VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Prajwal C (1BM22CS198)

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 **Prajwal C (1BM22CS198),** 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.

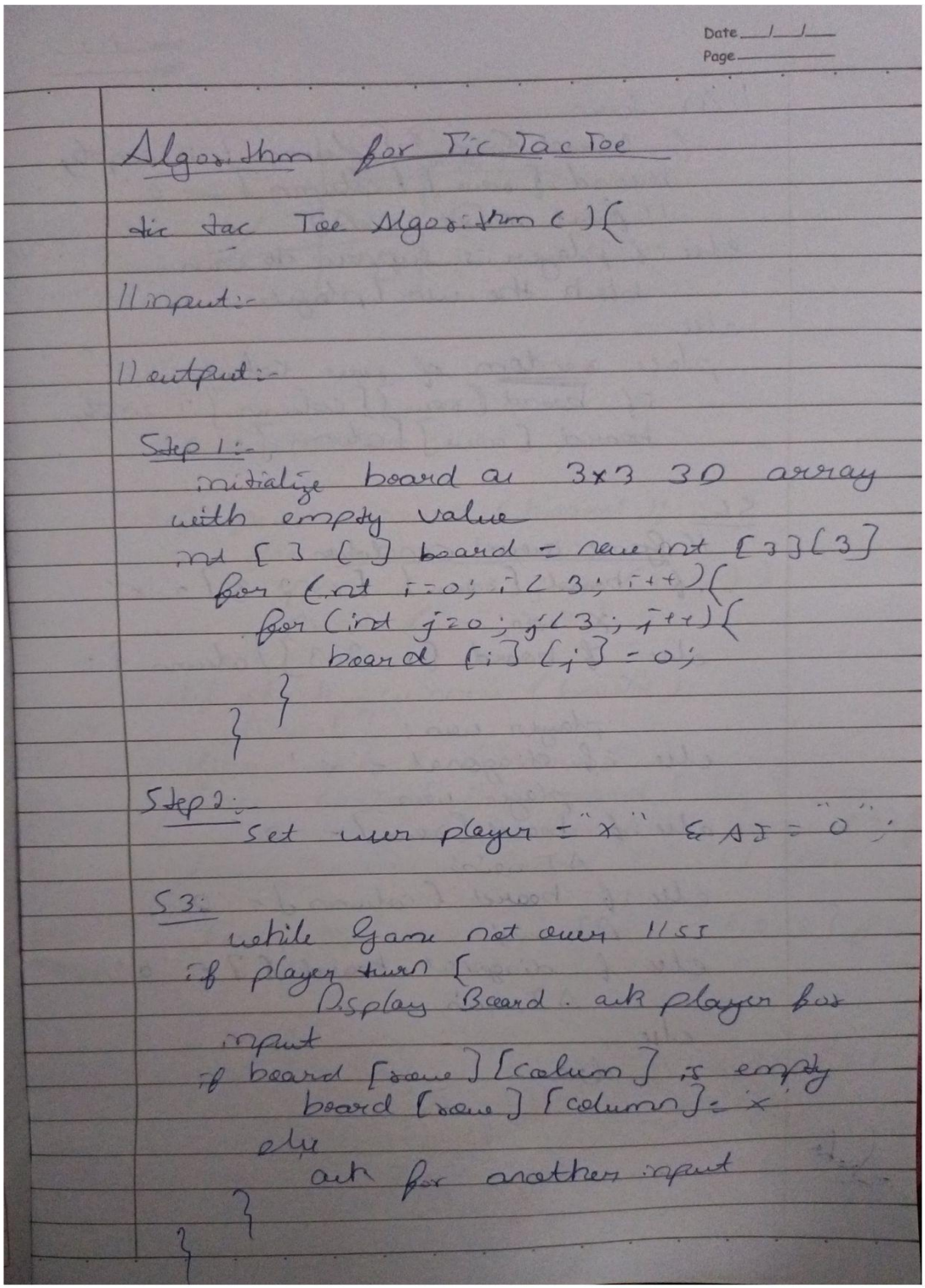
Prof. Sneha S Bagalkot Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
AI Lab Github Link

Implement Tic-Tac-Toe Game.

Algorithm:



Date / /
Page

Algorithm for Tic Tac Toe

tic tac Toe Algorithm ( ){

// input :-

// output :-

Step 1 :-
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] = 0;
}
}

Step 2 :-
Set user player = "x" & AI = "0";

S3:
while Game not over // S1
if player turn {
Display Board. ask player for input
if board [row] [column] is empty
board [row] [column] = x
else
ask for another input
}
}

Date __/__/__
Page _____

// AI turn
if board [row] [column] is empty
board [row] [column] = 'O'
// first move is AI
else if player is suppose to win
block the user/player
else
place random of row column
if board [row] [column] is empty
board [row] [column] = 'O'

S4:
// Game over condition
if board [row] [1, 2 3] = 'x'
x win;
else if board [1, 2, 3] [column] =
'x'
player win;
else if diagonal = 'x'
player win;
else if board [row] = 'O'
AI win;
else if board [column] = 'O'
AI win;
else if diagonal. board [] = 'O'
AI win
else
draw

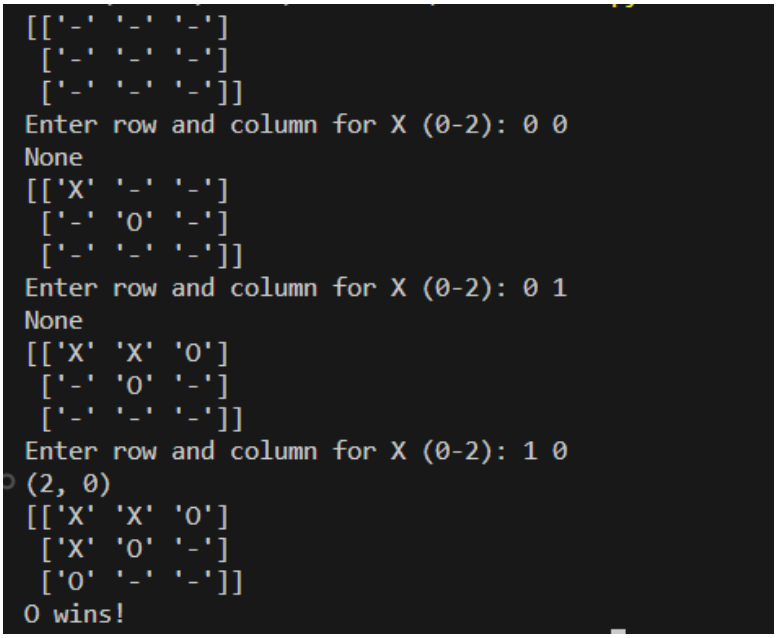
Code:

```
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
```

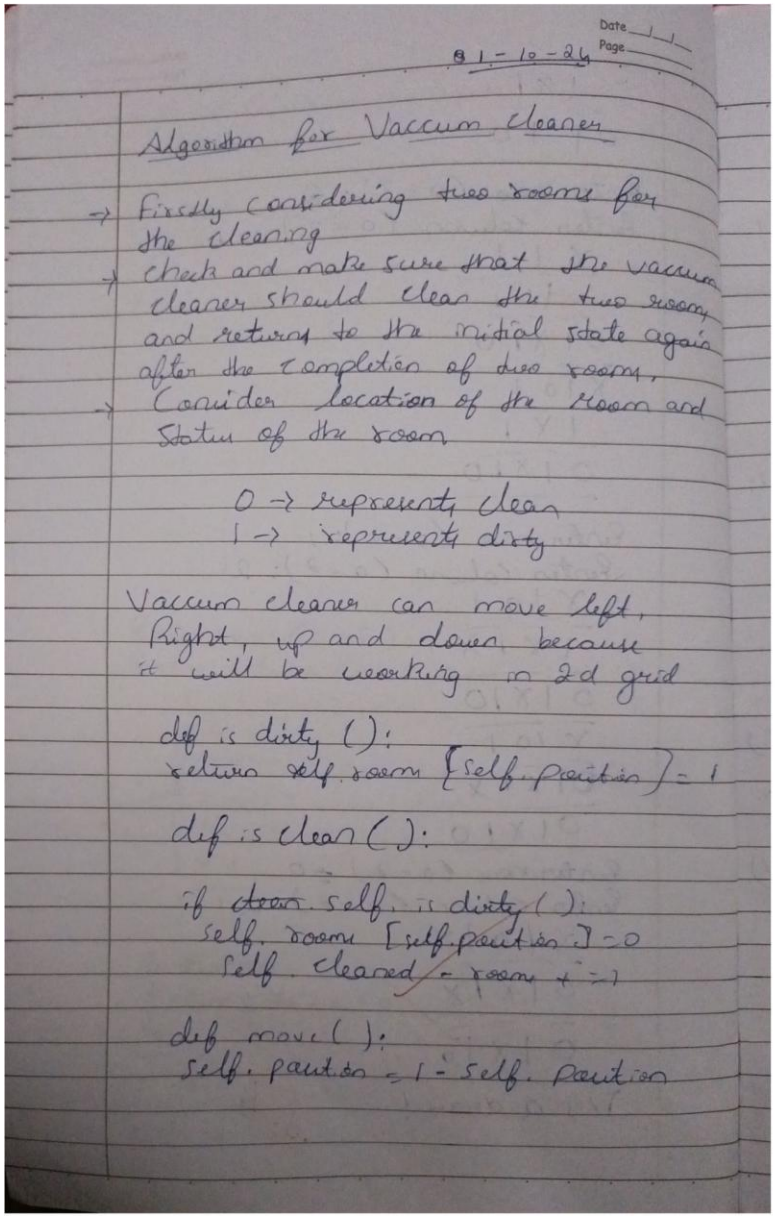
```
board[u[0]][u[1]] = "X"

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:



Date 1/1
Page
8 | - 10 - 24

Algorithm for Vaccum cleaner

→ Firstly considering two rooms for the cleaning
→ 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 rooms,
→ Conider location of the room and Status of the room

0 → represents clean
1 → represents dirty

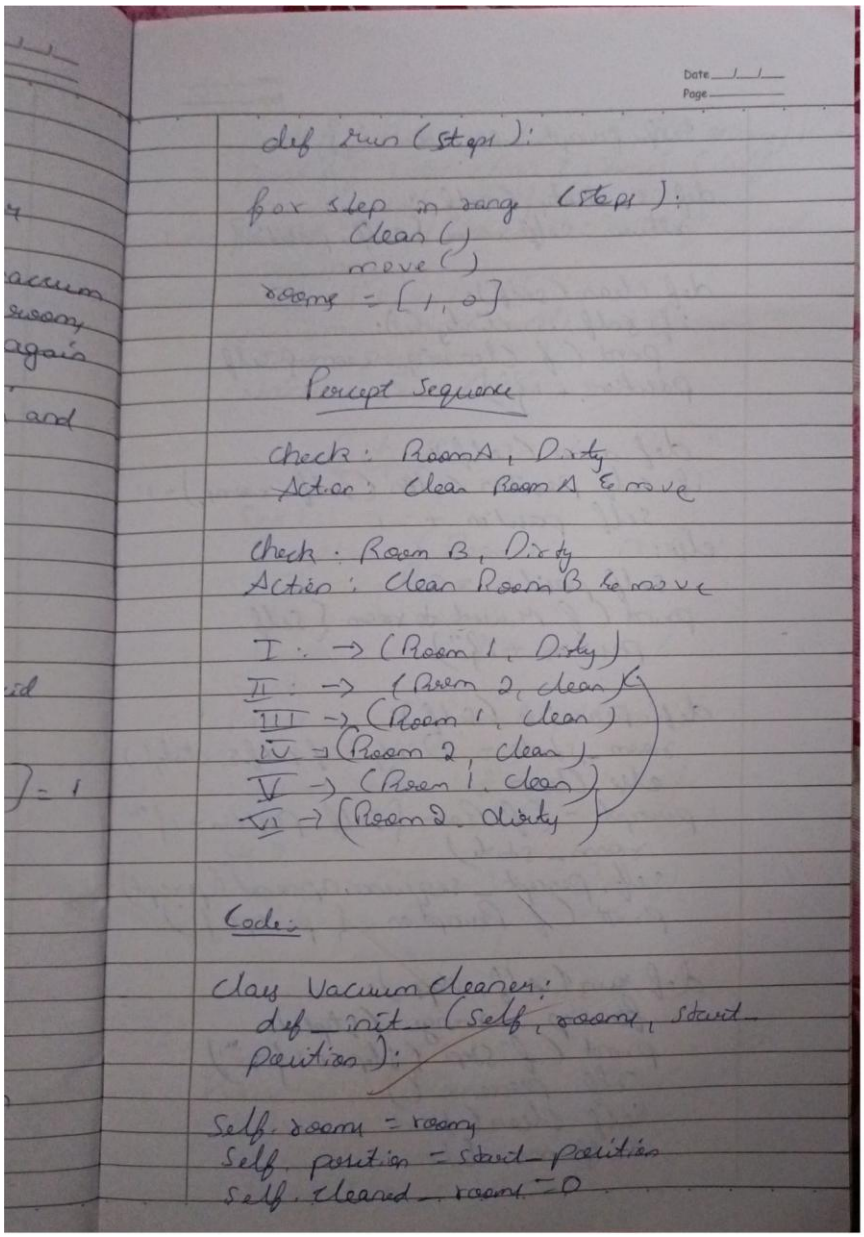
Vaccum cleaner can move left, Right, up and down because it will be working in 2d grid

def is dirty ():
return self.room [self.position] = 1

def is clean ():

if ~~room~~.self. is dirty ():
self. room [self.position] = 0
self. cleaned = room + = 1

def move ():
self. position = 1 - self. position



Code:

```
import random

l=[random.choice([0,1]),random.choice([0,1])]

def check(i):

  if l[i]==0:
```

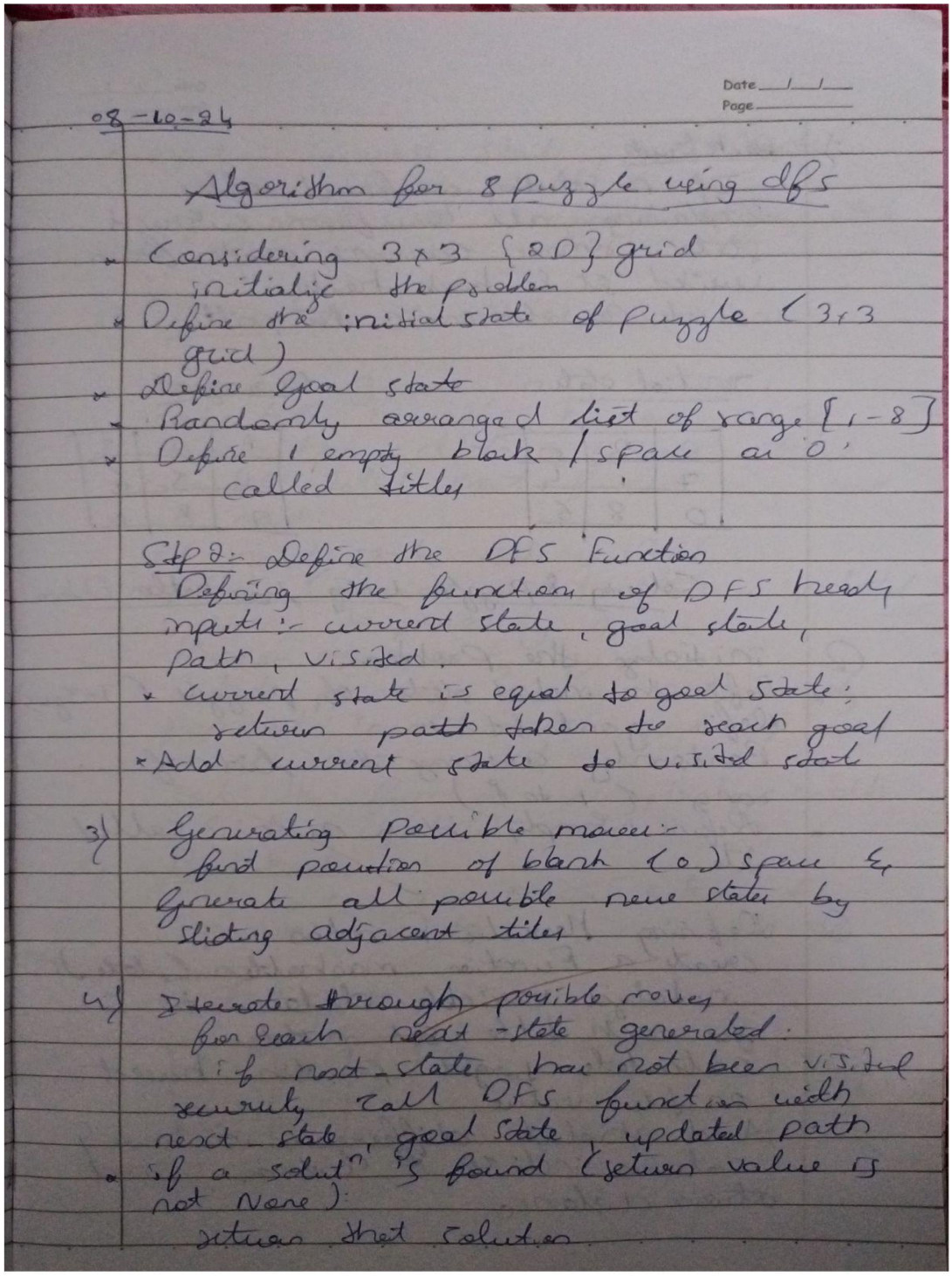
```
        l[i]=1
        print(f"Cleaned Room {i}")
    print(f"Moved to Room {(i+1)%2}")
    return (i+1)%2
i=random.choice([0,1])

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:



Date / /
Page

08-10-24

Algorithm for 8 Puzzle using dfs

- Considering 3x3 {2D} grid initialize the problem
- Define the initial state of puzzle (3x3 grid)
- Define goal state
- Randomly arranged list of range [1-8]
- Define 1 empty blank / space as '0' called tiles

Step 2:- Define the DFS Function
Defining the function of DFS heady
inputs:- current state, goal state, path, visited.
- current state is equal to goal state; return path (then we reach goal
- Add current state to visited state

3) Generating possible moves:-
find position of blank (0) space & 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 recursively call DFS function with next-state, goal state, updated path
- if a solution is found (return value is not None):
return that solution

Date / /
Page

5) Backtrack:
If no solution are found after exploring all born from current state, remove current state from visited set & backtrack indicate no solution in this path.

Initial state

2	3	5
7	1	4
0	8	6

Our goal

1	2	3
4	5	6
7	8	0

Solving 8 puzzle using Manhatten Distance

① Initialize the problem
- Define the initial state of puzzle (3×3 grid)
- Define goal state
- Randomly arrange the list of range (1 to 8)
- Define 1 Empty block as '0' called tile.

② Defining Heuristic Function:
Create a Function manhatten (state):
Initialize variable distance to 0
for each title set in state
Calculate target position based on tile value.
Add absolute difference in row & column indices to distance and return distance

Date / /
Page

* input :- Current state, goal state, g-cost visited, path
if current state is equal to goal state
Return path.

3) Generating Possible moves

* Calculate g-cost as g-cost + 1
Calculate heuristic h-cost using manhatten-distance (next-state)
calculate total cost f-cost = g-cost + h-cost.
if next state not visited
Add (f-cost, next-state, updated-state) to priority queue

4) Process of priority queue
while priority queue not empty
Pop state lowest total cost (f(n))
mark it as visited
call the function recursively with State & updated costs

5) Backtrack
if no solution occurs return to previous position

initial state

2	0	1
5	4	6
7	8	3

1	2	3
4	5	6
7	8	0

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}")
```

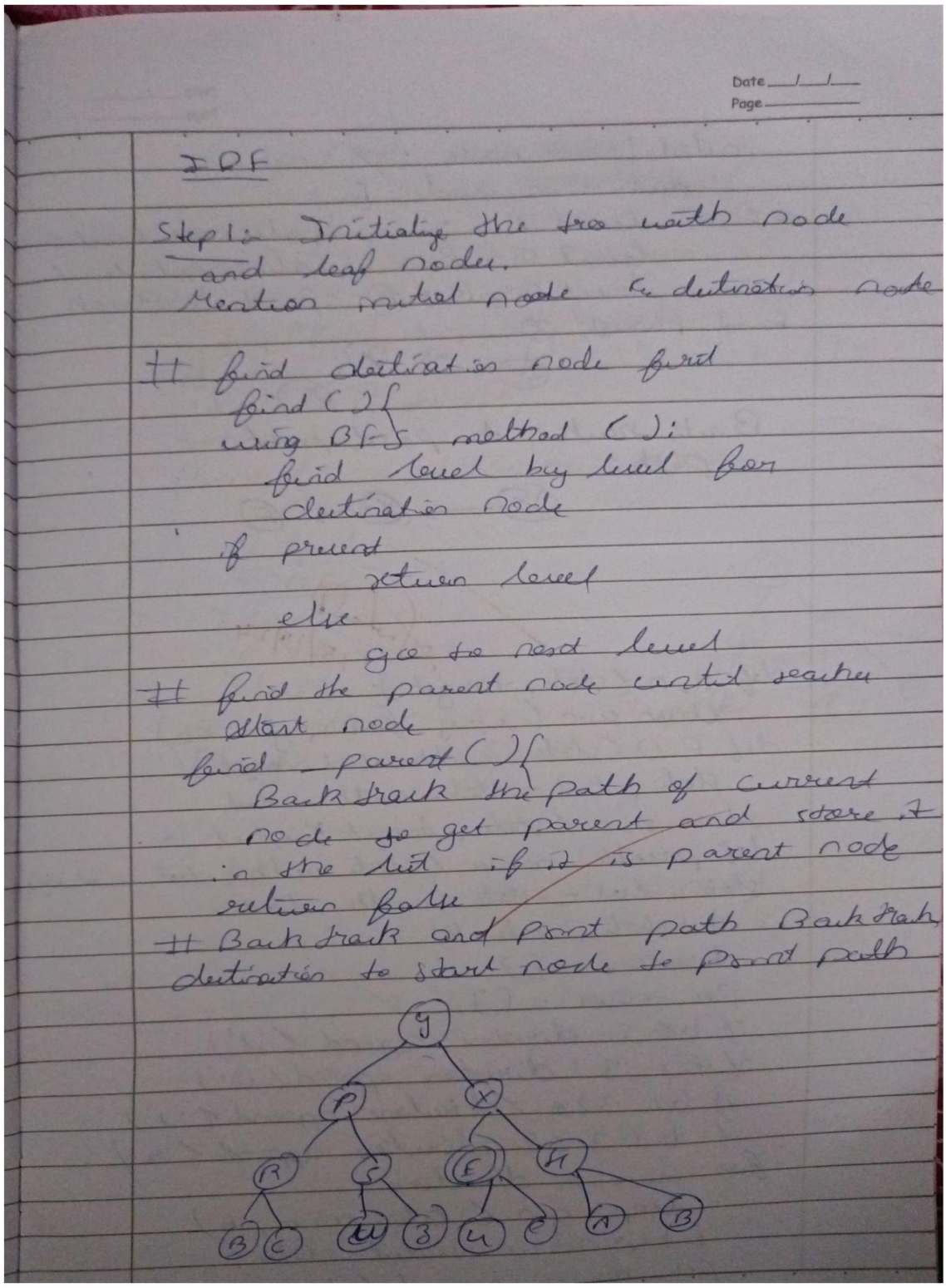
Output:

```
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:



Date __/__/__
Page

initial (start node : Y
destination node : E
BFS :- level 1 : Y False next level
level 2 : P, X False next level
level 3 : R S (E) Find E Break
Find Parent : (E) → (X)

Back track to print path
Path :-
(Y) → (X) → (E)

15/10/24

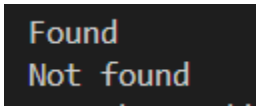
def H_n (state, target):
return sum(x != y for x, y in zip)
def F_D (state_with_hel, target):
state, hel = state_with_hel
return H_n(state, target) + hel
def possible_moves (state_with_hel, visited
state, hel = state_with_hel
b = state.index(0)
direction = []
per_moves = []
if b < 5: direction.append('d')
if b >= 3: direction.append('u')
if b % 3 > 0: direction.append('l')
if b % 3 < 2: direction.append('r')
for move in direction:
temp = gen(state, move, b)
if temp not in visited_states:
per_moves.append((temp, level+1))
return per_moves

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")
```

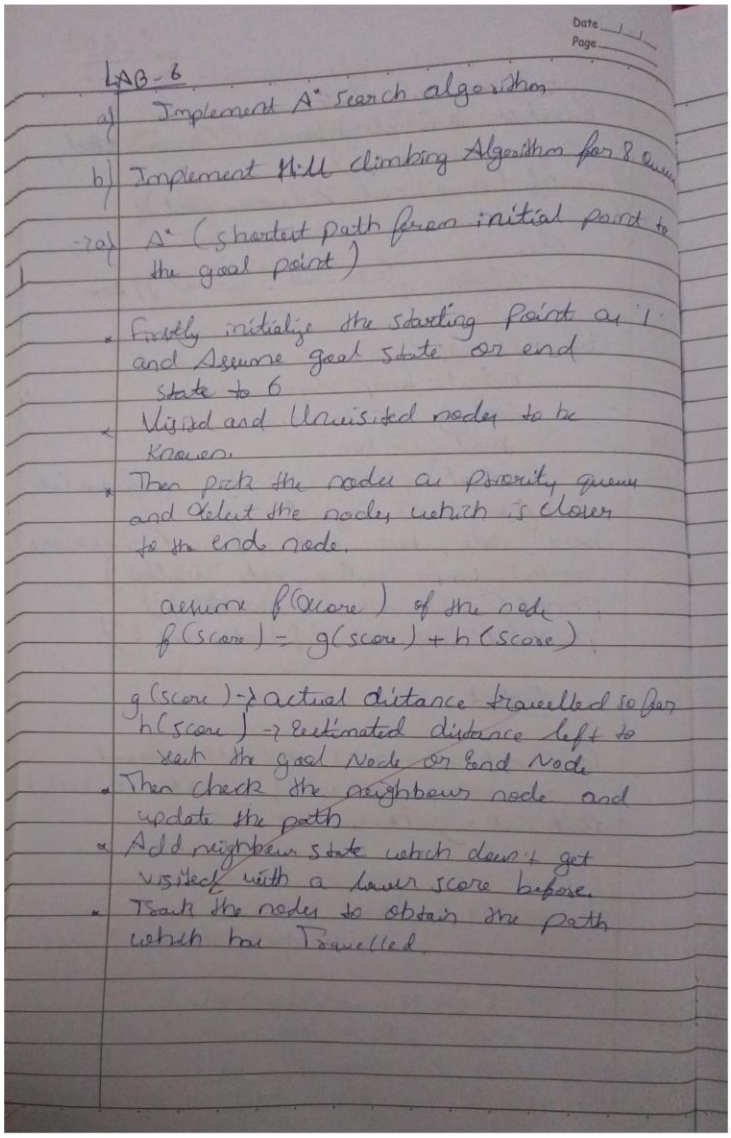
```
node5=TreeNode("F")
node6=TreeNode("H")
node7=TreeNode("B")
node8=TreeNode("C")
node9=TreeNode("S")

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:



LAB - 6

a) Implement A* search algorithm

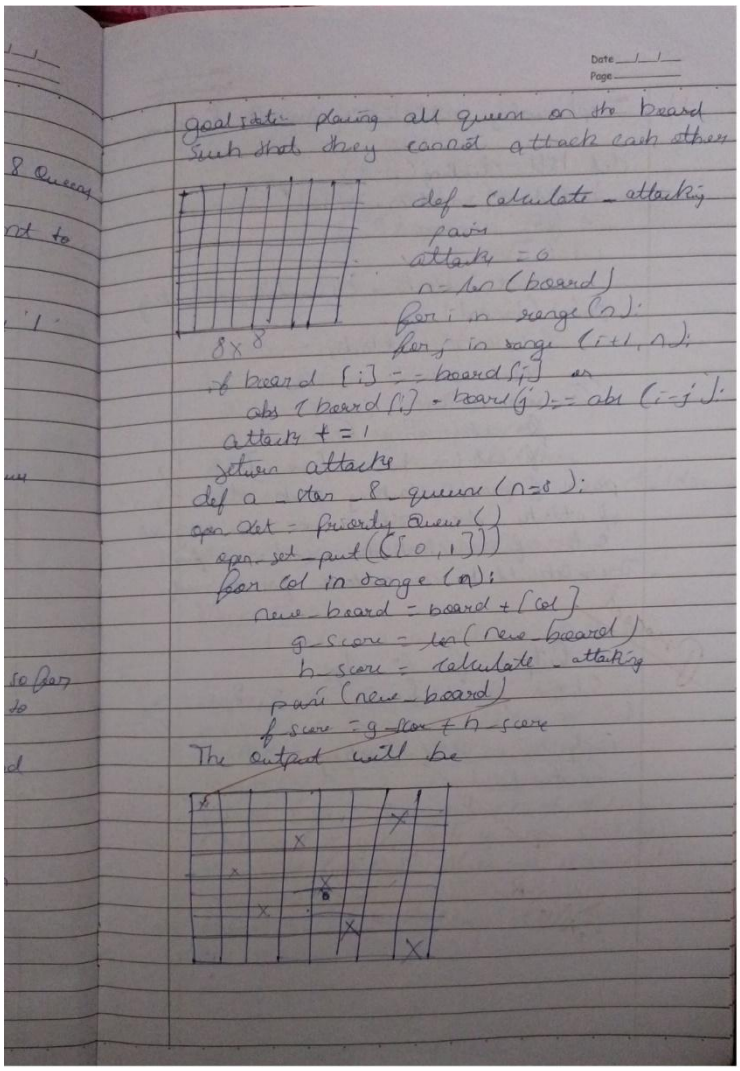
b) Implement Hill climbing Algorithm for 8 Queens

7a) A* (shortest path from initial point to the goal point)

- Firstly initialize the starting point as '1' and Assume goal state or end state to 6
- Visited and Unvisited nodes to be known.
- Then pick the nodes as priority queue and select the nodes which is closer to the end node.

  assume f(score) of the node
  f(score) = g(score) + h(score)

  g(score) -> actual distance travelled so far
  h(score) -> Estimated distance left to reach the goal Node or End Node
- Then check the neighbour node and update the path
- Add neighbour state which don't get visited with a lower score before.
- Trace the nodes to obtain the path which has Travelled.



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("+---+---+---+")
```

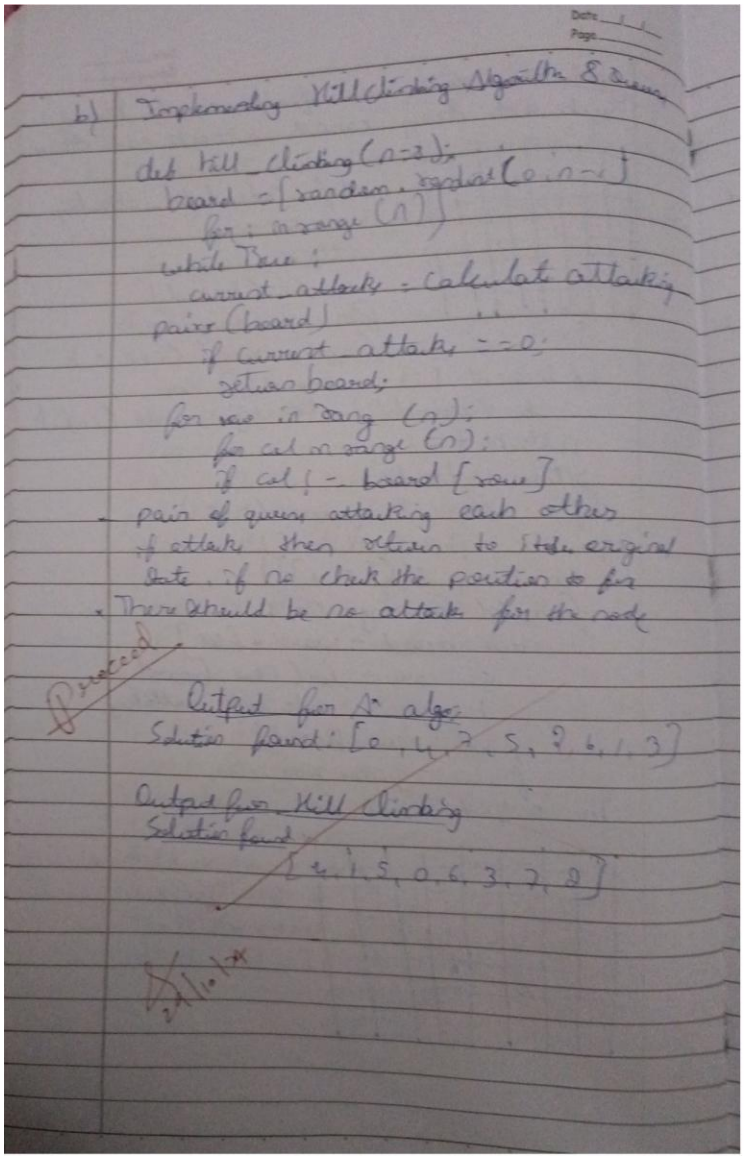
```
c =[[] for i in range(3)]
for i in range(3):
  print(f"Enter elements of row {i+1}")
  c[i]=list(map(int,input().split()))
a_star(c,0)

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:

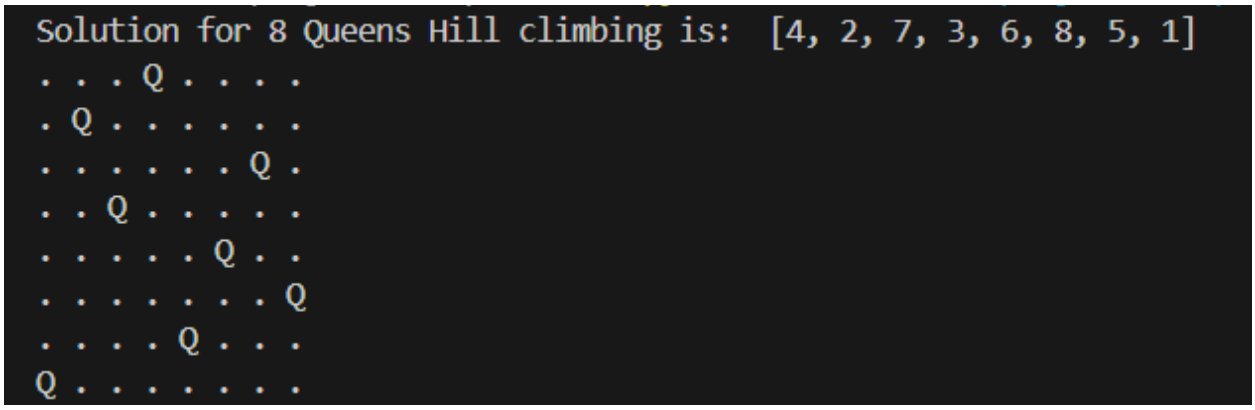


Code:

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())
```

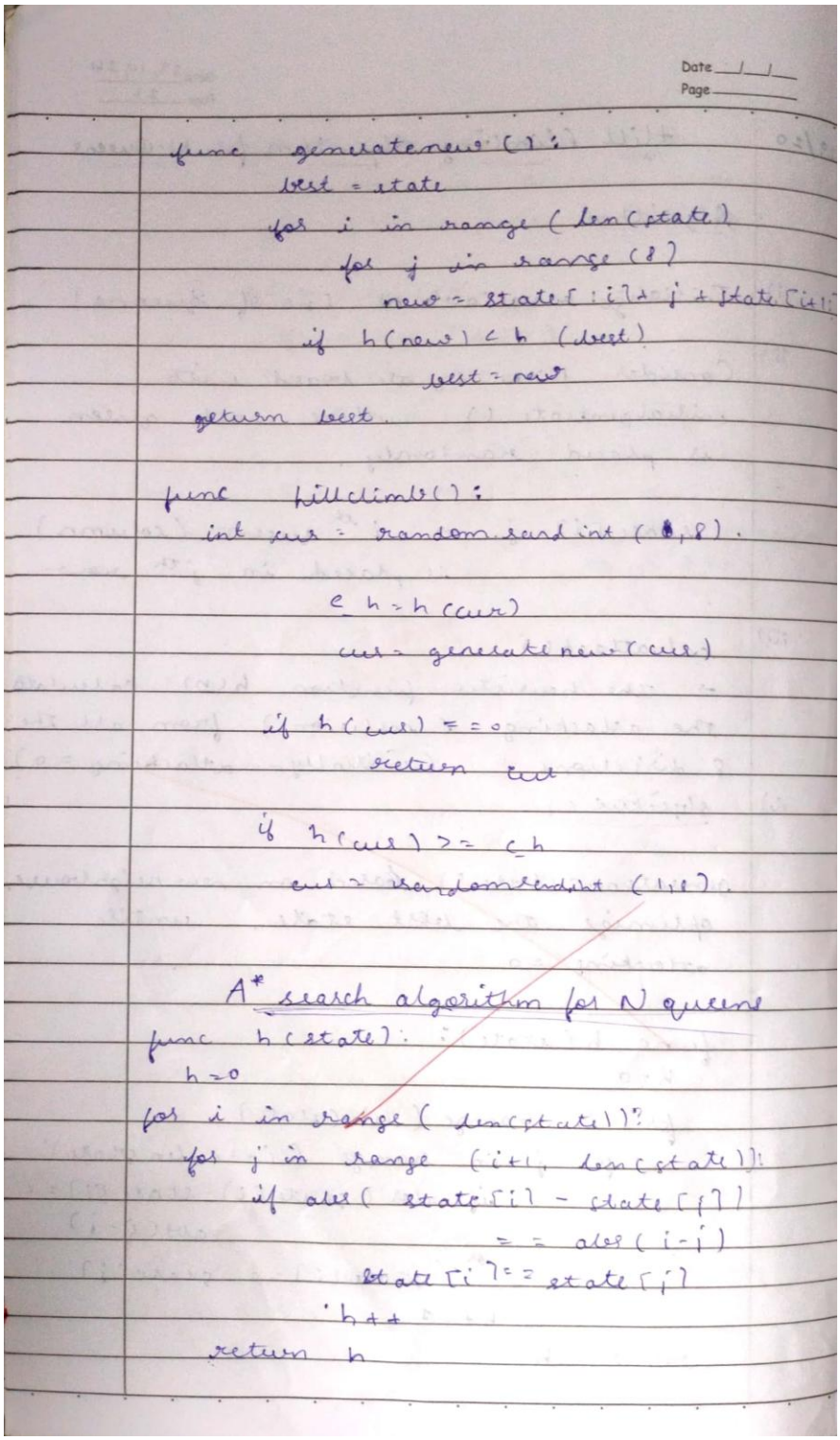
Output:



Implement A star algorithm to solve

N-Queens problem.

Algorithm:



```
Date __/__/__
Page

func generatenew():
    best = state
    for i in range(len(state)
        for j in range(8)
            new = state[:i]+j+state[i+1:]
        if h(new) < h(best)
            best = new
    return best

func hillclimb():
    int cur = random.randint(0,8).

    c_h = h(cur)
    cur = generatenew(cur)

    if h(cur) == 0
        return cur

    if h(cur) >= c_h
        cur = randomrandint(1,8).

A* search algorithm for N queens
func h(state):
    h = 0
for i in range(len(state)):
    for j in range(i+1, len(state)):
        if abs(state[i] - state[j])
            == abs(i-j)
            state[i] == state[j]
            h++
    return h
```

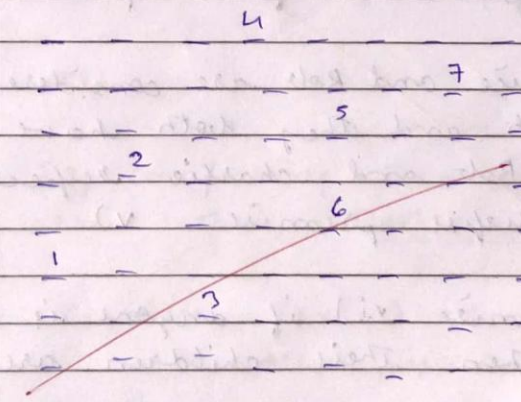
Date __/__/__
Page _____

```
func a star():
    initial []
    h = [], g = 8
    heap.push(h, (heuristic(initial) + g, white h: initial))
    c = heap.pop()
    if h(c[1]) + c[2] == 0:
        return c
    if len(c[1]) == 8:
        continue
    for i in range(1, 9):
        n = c[1] + [i]
        heap.push(q, h(n) + (c[2] - 1, n, g - 1))
```

proceed

Output:

[4 7 5 2 6 1 3 8]



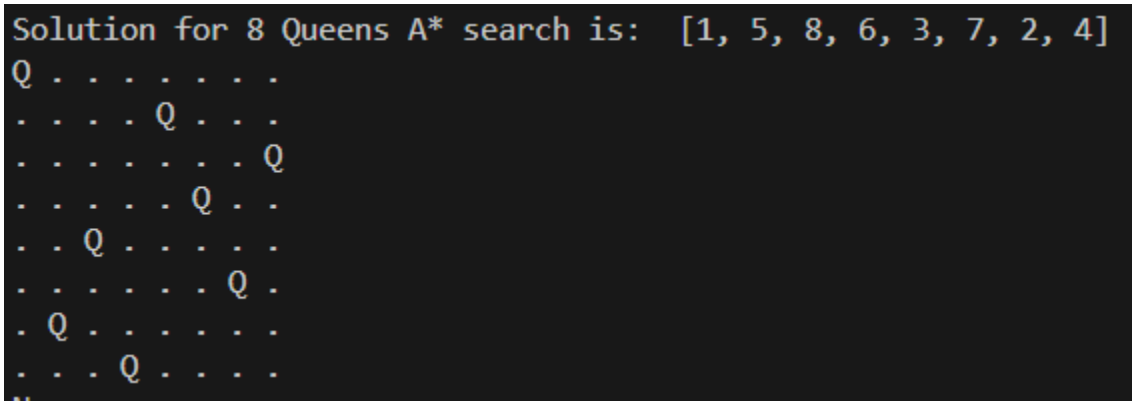
29/10/24

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:
```

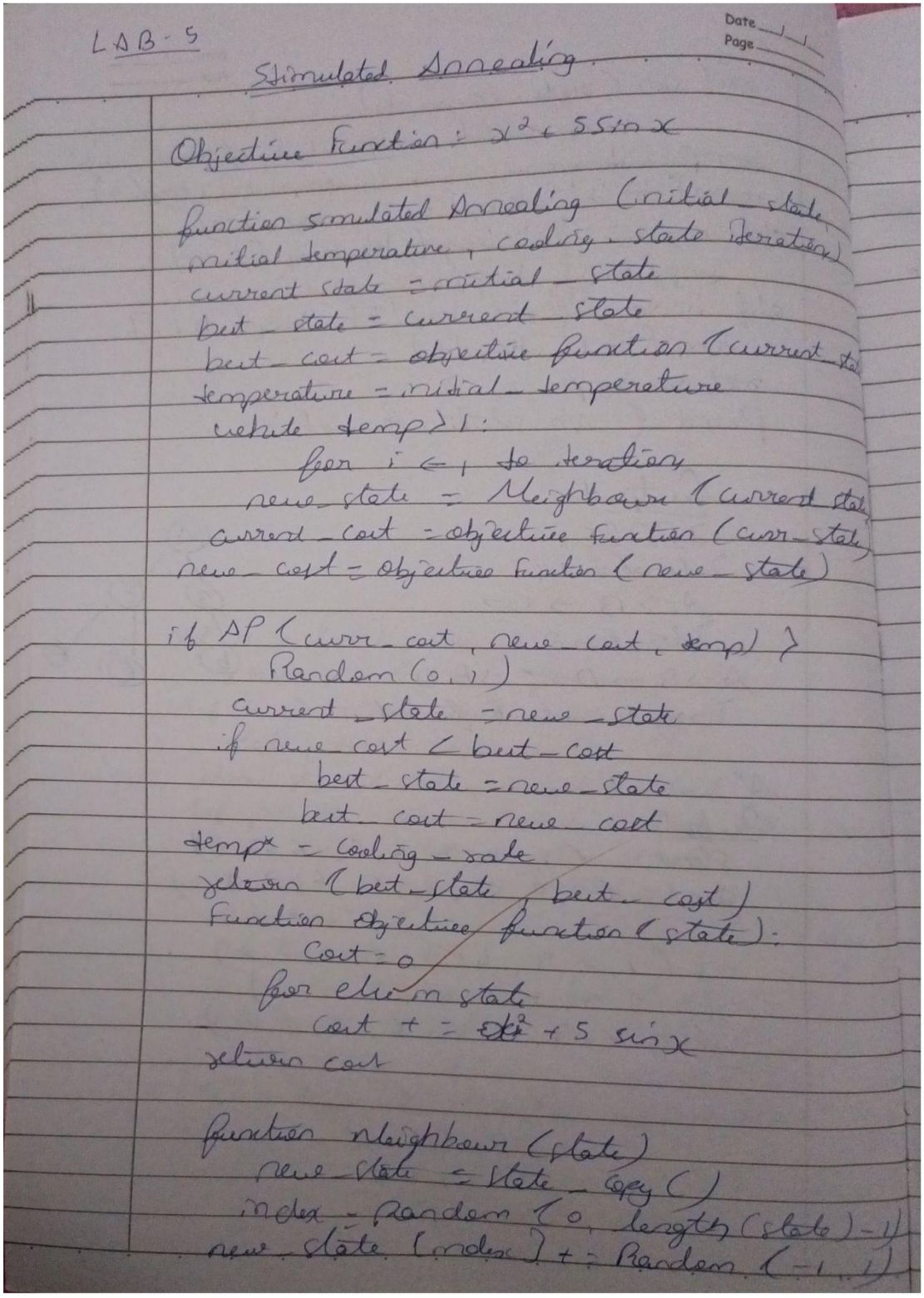
```
            new_state = state + [i]
            heapq.heappush(q, (h(new_state) + g, new_state, g - 1))

    return None

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

Output:



Implement Simulated Annealing:

Algorithm:



LAB-5

Simulated Annealing

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

```
function simulated Annealing (initial_state,
initial temperature, cooling_state iteration)
current state = initial_state
best_state = current_state
best_cost = objective function (current_sta
temperature = initial_temperature
while temp > 1:
    for i <- 1 to iteration
    new_state = Neighbour (current stat
    current_cost = objective function (curr_state)
new_cost = objective function (new_state)

if AP (curr_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 elem in state
        cost += x^2 + 5 sin x
return cost

function neighbour (state)
    new_state = state.copy()
    index = Random (0, length (state) - 1)
new_state [index] += Random (-1, 1)
```

```
    return new_state
function acceptance_probability(curr_state,
    new_cost, temp)
    if (new_cost < curr_cost):
        return 1
    else
        return e^((curr_cost - new_cost) / temp)

def main():
    initial_temp = 1000
    cooling_rate = 0.9
    iteration = 1000
    initial_state = [random.uniform(-10,
    10) for i in range(7)]
best_state, best_cost = simulated_annealing(initial_state,
initial_temp, cooling_rate, iteration)
print("Best state : {best_state}")
print("Best cost : {best_cost}")
```

Output

Best state = [-0.2587, -0.13911, -0.1005,
-0.0901, -0.2483, 0.0074, -0.1004]
Best cost = 0.17660

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:
```

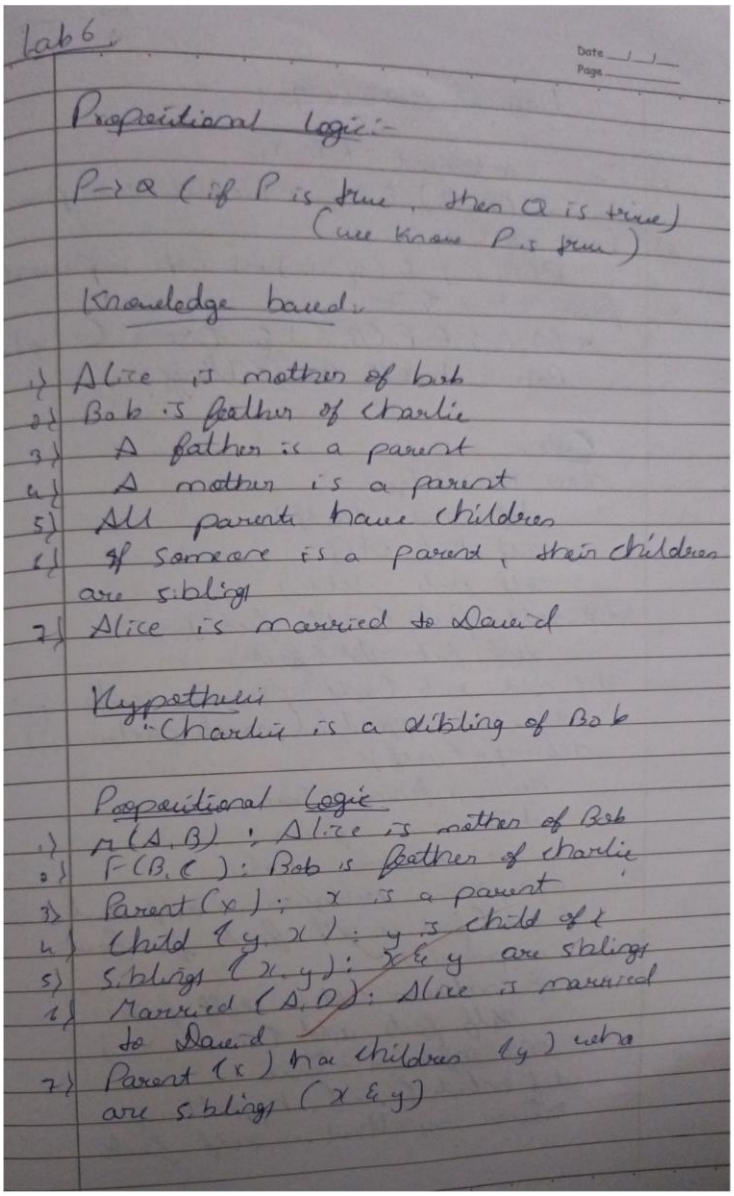
```
        c += i**2 + 2*i + 1
    return c

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:



Lab 6

Date / /
Page

Propositional logic:-

P → Q (if P is true, then Q is true)
(we know P is true)

Knowledge based.

1) Alice is mother of bob
2) Bob is father of Charlie
3) A father is a parent
4) A mother is a parent
5) All parents have children
6) If someone is a parent, their children are sibling
7) Alice is married to David

Hypothesis
: Charlie is a sibling of Bob

Propositional Logic

1) M(A, B) : Alice is mother of Bob
2) F(B, C) : Bob is father of Charlie
3) Parent(x) : x is a parent
4) Child(y, x) : y is child of x
5) Sibling(x, y) : x & y are siblings
6) Married(A, D) : Alice is married to David
7) Parent(x) ∧ a children(y) ∧ are sibling(x & y)

Date / /
Page

Logical reasoning

1) From statement 1 & 4
   M(A, B) & (y, x) -> Alice is parent
2) From 2 & 4:
   F(B, C) & (y, x) -> Bob is parent
3) from 1 & 2 & 7:
   M(A, B) & F(B, C) & (x & (x, y)) -> Bob & Charlie are not siblings

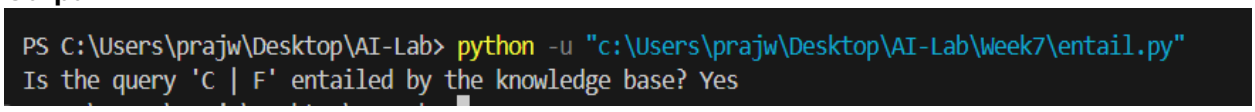
Code:-

```
class Knowledge:
    def __init__(self):
        self.rules = []
        self.facts = set()
    def add_fact(self, fact):
        self.facts.add(fact)
    def add_rule(self, premises, conclusion):
        self.rules.append((premises, conclusion))
    def infer(self):
        new_inferences = True
        while new_inferences:
            new_inferences = False
            for premises, conclusion in self.rules:
                if all(fact in self.facts for fact in
                    premises):
                    if conclusion not in self.facts:
                        self.facts.add(conclusion)
                        new_inferences = True
    def entails(self, hypothesis):
        return hypothesis in self.facts
```

Code:

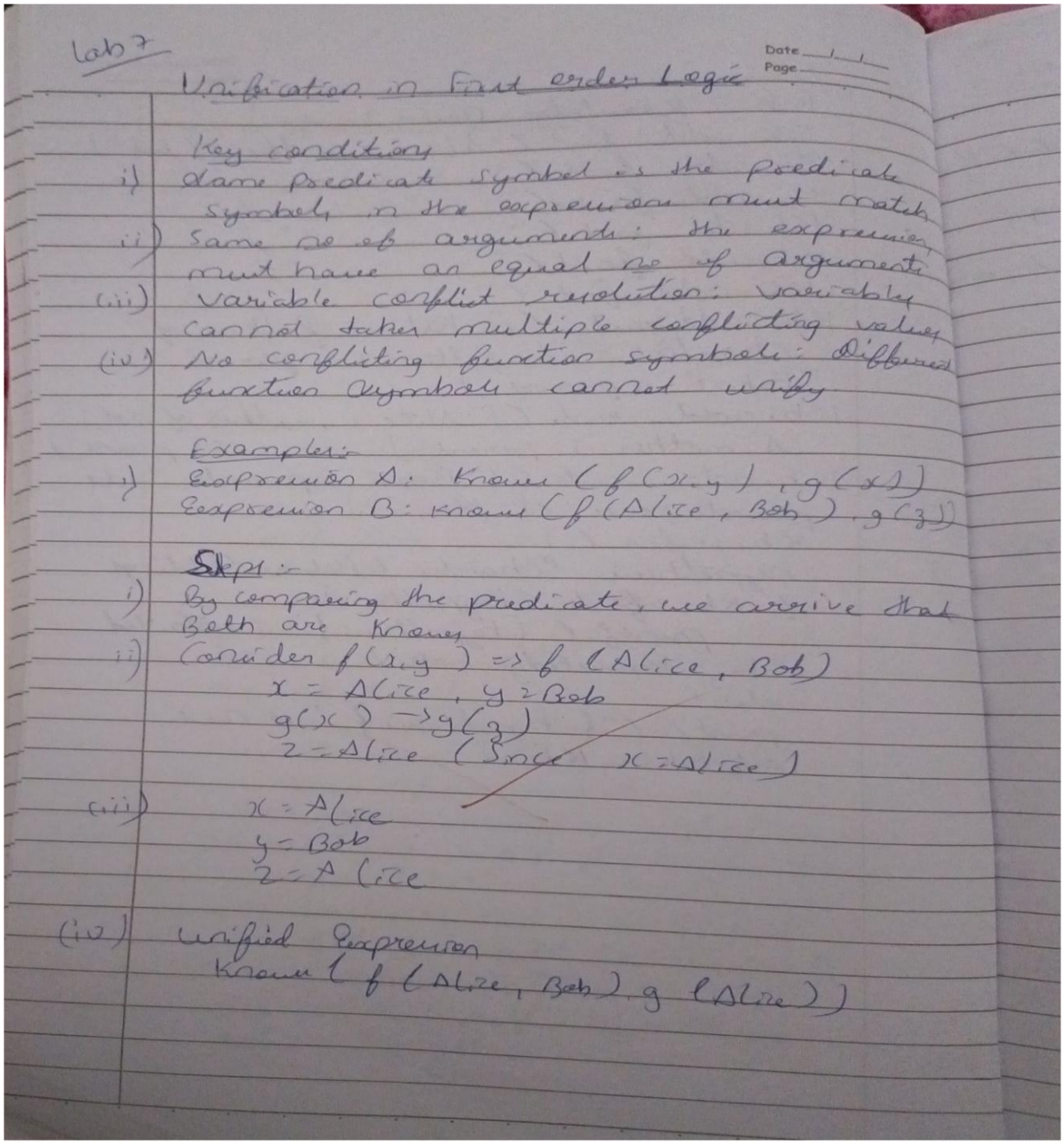
```
from sympy.logic.boolalg import Or, And, Not
from sympy.abc import A, B, C, D, E, F
from sympy import simplify_logic

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:



Implement unification in first order logic.

Algorithm:


Lab 7

Date / /
Page

Unification in First order Logic

Key conditions
i) Same predicate Symbol : the predicate symbols in the expression must match
ii) Same no of arguments : the expression must have an equal no of arguments
iii) Variable conflict resolution : variables cannot take multiple conflicting values
iv) No conflicting function symbols : Different function symbols cannot unify

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

Steps :
i) By comparing the predicate, 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))

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:
```

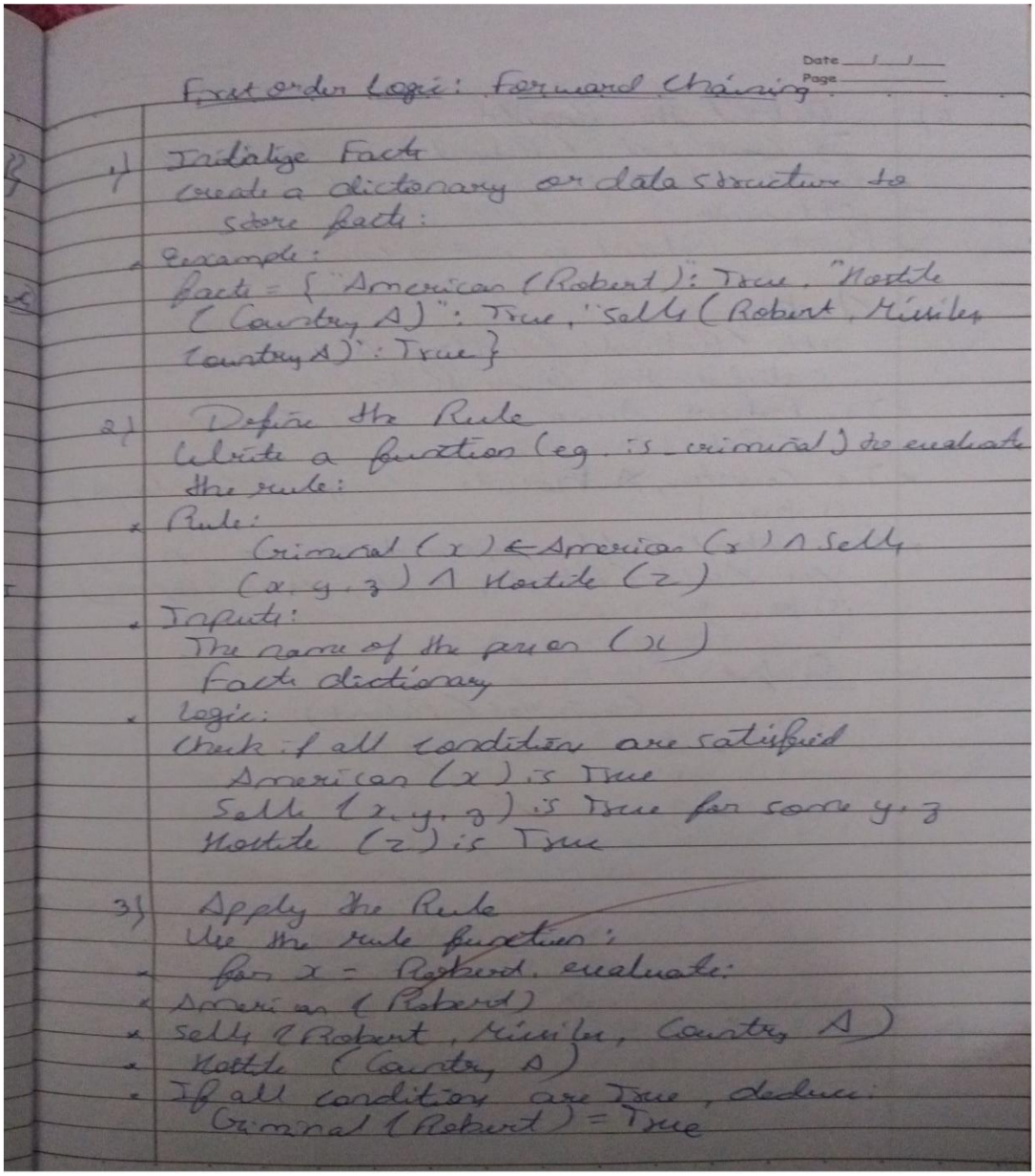
```
        return "FAILURE"

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:

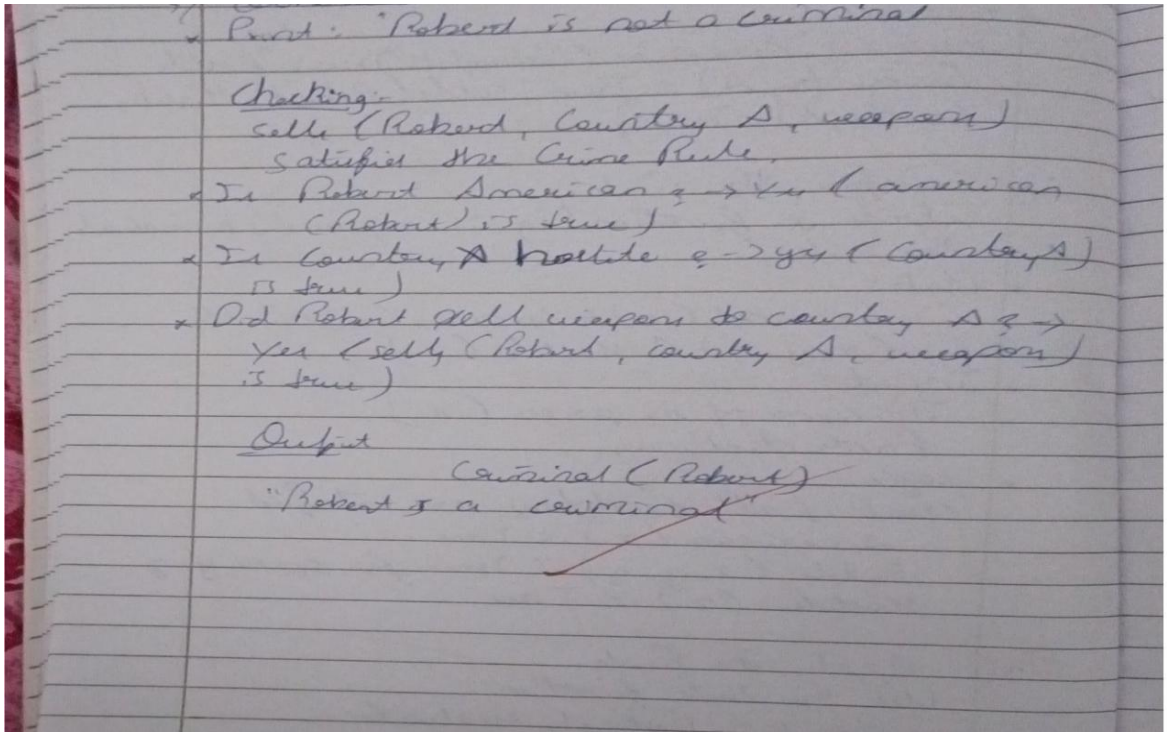


First order Logic: Forward Chaining

1) Initialize Facts
Create a dictionary or data structure to store facts:
- Example:
facts = {"American (Robert)": True, "Hostile (Country A)": True, "Sells (Robert, Missiles, Country A)": True}

2) Define the Rule
Write a function (eg. is_criminal) to evaluate the rule:
- Rule:
Criminal (x) ← American (x) ∧ Sells (x, y, z) ∧ Hostile (z)
- Inputs:
The 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

3) Apply the Rule
Use the rule function:
- for x = Robert, evaluate:
- American (Robert)
- Sells (Robert, Missiles, Country A)
- Hostile (Country A)
- If all conditions are True, deduce:
Criminal (Robert) = True

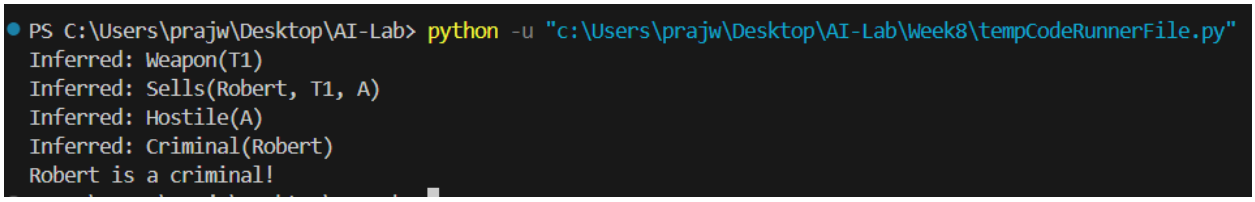


Code:

```
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)")
```

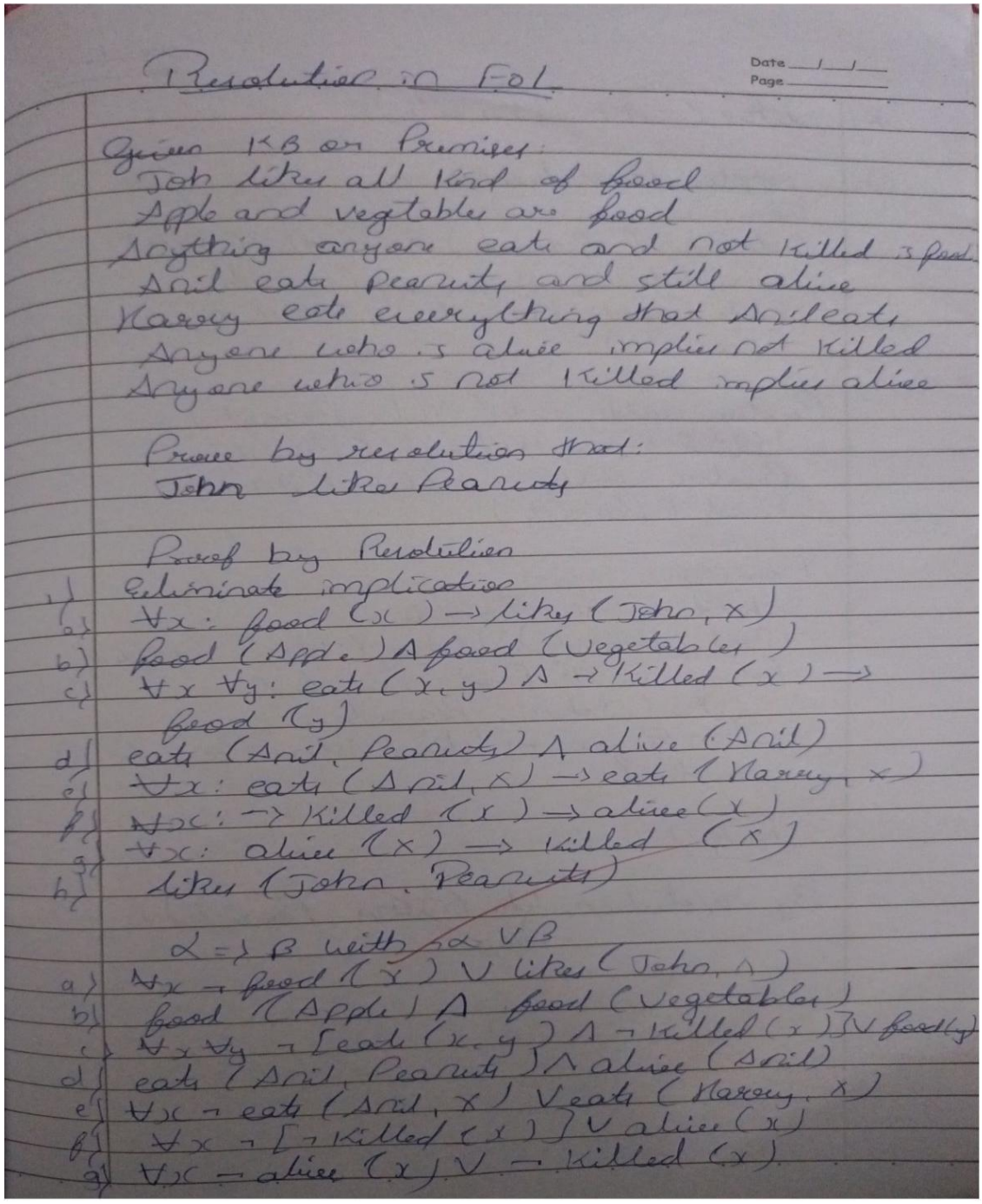
```
    if 'American(Robert)' in KB and 'Weapon(T1)' in KB and 'Sells(Robert, T1, A)' in KB and 'Hostile(A)' in KB:
        KB.add('Criminal(Robert)')
        print("Inferred: Criminal(Robert)")

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

forward_chaining()
```

Output:



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

Algorithm:



Date / /
Page

b) likes (John, Peanuts)

3) Apply Resolution
From (2):
Eats (Anil, Peanuts) ∧ ¬killed (Anil)
Substitute y = Anil, x = Peanuts:
¬Eats (Anil, Peanuts) V killed (Anil) V
(Peanuts)

Resolve with eat (Anil, Peanuts):
killed (Anil) V Food (Peanuts)
Resolve with ¬killed (Anil):
Food (Peanuts)

From (1):
Substitute x = Peanuts:
¬Food (Peanuts) V likes (John, Peanuts)
Resolve with Food (Peanuts):
likes (John, Peanuts)
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:
```

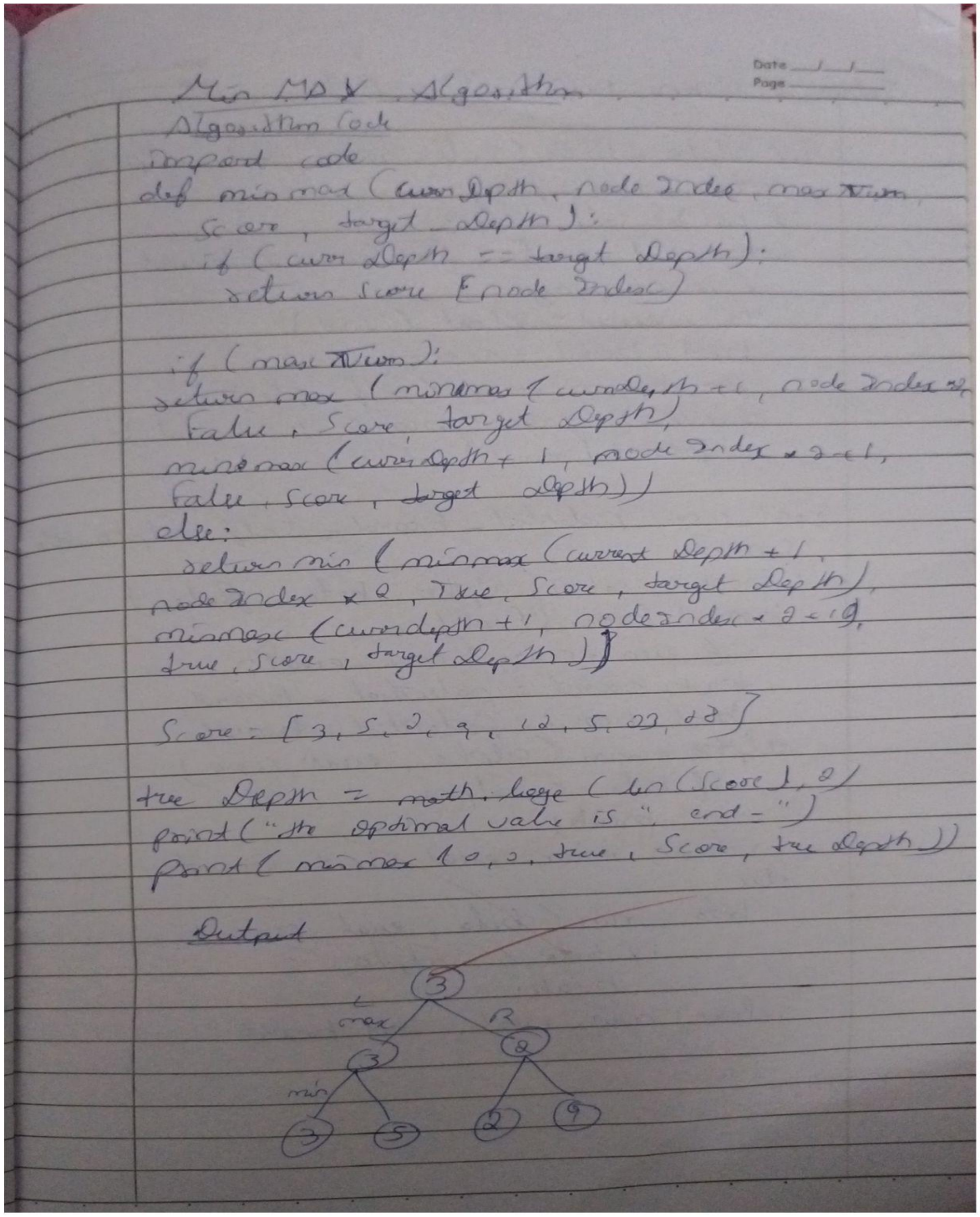
```
        func, args = predicate.split("(")
        args = args.strip(")").split(", ")
        # Handle philosopher and human link
        if func == "philosopher":
            return resolve(f"human({args[0]})")
        # Handle writesBooks rule explicitly
        if func == "writesBooks":
      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



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
```

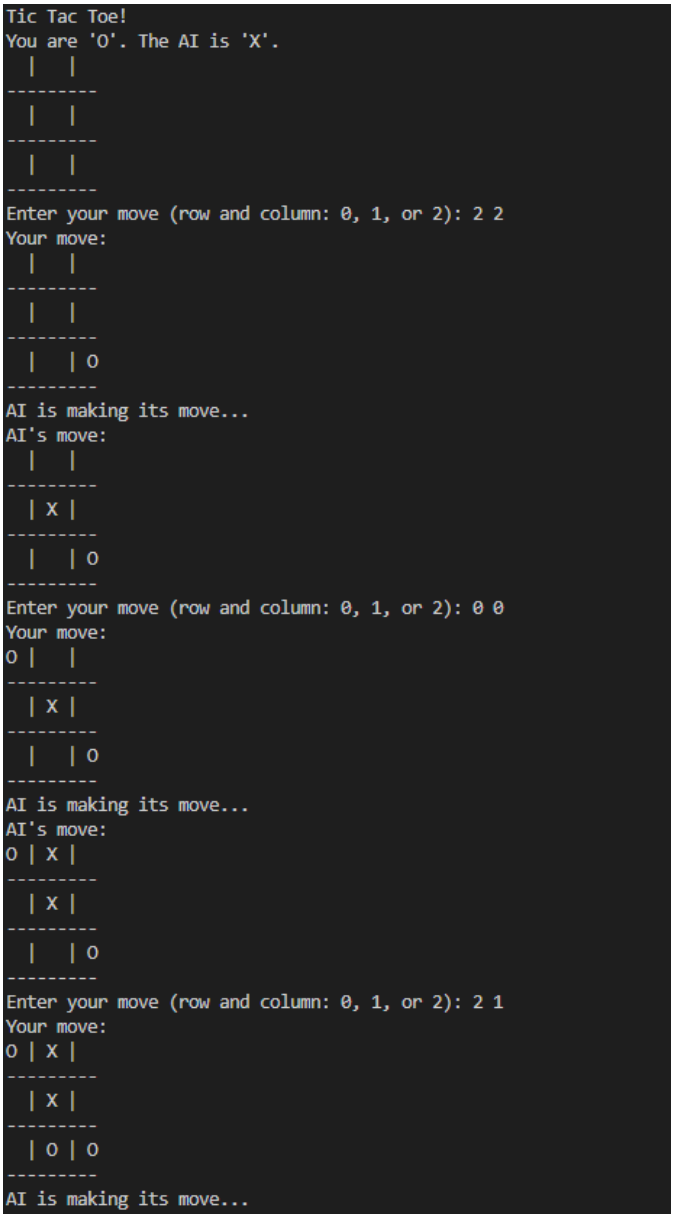
```
def minimax(board, depth, isMaximizing):
    score = evaluateBoard(board)
    if score == 10 or score == -10:
        return score
    if isDraw(board):
        return 0

    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
```

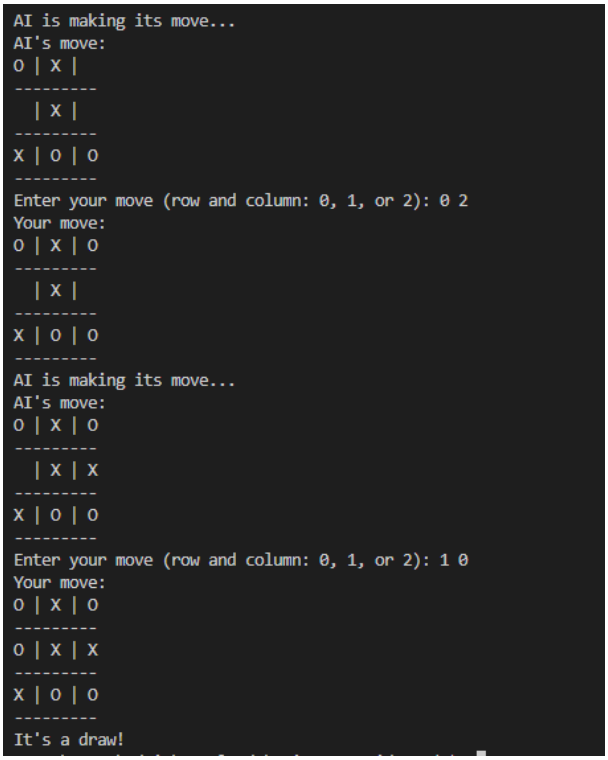
```
def findBestMove(board):
    bestValue = -math.inf
    bestMove = (-1, -1)
    for i in range(3):
        for j in range(3):
            if board[i][j] == "":
                board[i][j] = 'X'
                moveValue = minimax(board, 0, False)
                board[i][j] = ""
                if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

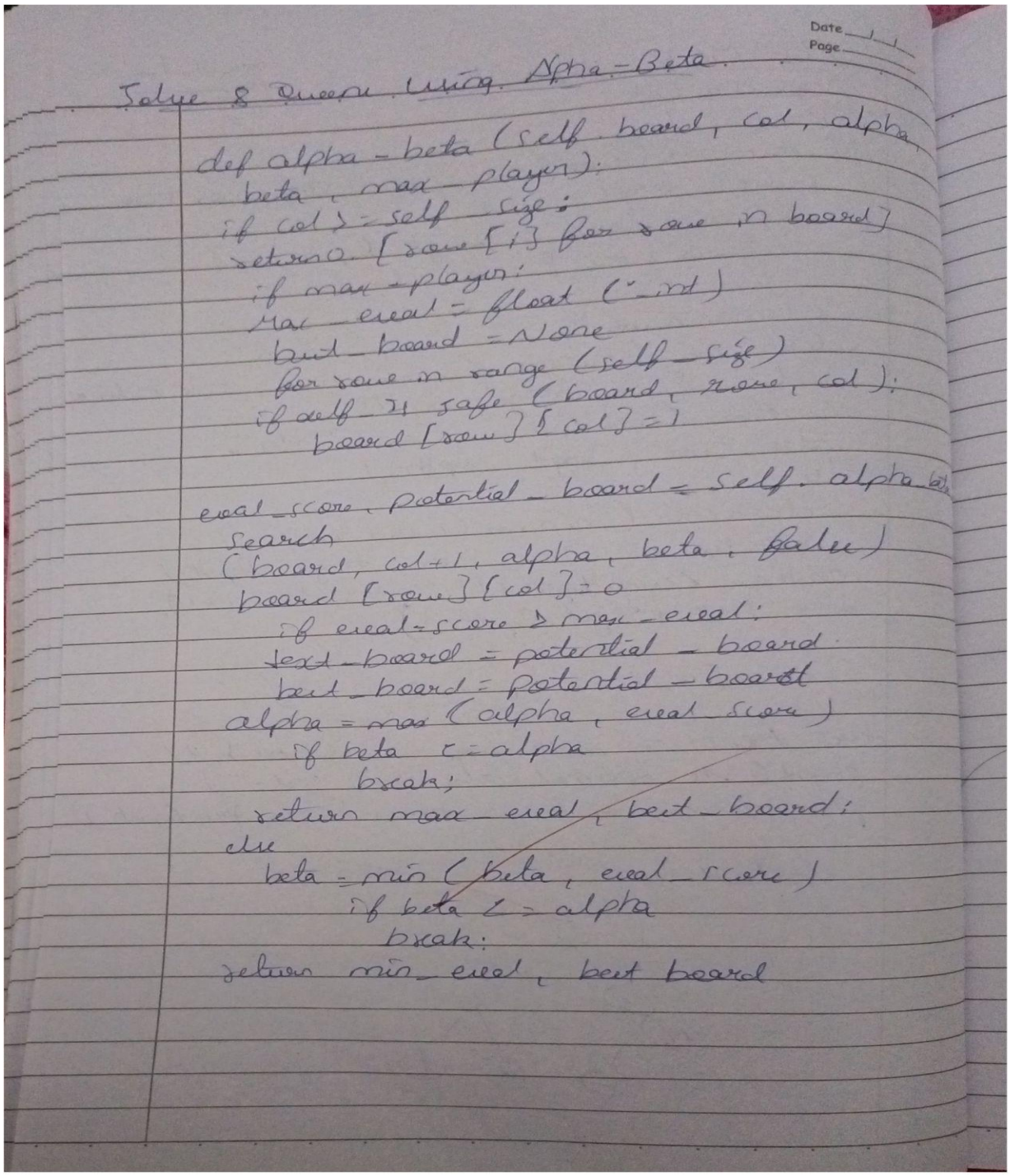
Output:





Implement Alpha-Beta Pruning for 8Queens.

Algorithm:



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: