

Solving 8 puzzle using IDSA & A*

Step 1 :- initialize the problem
* Define the initial state of puzzle (3 x 3 grid)
* Define Goal state.
* Randomly arranged list of range [1 - 8]
* Define 1 empty block / space as '0' called tiles.

initial state.

1	2	3
8		4
7	6	5

final state.

2	8	1
	4	3
7	6	5

Step 2 :- Defining the method A*

to solve the problem we use the manhattan method to find distance btw initial final states.

distance += abs(i - goal_i) + abs(j - goal_j)
return distance.

If get neighbour state
using moves [(0 1) (1 0) (-1 0) (0 -1)
find neighbour state to present state

priority queue
implementing priority queue, to select (or) chose next move.
chose the lowest distance and move the current state to lowest state
if (curnt state == final state):
find lowest (distance): move to lowest state

caps lock

Date __/__/__
Page

// using back tracking to return path

path of

backtrack the move to print the path

Q

Initial State

1	2	3
8	0	4
7	6	5

Final State

2	8	1
	4	3
7	6	5

// Priority Queue

Priority	State	H	V	Dist
2	1	2	0	2
1	2	1	0	1
1	3	0	1	1
1	4	1	0	1
0	5	0	0	0
0	6	0	0	0
0	7	0	0	0
02	8	1	1	2

highest Distance state has highest priority.

lowest priority perform first

1 2 3 → 1 0 3 → 0 1 3 → 8 1 3 → 8 1 3
8 0 4 → 8 2 4 → 8 2 4 → 0 2 4 → 2 0 4
7 6 5 → 7 6 5 → 7 6 5 → 7 6 5 → 7 6 5

8 1 3 → 8 1 0 → 8 0 1 → 0 8 1 → 2 8 1
2 4 0 → 2 4 3 → 2 4 3 → 2 4 3 → 0 4 3
7 6 5 → 7 6 5 → 7 6 5 → 7 6 5 → 7 6 5

Code:

```
import heapq
import numpy as np

goal = [[2,8,1], [0,4,3], [7,6,5]]
vis = set()
q = []
parent_map = {}
move_map = {}

def manhattan(curr):
    ans = 0
    pos = {goal[i][j]: (i, j) for i in range(3) for j in range(3)}
    for i in range(3):
        for j in range(3):
            x, y = pos[curr[i][j]]
            ans += abs(i - x) + abs(j - y)
    return ans

def moves(curr,g):
    x, y = [(i, j) for i in range(3) for j in range(3) if curr[i][j] == 0][0]
    poss = [[0, -1, 'left'], [-1, 0, 'up'], [1, 0, 'down'], [0, 1, 'right']]
    for dx, dy, direction in poss:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            curr1 = [row[:] for row in curr]
            curr1[x][y], curr1[nx][ny] = curr1[nx][ny], curr1[x][y]
            tuple_curr1 = tuple(map(tuple, curr1))
            if tuple_curr1 not in vis:
                f=g+1+manhattan(curr1)
                heapq.heappush(q, (f, curr1,g+1))
                vis.add(tuple_curr1)
                parent_map[tuple(map(tuple, curr1))] = curr
                move_map[tuple(map(tuple, curr1))] = direction

def a_star(curr,g):
    vis.add(tuple(map(tuple, curr)))
    if curr == goal:
        return True
    moves(curr,g)
    if q:
        curr = heapq.heappop(q)
        if a_star(curr[1],curr[2]):
            return True
    return False
```

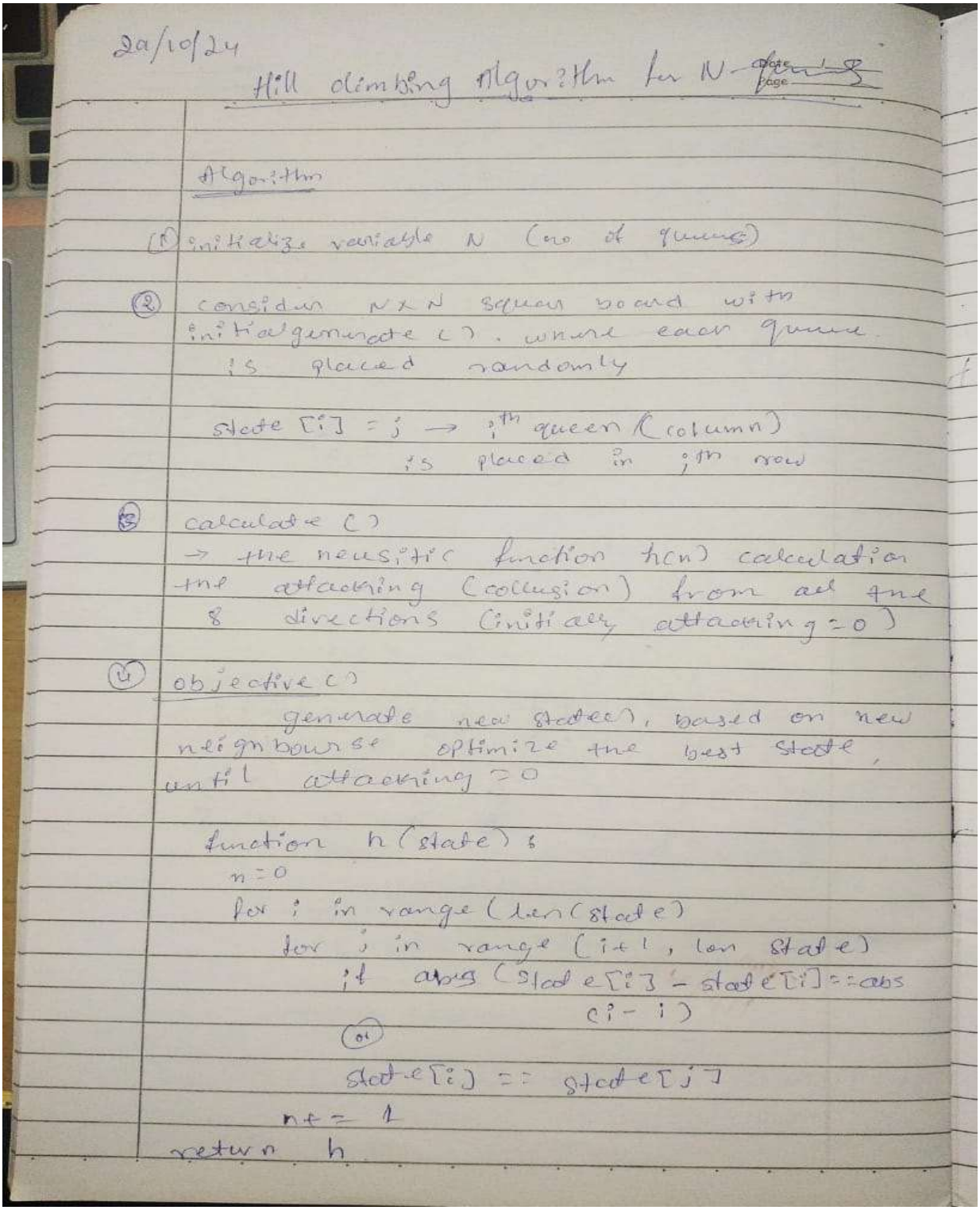
```
def display_board(board):
    print("+---+---+---+")
    for row in board:
        print("| " + " | ".join(str(x) if x != 0 else ' ' for x in row) + " |")
        print("+---+---+---+")

c =[[] for i in range(3)]
for i in range(3):
  print(f"Enter elements of row {i+1}")
  c[i]=list(map(int,input().split()))
a_star(c,0)

result_path = []
directions = []
state = goal
while state:
    result_path.append(state)
    directions.append(move_map.get(tuple(map(tuple, state)), None))
    state = parent_map.get(tuple(map(tuple, state)))

for ind, (state, direction) in enumerate(reversed(list(zip(result_path, directions)))):
    print(f"Step {ind}:")
    display_board(state)
    if ind==0:
      print("Initial state")
    if direction:
        print(f" Move empty space {direction}")
    print()

print(f"Steps taken: {len(result_path) - 1}")
```

Output:

```
Enter elements of row 1
1 2 3
Enter elements of row 2
8 0 4
Enter elements of row 3
7 6 5
Step 0:
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 8 |   | 4 |
+---+---+---+
| 7 | 6 | 5 |
+---+---+---+
Initial state

Step 1:
+---+---+---+
| 1 |   | 3 |
+---+---+---+
| 8 | 2 | 4 |
+---+---+---+
| 7 | 6 | 5 |
+---+---+---+
 Move empty space up
```

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

Algorithm:



29/10/24

Hill climbing algorithm for N-

Algorithm

(1) initialize variable N (no of queens)

(2) consider N x N square board with initial generate ( ). where each queen is placed randomly

state [i] = j → i^{th} queen (column) is placed in j^{th} row

(3) calculate ( )
→ the heuristic function h(n) calculation the attacking (collusion) from all the 8 directions (initially attacking = 0)

(4) objective ( )
generate new state(s), based on new neighbourse optimize the best state, until attacking = 0

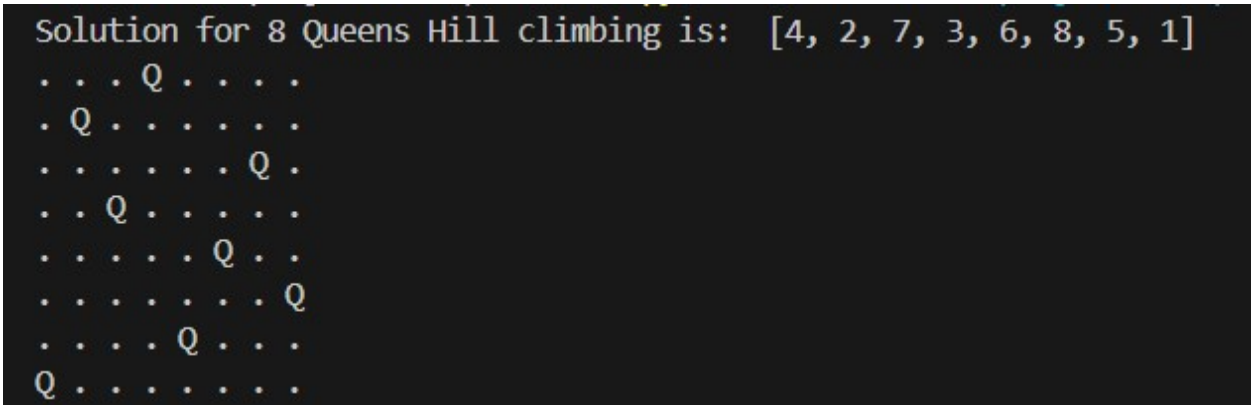
function h(state):
n = 0
for i in range (len(state)
for j in range (i+1, len state)
if abs (state[i] - state[i] == abs (i - j)
(or)
state[i] == state[j]
n += 1
return h

Code:

```
import random
def h(s):
    h = 0
    n = len(s)
    for i in range(n):
        for j in range(i + 1, n):
            if s[i] == s[j] or abs(s[i] - s[j]) == abs(i - j):
                h += 1
    return h

def new(s):
    best=s
    for i in range(len(s)):
        for j in range(1,9):
            if j!=s[i]:
                n=s[:i]+[j]+s[i+1:]
                if h(n)<h(best):
                    best=n
    return best

def hc():
    curr=[random.randint(1,8) for i in range(8)]
    while True:
        ch=h(curr)
        curr=new(curr)
        if h(curr)==0:
            return curr
        if h(curr)>=ch:
            curr=[random.randint(1,8) for i in range(8)]

def print_board(solution):
    print("Solution for 8 Queens Hill climbing is: ",solution)
    if solution is None:
        print("No solution found.")
        return

    board = [['.' for _ in range(8)] for _ in range(8)]

    for row in range(len(solution)):
        col = solution[row] - 1
        board[row][col] = 'Q'
```

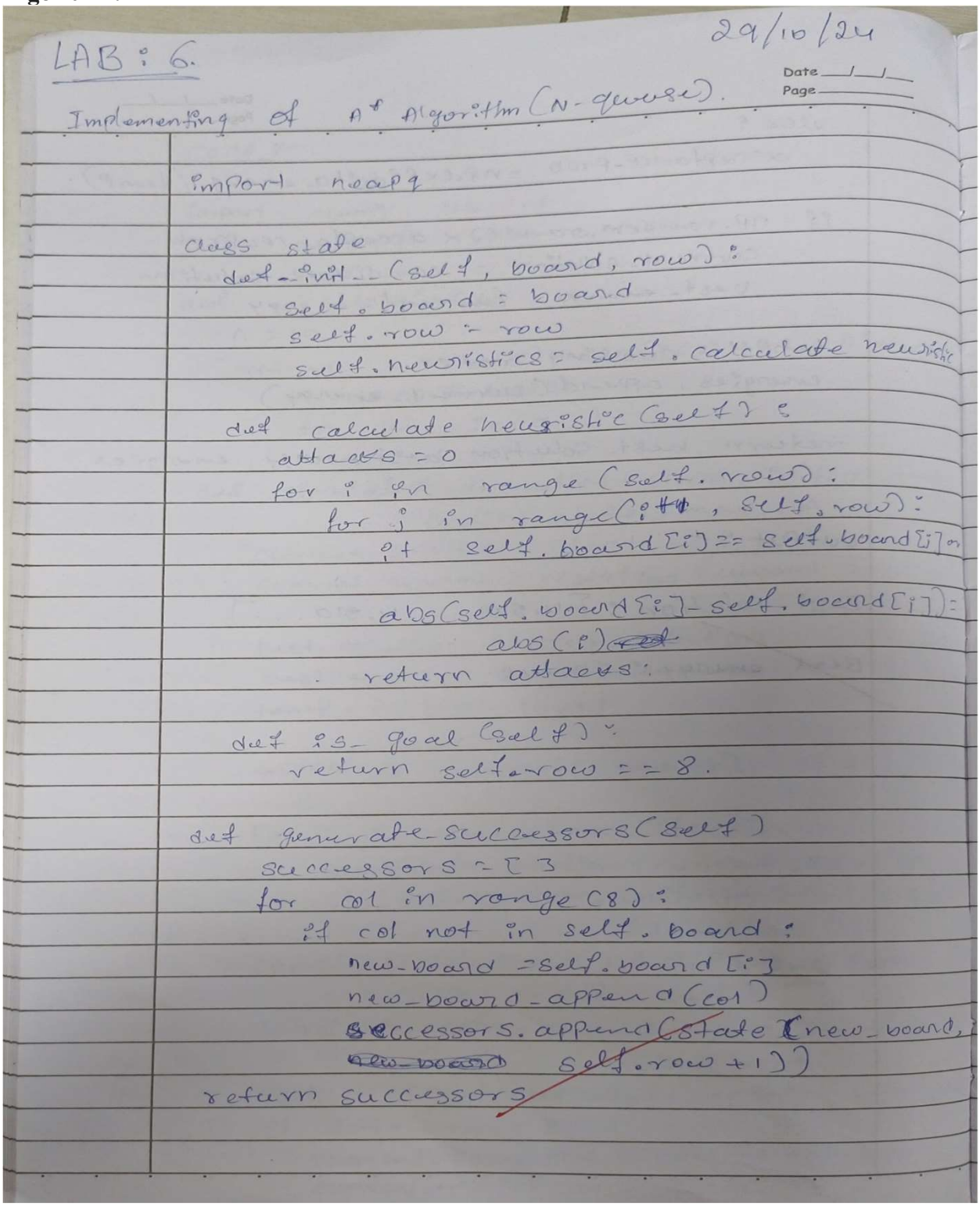
```
    for row in board:
        print(' '.join(row))

print_board(hc())
```

Output:



Implement A star algorithm to solve N-Queens problem.

Algorithm:



LAB : 6.

29/10/24

Implementing of A* Algorithm (N-queens).

```
import heapq

class state
    def __init__(self, board, row):
        self.board = board
        self.row = row
        self.heuristics = self.calculate heuristic

    def calculate heuristic(self):
        attacks = 0
        for i in range(self.row):
            for j in range(i+1, self.row):
                if self.board[i] == self.board[j] or
                    abs(self.board[i] - self.board[j]) =
                        abs(i)
        return attacks:

    def is_goal(self):
        return self.row == 8.

    def generate_successors(self)
        successors = []
        for col in range(8):
            if col not in self.board:
                new_board = self.board[:]
                new_board.append(col)
                successors.append(state(new_board,
                    self.row + 1))
        return successors
```

Date __/__/__
Page _____

```
def a_star_8_queens():
    initial_state = state([], 0)
    open_set = []
    heapq.heappush(open_set, (initial_state.
                    heuristic, initial_state))
    while open_set:
        current_state = heapq.heappop(open_set)

    if current_state.is_goal():
        print_solution(current_state, generate_
                        successors():
                    heapq.heappush(open_set, (successor.
                    heuristic + len(successor_board),
                        successors))
        print("NO solution exists")
        return False.

if __name__ == "__main__"
    a_star_8_queens()
```

Implementing hill climbing method (8-queens)

code :-

```
import random

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

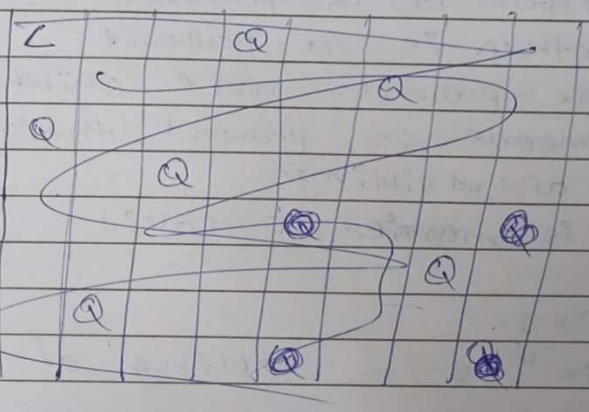
Date __/__/__
Page _____

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

def hill_climbing(state):
    current = state
    while True:
        neighbors = get_neighbors(current)
        current_h = heuristic(current)
        best_h = current_h
        best_neighbor = current

        for neighbor in neighbors:
            h = heuristic(neighbor)
            best_h = current_h
            best_neighbor = current

        for neighbor in neighbors:
            h = heuristic(neighbor)
            if h < best_h:
                best_h = h;
                best_neighbor = neighbor
                if best_h >= current_h:
                    break
        current = best_neighbors.
    return current, heuristic(current)
```

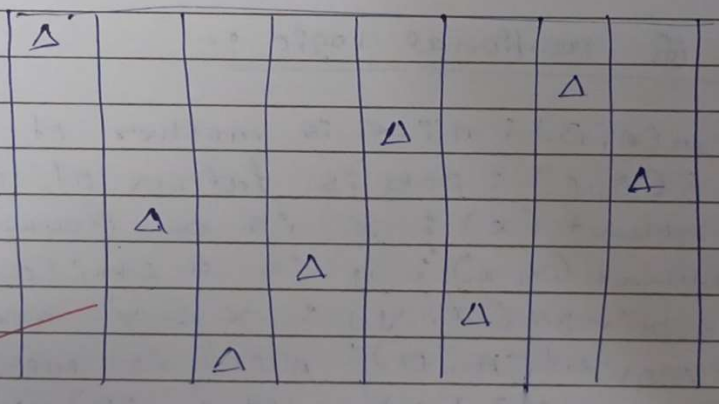
Date __/__/__
Page _____

```
initial_state = [random.randint(0,7)
    for i in range(8)]

solution = final_neurestic = hill_climbing(
        initial_state)
Print("solution", solution)
```

Output :-



Hill climb :-
random placem
then finding
optimization

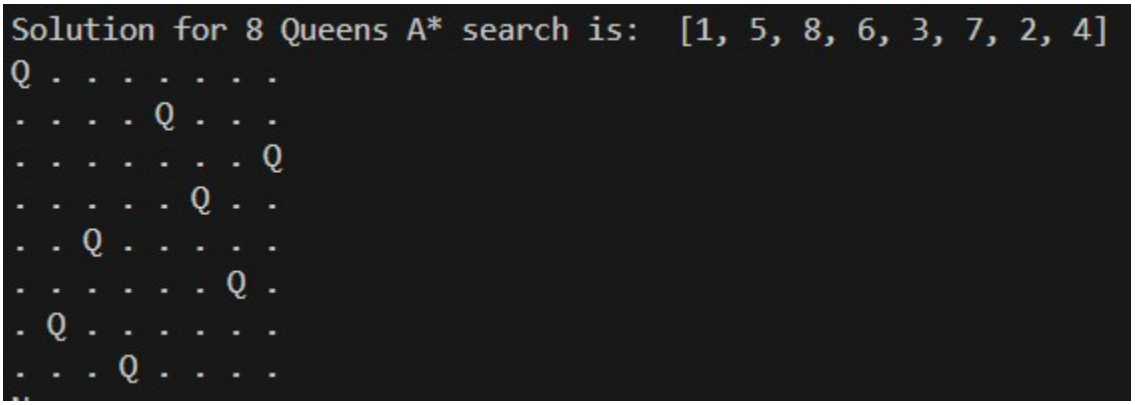


Code:

```
import heapq


def h(s):


    h = 0
    n = len(s)
    for i in range(n):
        for j in range(i + 1, n):
            if s[i] == s[j] or abs(s[i] - s[j]) == abs(i - j):
                h += 1
    return h


def a_star():
    initial_state = []
    q = []
    g = 8
    heapq.heappush(q, (h(initial_state), initial_state, g))


    while q:
        f, state, g = heapq.heappop(q)


        if len(state) == 8 and h(state) == 0:
            return state


        for i in range(1, 9):
```

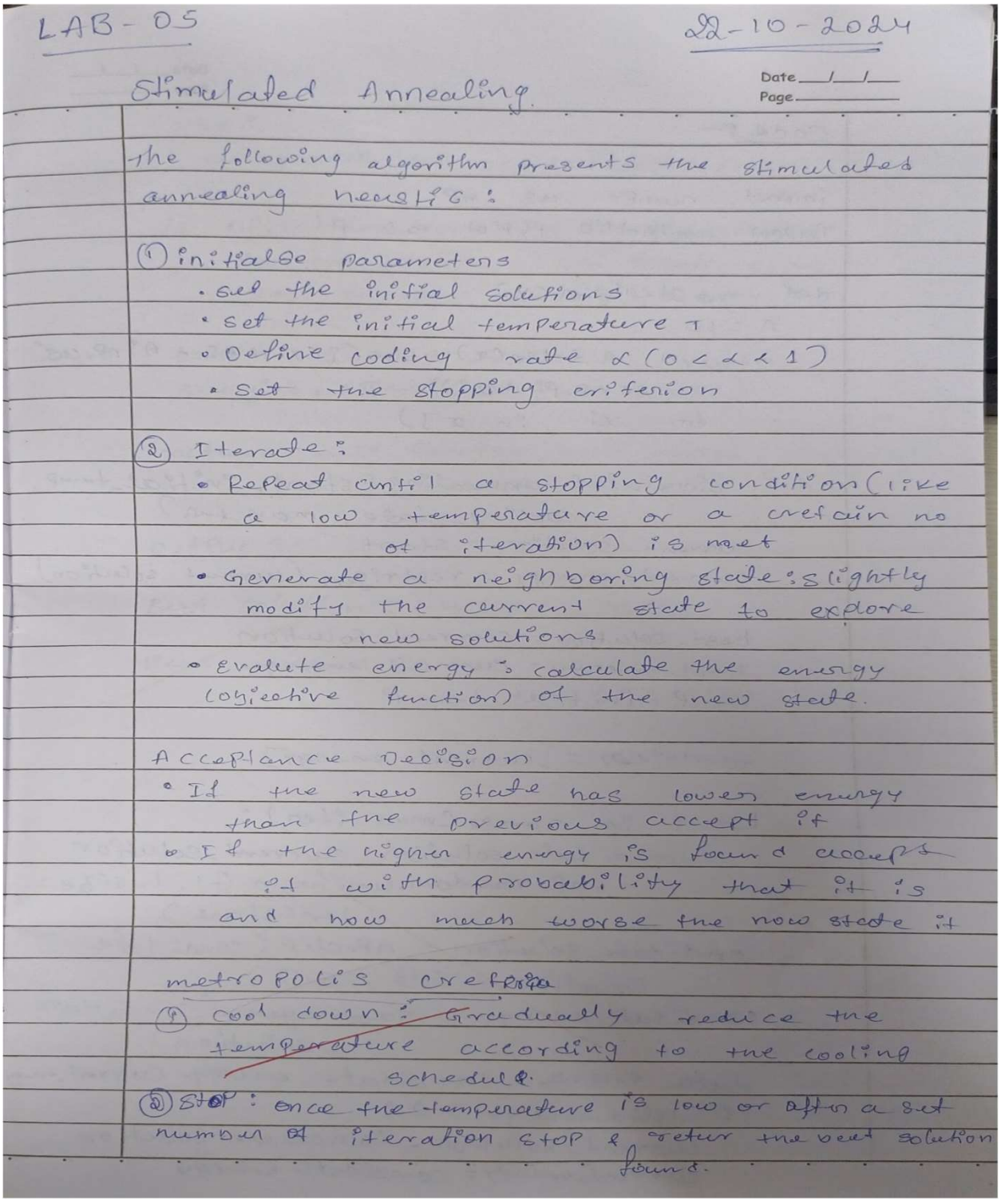
```
            if i not in state:
                new_state = state + [i]
                heapq.heappush(q, (h(new_state) + g, new_state, g - 1))

    return None

solution = a_star()
print("Solution:", solution)
```

Output:



Implement Simulated Annealing:

Algorithm:



LAB - 05 22-10-2024

Simulated Annealing

Date __/__/__
Page

the following algorithm presents the simulated annealing heuristic:

(1) initialise parameters
- Get the initial solutions
- set the initial temperature T
- Define coding rate α ($0 < \alpha < 1$)
- Set the stopping criterion

(2) Iterate:
- Repeat until a stopping condition (like a low temperature or a certain no of iteration) is met
- Generate a neighboring state: slightly modify the current state to explore new solutions
- Evaluate energy: calculate the energy (objective function) of the new state.

Acceptance Decision
- If the new state has lower energy than the previous accept it
- If the higher energy is found accept it with probability that it is and how much worse the new state it

metropolis criteria

(1) cool down: Gradually reduce the temperature according to the cooling schedule.

(2) Stop: once the temperature is low or after a set number of iteration stop & return the best solution found.

Code :-

```
import numpy as np
import matplotlib.pyplot as plt

def rastrigin(x):
    A = 10
    return A * len(x) + sum([(xi ** 2 - A * np.cos(2 * np.pi * xi)) for xi in x])

def simulated_annealing(start, initial_temp, cooling_state, max_iter):
    current_solution = start
    current_energy = rastrigin(current_solution)

    best_solution = current_solution
    best_energy = current_energy
    temp = initial_temp

    energies = [current_energy]

    for i in range(max_iter):
        candidate_solution = current_solution + np.random.uniform(-1, 1, size=len(start))
        candidate_solution = np.clip(candidate_solution, -5.12, 5.12)
        candidate_energy = rastrigin(candidate_solution)
        delta_energy = candidate_energy - current_energy
        if delta_energy > 0:
            current_solution = candidate_solution
            current_energy = candidate_energy
```

Date __/__/__
Page _____

```
else:
    acceptance_prob = np.exp(delta_energy / temp)

if np.random.rand() < acceptance_prob
    current_solution = candidate_solution
    best_energy = current_energy

temp *= cooling_rate
energies.append(current_energy)

return best_solution, best_energy, energies
```

Output :-

Best solution : [-4.540... 4.514...]

Best energy : 80.642.

Code:

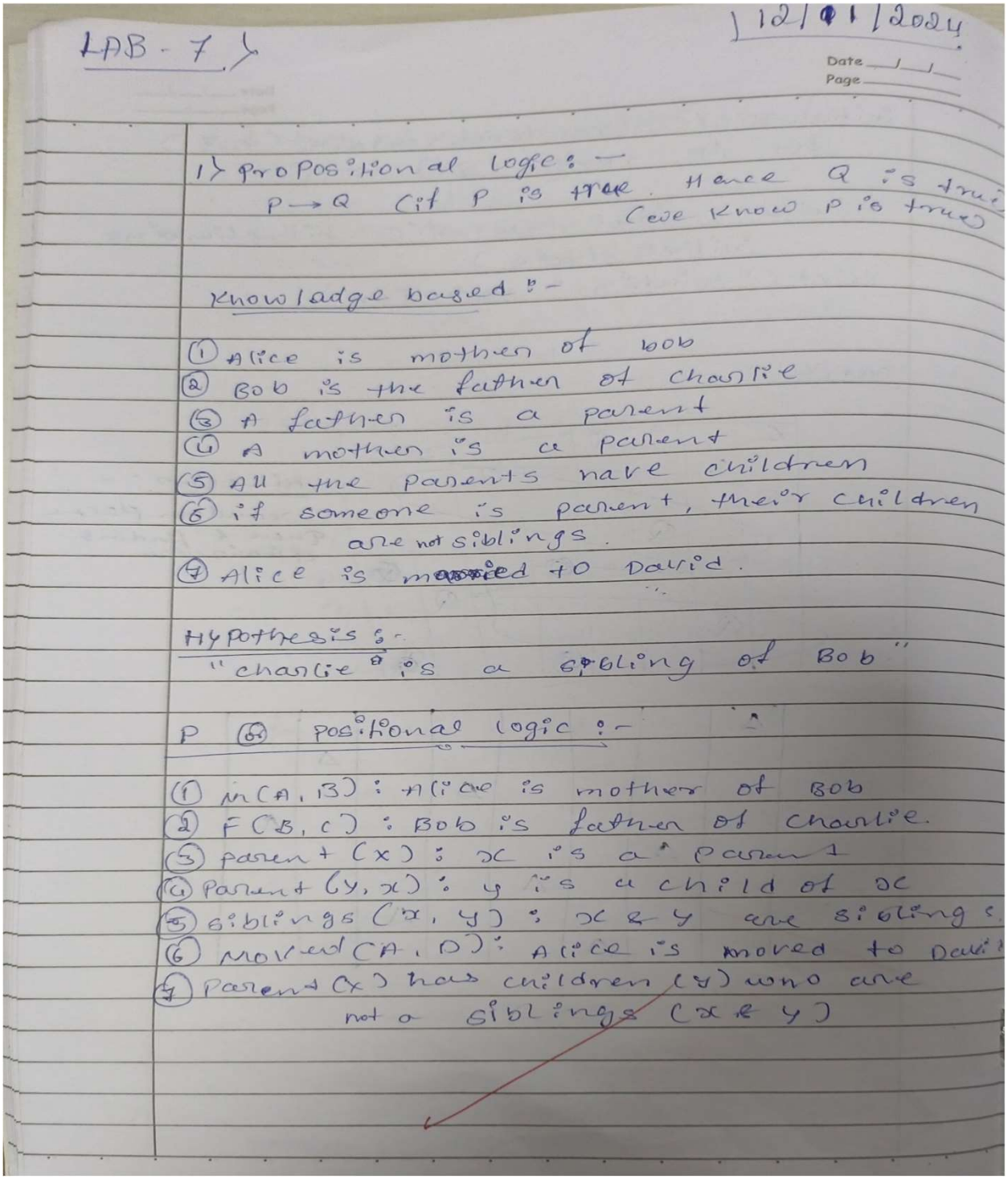
```
import random
import math

def sim_anneal(ini, in_temp, max_iter, cool):
    # Initialize current state and best state
    curr_s = ini
    best_s = curr_s
    best_c = obj(best_s)
    temp = in_temp  # Set the initial temperature

    # While the temperature is above a threshold
    while temp > 1:
        for i in range(max_iter):
            new_s = neig(curr_s)
            curr_c = obj(curr_s)
            new_c = obj(new_s)

            if ap(curr_c, new_c, temp) > random.random():
                curr_s = new_s  # Move to the new state
            if new_c < best_c:
                best_s = new_s
                best_c = new_c
        temp *= cool

    return best_s, best_c

def neig(state):
    new_s = state.copy()
    ind = random.randint(0, len(state) - 1)
    new_s[ind] += random.uniform(-1, 1)
    return new_s

def obj(state):
    c = 1
    for i in state:
        c += i**2 + 2*i + 1
    return c

def ap(curr_c, new_c, temp):
    if new_c < curr_c:
        return 1
    else:
        return math.exp((curr_c - new_c) / temp)

print(sim_anneal([1, 2, 3, 4, 5], 1000, 1000, 0.99))
```

Output:

```
[Running] python -u "c:\Users\bmsce\Desktop\san.py"
([-0.97275454497846, -1.036978056021493, -1.0024102215924622, -1.059180212134072, -0.9858194523412274], 0.0058188860547206955)
```

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

Algorithm:



Date __/__/__
Page _____

logical reasoning :-

① From statement ① & ④
M(A, B) & (y, x) → Alice is Parent

② From statement ② & ④
F(B, C) & (y, x) → Bob is a Parent

③ from statement ① & ② & ⑦

M(A, B) & F(B, C) & (∀ & (x, y)) →
→ bob & charlie are not siblings.

Prob
19/9/24

Code:

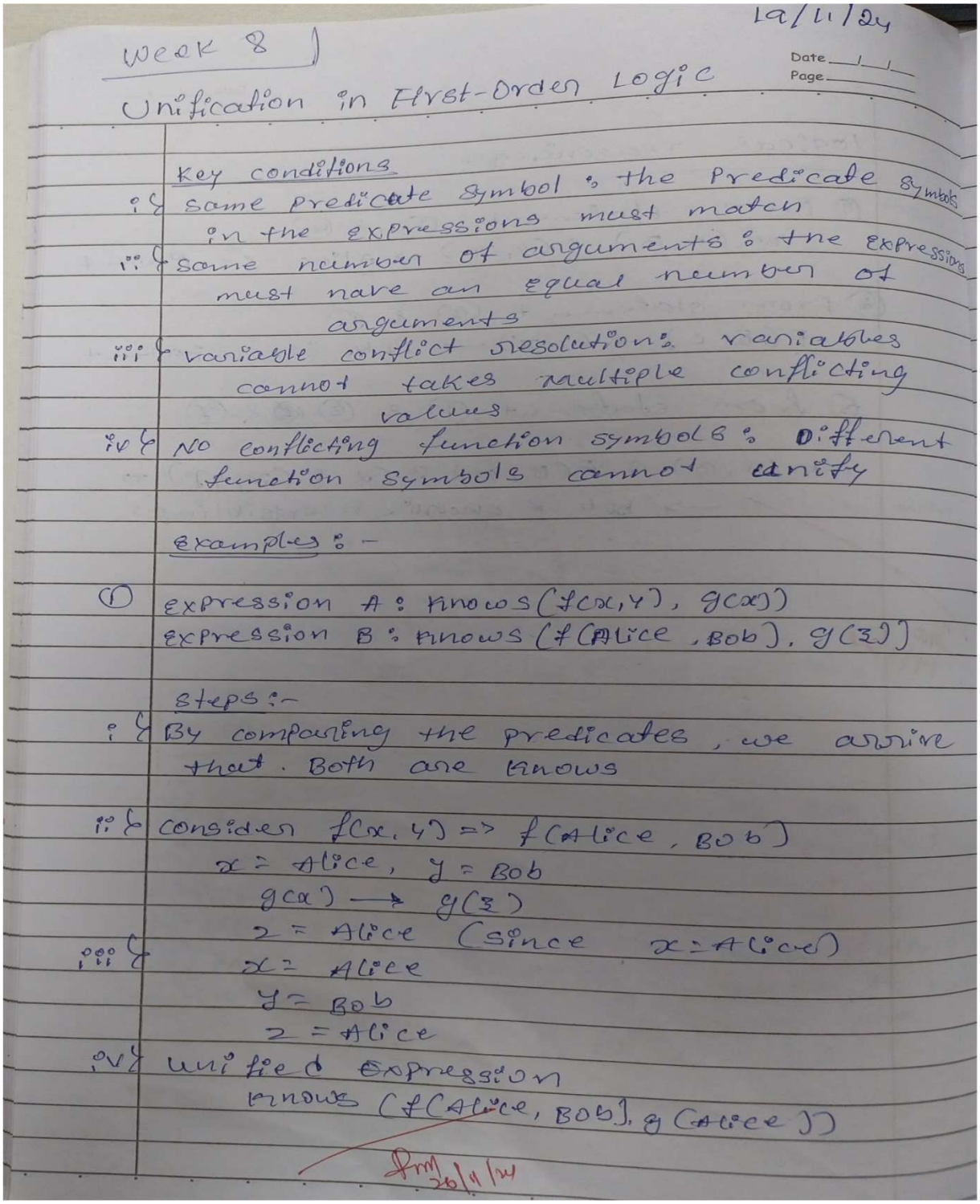
```
from sympy.logic.boolalg import Or, And, Not
from sympy.abc import A, B, C, D, E, F
from sympy import simplify_logic

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:

```
PS C:\Users\prajw\Desktop\AI-Lab> python -u "c:\Users\prajw\Desktop\AI-Lab\Week7\entail.py"
Is the query 'C | F' entailed by the knowledge base? Yes
```

Implement unification in first order logic.

Algorithm:



Week 8

19/11/24

Unification in First-Order Logic

Key conditions

i) Same Predicate Symbol : the Predicate Symbols in the expressions must match

ii) Same number of arguments : the expressions must have an equal number of arguments

iii) Variable conflict resolution : variables cannot takes multiple conflicting values

iv) No conflicting function symbols : Different function symbols cannot unify

Examples :-

(1) Expression A : Knows(f(x,y), g(x))
Expression B : Knows(f(Alice, Bob), g(z))

Steps :-

i) By comparing the predicates, we arrive that Both are Knows

ii) Consider f(x, y) => f(Alice, Bob)
x = Alice, y = Bob
g(x) → g(z)
z = Alice (since x = Alice)

iii) x = Alice
y = Bob
z = Alice

iv) Unified Expression
Knows(f(Alice, Bob), g(Alice))

Rm 26/11/24

/24

Date __/__/__
Page _____

code :-

```
Knowledge_base = [
{"type": "rule", "rule": "∀x∀y (Doctor(x) ∧ Sickly(y) → Treats(x,y))"},
{"type": "fact", "fact": Doctor(Sopheen)"},
{"type": "fact", "fact": ∃x (Doctor(x) → ∃h (Hospital(h) ∧
        WorksAt(x,h)))"},
{"type": "fact", "fact": Hospital(GeneralHospital)"},
{"type": "fact", "fact": "WorksAt(John, GeneralHospital)"},
]

query = {"predict": "Treats", "arguments": ["?", "Mary"]}.

def unify(kb, query):
    predicate = query["predicate"]
    target_args = query["arguments"][1]
    result = None.

for item in kb:
    if item["type"] == "rule" and predicate in item["rule"]:
    rule = item["rule"]
    if "Doctor(x)" in rule and "Sickly" in rule:
        doctor = None
        sick_person = None.

for fact in kb:
    if fact["type"] == "fact" and "Doctor" in fact["fact"]:
        doctor = fact["fact"].split("(")[1][:-1]
    if fact["type"] == "fact" and "Sick" in fat["fat"],
    sick_person = fact["fact"].split("(")[1][:-1]
```

LAB 9

Date __/__/__
Page _____

First

```
if result:
    return f"The query '{query['predicate']}({result}, {target_args})' is unified: {result} treats {target_arg2}."
else:
    return "The query '{query['predicate']}({query['arguments'][0]}, {target_args})' could not required with knowledge base."

result = unify(knowledge_base, query)
print(result)
```

Output :-

The query can't Access?, Project x' is unified:
John Treats Mary

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):
```

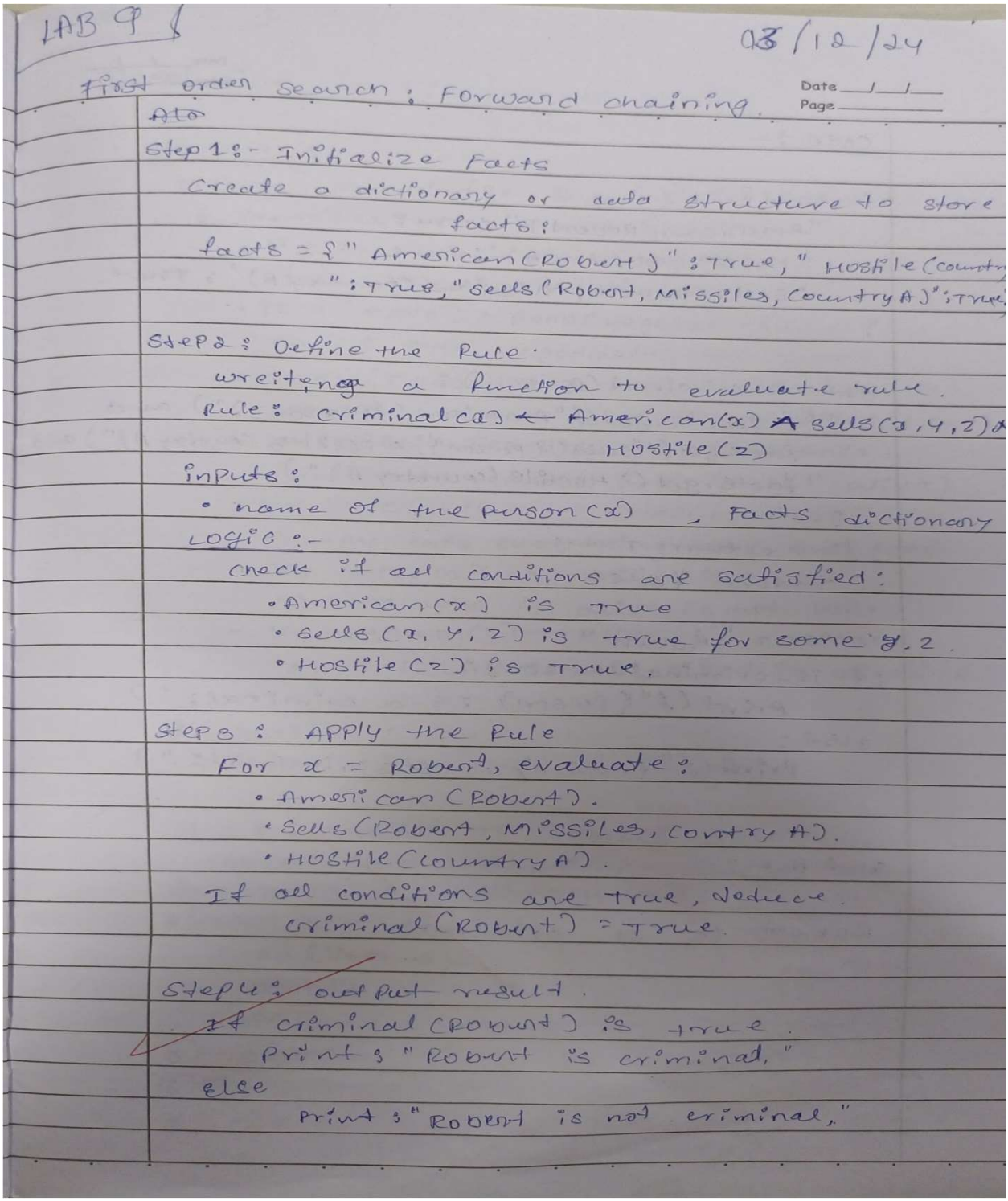
```
    result = unify(expr1, expr2)
    if result == "FAILURE":
        return False, None
    return True, result

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:

```
Output: 1BM22CS200
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:



LAB 9 08/10/24

First order search : Forward chaining.

Date __/__/__
Page

Step 1:- Initialize Facts

Create a dictionary or data structure to store facts:

facts = {"American(Robert)": True, "Hostile(country": True, "Sells(Robert, Missiles, Country A)": True

Step 2: Define the Rule.

writing a function to evaluate rule.

Rule: Criminal(x) ← American(x) ∧ Sells(x,y,z) ∧ Hostile(z)

inputs:

- name of the person(x) , Facts dictionary

Logic :-

Check if all conditions are satisfied:

- American(x) is true
- Sells(x, y, z) is true for some y, z.
- Hostile(z) is true.

Step 3: Apply the Rule

For x = Robert, evaluate:

- American(Robert).
- Sells(Robert, Missiles, Country A).
- Hostile(Country A).

If all conditions are true, deduce Criminal(Robert) = True.

Step 4: Output result.

If criminal(Robert) is true.

Print: "Robert is criminal."

else

Print: "Robert is not criminal."

Code :-

```
facts = {
    "American(Robert)": True,
    "Hostile(Country A)": True,
    "sells(Robert, Missiles, Country A)": True,
}

def is_criminal(person):
    if (facts.get(f"American({person})") and
        facts.get(f"sells({person}, Missiles, Country A)") and
        facts.get("Hostile(Country A)")
    ):
        return True
    return False

person = "Robert"
if is_criminal(person):
    print(f"{person} is a criminal.")
else:
    print(f"{person} is not a criminal.")
```

Out Put :-

Robert is a criminal.

Code:

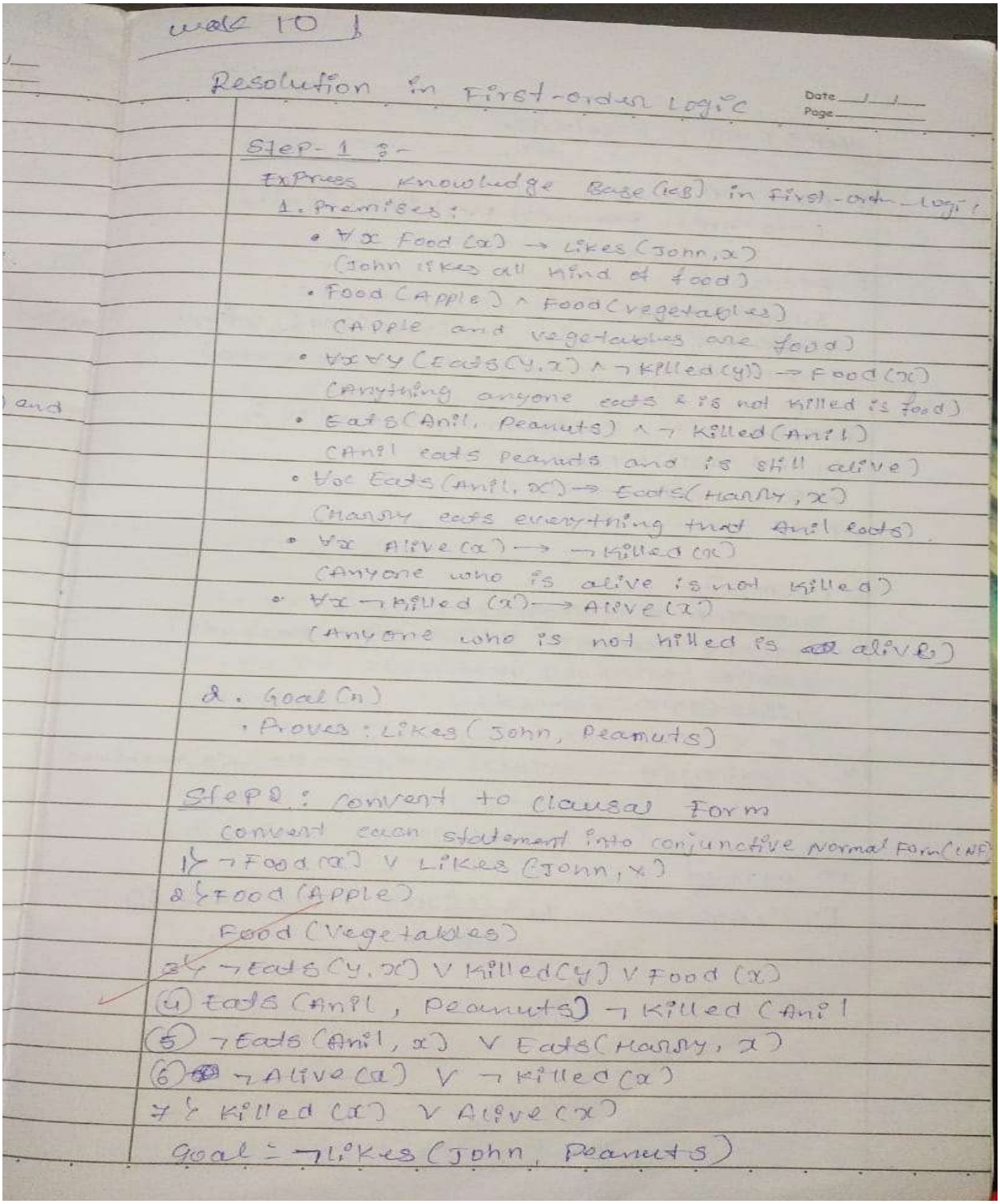
```
KB = set()

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:

```
PS C:\Users\prajw\Desktop\AI-Lab> python -u "c:\Users\prajw\Desktop\AI-Lab\Week8\tempCodeRunnerFile.py"
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:



Date / /
Page

Step 3 : Apply Resolution

(6) From (4) :
Eats (Anil, Peanuts) ∧ ¬Killed (Anil)

(7) From (3) :
Substitute y = Anil, x = Peanuts :
¬Eats (Anil, Peanuts) ∨ Killed (Anil) ∨ Food (Peanuts)

Resolve with Eats (Anil, Peanuts) :
Killed (Anil) ∨ Food (Peanuts)
Resolve with ¬Killed (Anil) :
Food (Peanuts)

(8) From (1) :
Substitute x = Peanuts :
¬Food (Peanuts) ∨ Likes (John, Peanuts)
Resolve with Food (Peanuts) :
Likes (John, Peanuts)

(9) Negation of goal ¬Likes (John, Peanuts) is resolved

→ Output
By resolution, Likes (John, Peanuts) is proven.

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: 1BM22CS200")
print(f"Does Socrates write books? {'Yes' if result else 'No'}")
```

Output:

```
PS C:\Users\prajw\Desktop\AI-Lab> python -u "c:\Users\prajw\Desktop\AI-Lab\Week7\resloution.py"
Output: 1BM22CS200
Does Socrates write books? Yes
```

Implement MinMax Algorithm for TicTacToe.

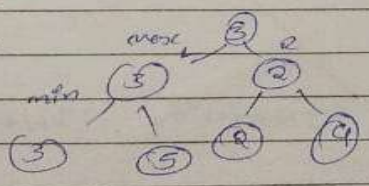
Algorithm:

Min MAX Algorithm

Date / /
Page

Algorithm code:-

```
import math

def minmax(curDepth, nodeIndex, maxTurn
        scores, targetDepth):
    if (curDepth == targetDepth):
        return scores[nodeIndex]

    if (maxTurn):
        return max(minimax(curDepth+1, nodeIndex
                *2, False, scores, targetDepth),
            minimax(curDepth+1, nodeIndex*2+1,
                False, scores, targetDepth))
    else:
        return min(minimax(currentDepth+1, nodeIndex
                *2, true, scores, targetDepth),
            minimax(curDepth+1, nodeIndex*2+1,
                true, scores, targetDepth))

scores = [3, 5, 2, 9, 12, 5, 23, 23]

treeDepth = math.log(len(scores), 2)
print("The optimal value is", end=" ")

print(minimax(0, 0, true, scores, treeDepth))
```



Code:

```
import math


def printBoard(board):
    for row in board:
        print(" | ".join(cell if cell != "" else " " for cell in row))
        print("-" * 9)


def evaluateBoard(board):
    for row in board:
        if row[0] == row[1] == row[2] and row[0] != "":
            return 10 if row[0] == 'X' else -10
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] and board[0][col] != "":
            return 10 if board[0][col] == 'X' else -10
    if board[0][0] == board[1][1] == board[2][2] and board[0][0] != "":
        return 10 if board[0][0] == 'X' else -10
    if board[0][2] == board[1][1] == board[2][0] and board[0][2] != "":
        return 10 if board[0][2] == 'X' else -10
    return 0


def isDraw(board):
    for row in board:
        if "" in row:
            return False
    return True
```

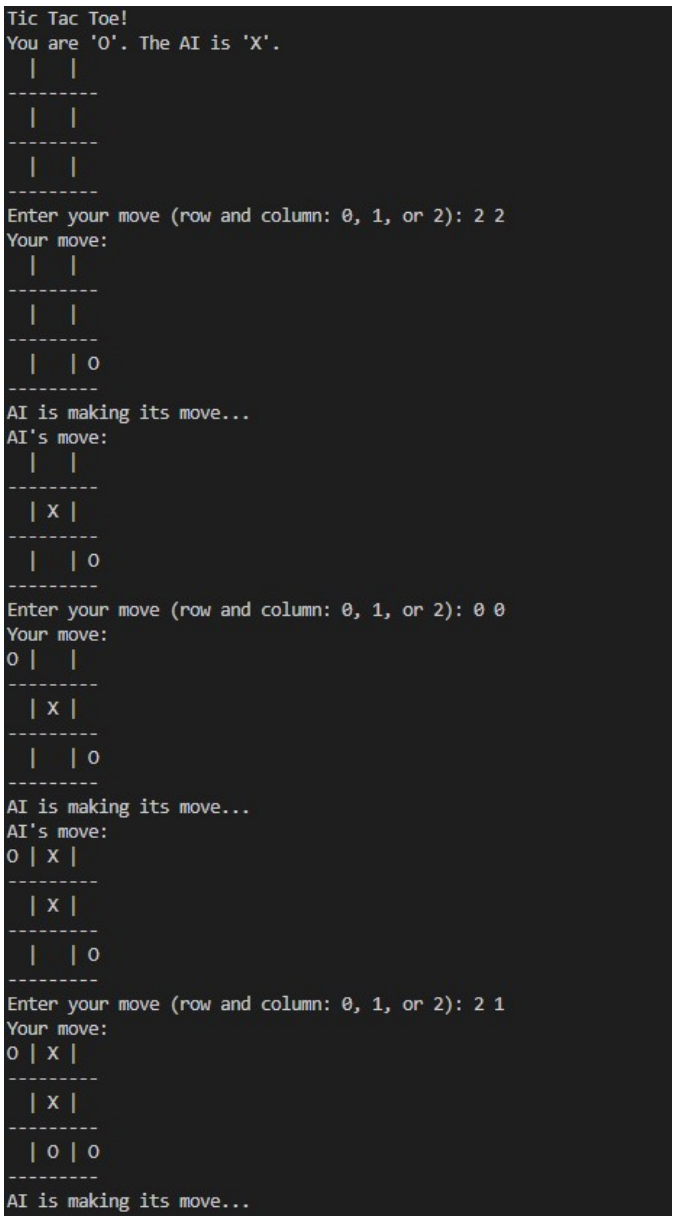
```
def minimax(board, depth, isMaximizing):
    score = evaluateBoard(board)
    if score == 10 or score == -10:
        return score
    if isDraw(board):
        return 0

    if isMaximizing:
        bestScore = -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] = ""
                    bestScore = max(bestScore, score)
        return bestScore
    else:
        bestScore = 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] = ""
                    bestScore = min(bestScore, score)
        return bestScore
```

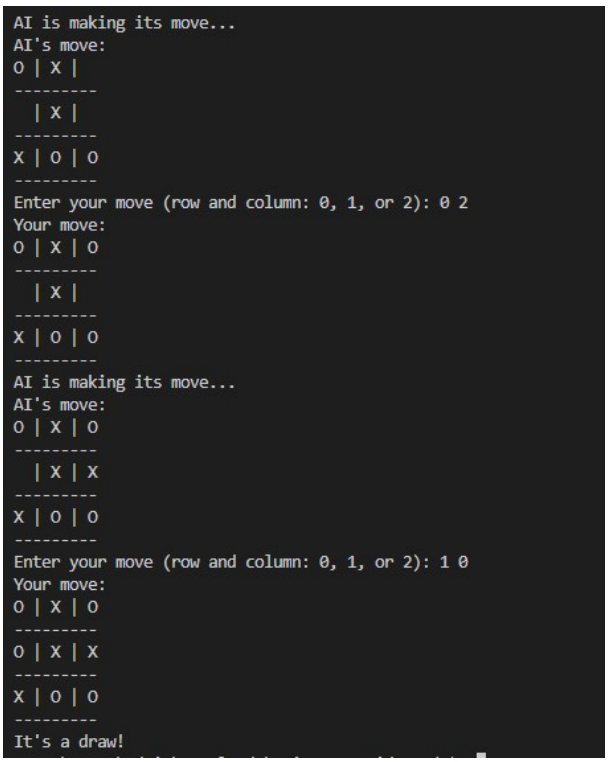
```
def findBestMove(board):
    bestValue = -math.inf
    bestMove = (-1, -1)
    for i in range(3):
        for j in range(3):
            if board[i][j] == "":
                board[i][j] = 'X'
                moveValue = minimax(board, 0, False)
                board[i][j] = ""
                if moveValue > bestValue:
                    bestMove = (i, j)
                    bestValue = moveValue
    return bestMove

def playGame():
    board = [["" for _ in range(3)] for _ in range(3)]
    print("Tic Tac Toe!")
    print("You are 'O'. The AI is 'X'.")
    printBoard(board)

    while True:
        while True:
            try:
                row, col = map(int, input("Enter your move (row and column: 0, 1, or 2): ").split())
                if board[row][col] == "":
                    board[row][col] = 'O'
```

```
            break
        else:
            print("Cell is already taken. Choose another.")
    except (ValueError, IndexError):
        print("Invalid input. Enter row and column as two numbers between 0 and 2.")

print("Your move:")
printBoard(board)

if evaluateBoard(board) == -10:
    print("You win!")
    break
if isDraw(board):
    print("It's a draw!")
    break

print("AI is making its move...")
bestMove = findBestMove(board)
board[bestMove[0]][bestMove[1]] = 'X'

print("AI's move:")
printBoard(board)

if evaluateBoard(board) == 10:
    print("AI wins!")
    break
if isDraw(board):
```

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

playGame()
```

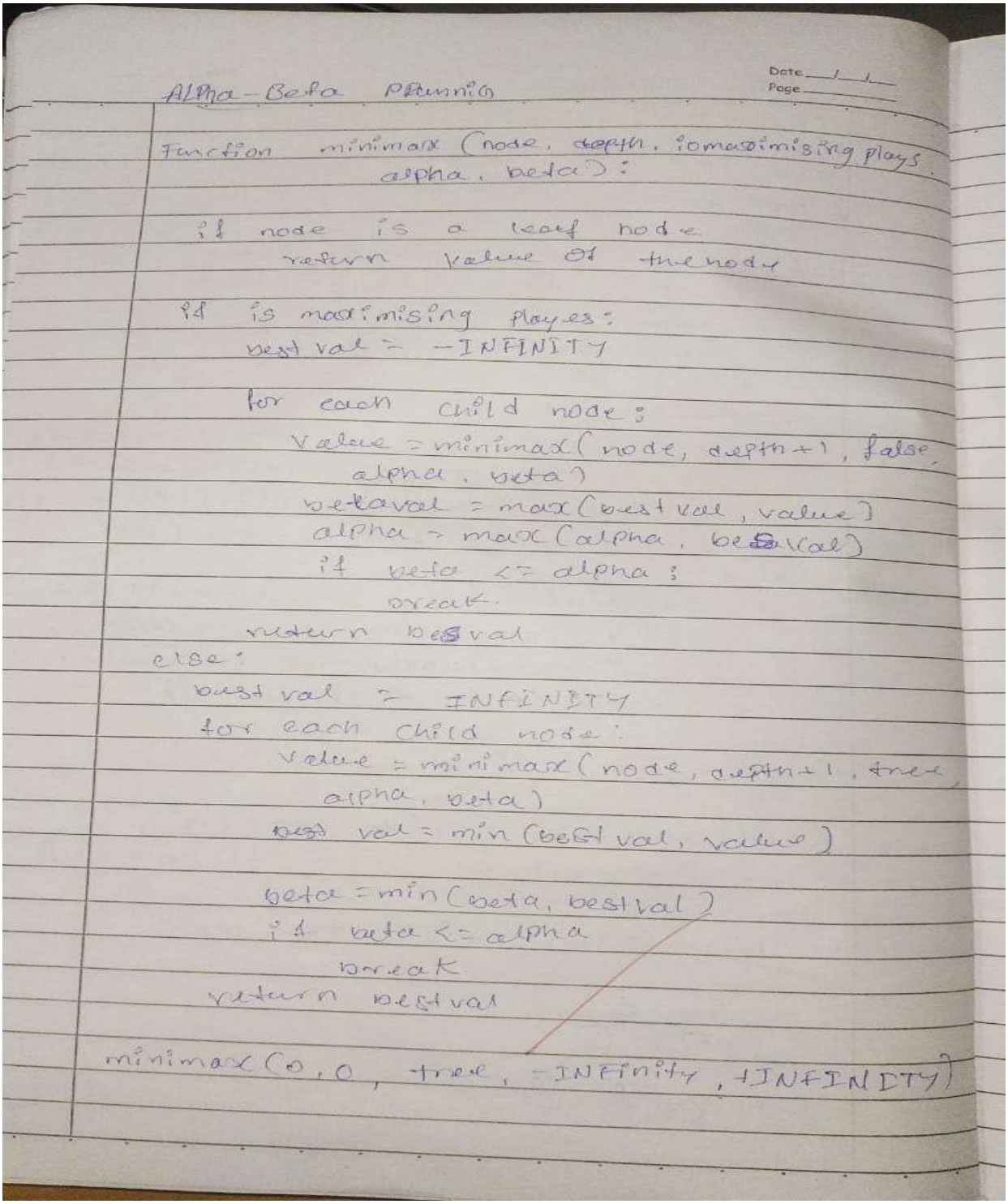
Output:





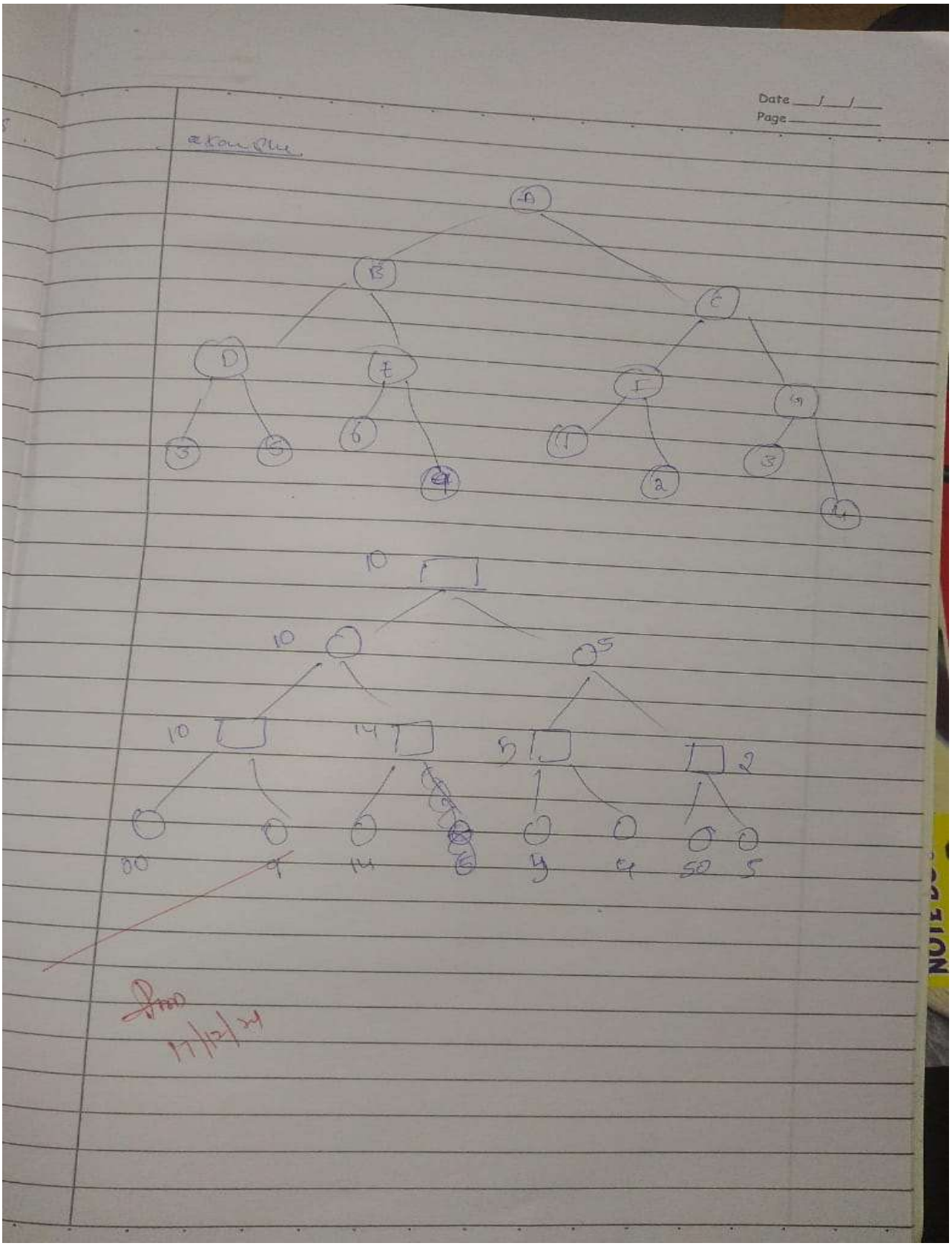
Implement Alpha-Beta Pruning for 8Queens.

Algorithm:



Alpha-Beta Pruning

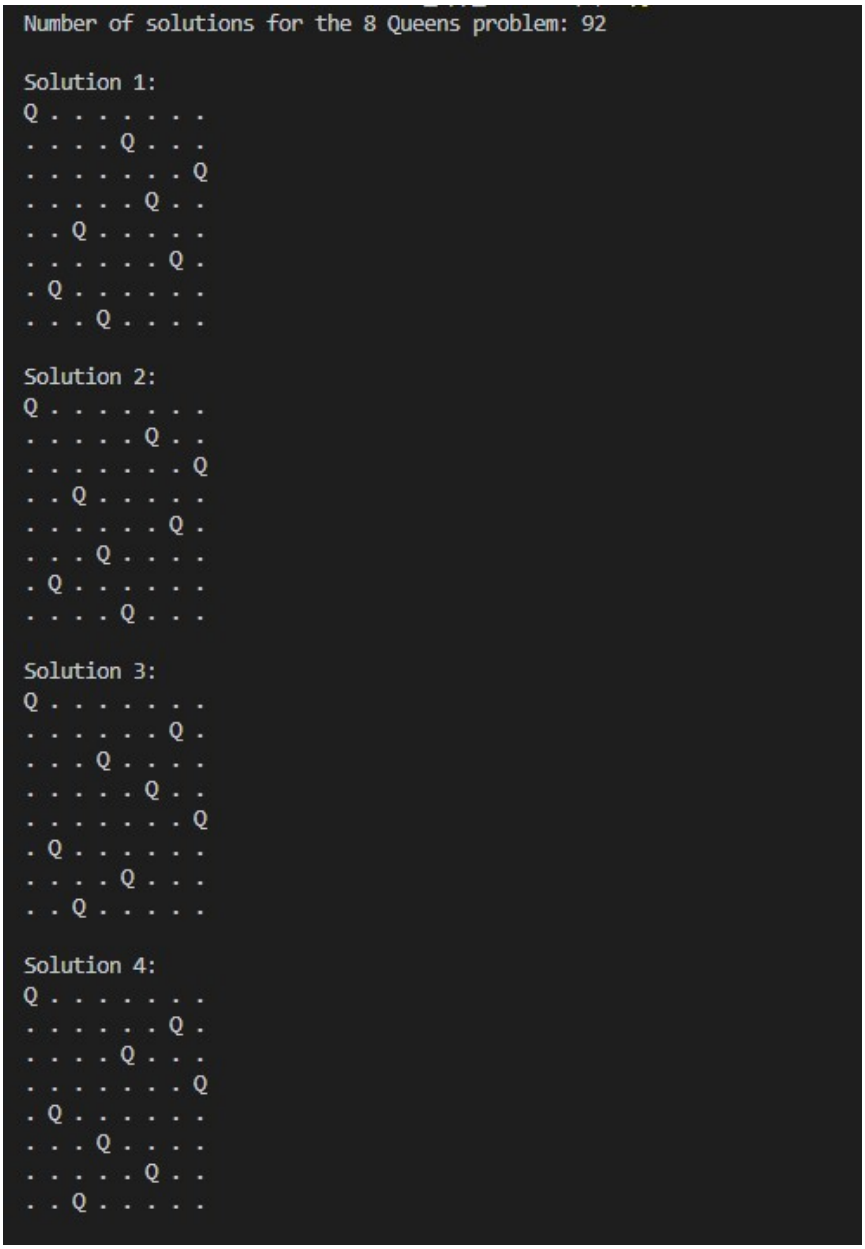
```
Function minimax (node, depth, ismaximising player,
                  alpha, beta):

    if node is a leaf node
        return value of the node

    if is maximising player:
        best val = -INFINITY

        for each child node:
            value = minimax(node, depth+1, false,
                alpha, beta)
            bestval = max(best val, value)
            alpha = max(alpha, best val)
            if beta <= alpha:
                break
        return best val
    else:
        best val = INFINITY
        for each child node:
            value = minimax(node, depth+1, true,
                alpha, beta)
            best val = min(best val, value)

            beta = min(beta, best val)
            if beta <= alpha
                break
        return best val

minimax(0, 0, true, -INFinity, +INFINITY)
```


Date
Page
example
A
B
C
D
E
F
G
H
I
J
K
L
10
10
05
10
14
5
2
00
4
14
6
4
4
50
5

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 . . . . .