VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Ranisha Giri (1BM22CS218)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by Ranisha Giri **(1BM22CS218),** 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.

Radhika A D. Assistant Professor Department of CSE, BMSCE	Dr. Jyothi S Nayak Professor & HOD Department of CSE, BMSCE

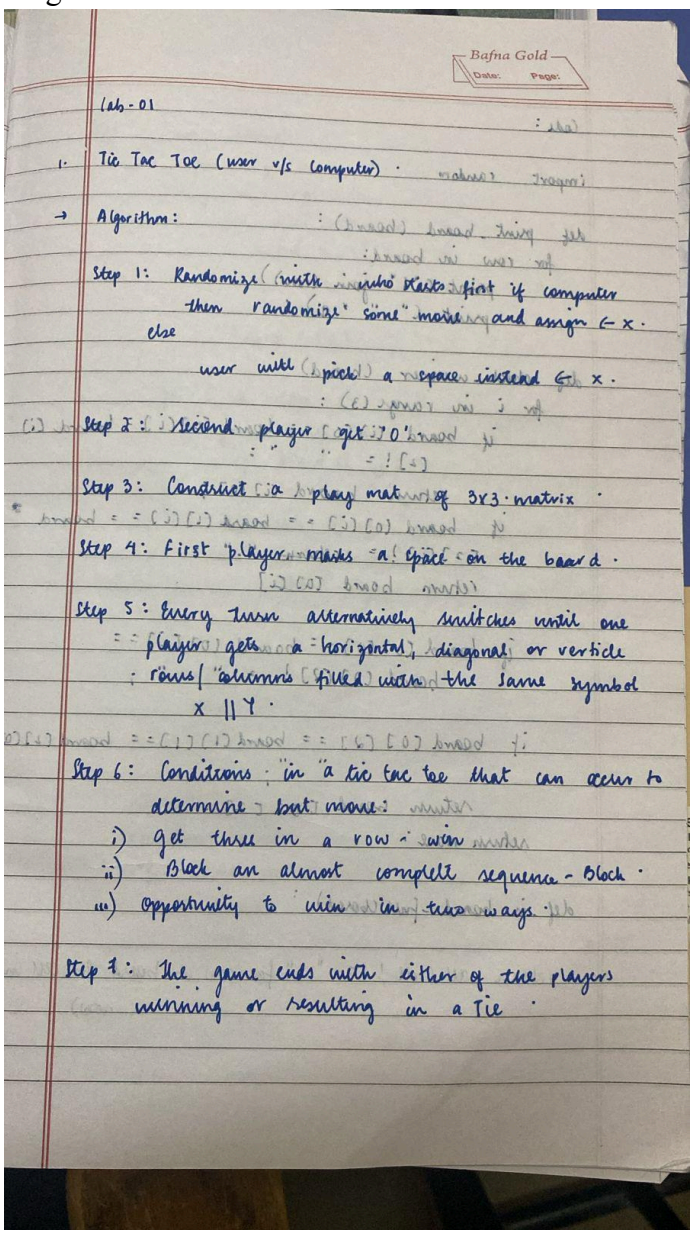
Index

Github Link: https://github.com/ranishagiri/AI-Lab-218

Program 1

Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent
Tic-Tac-Toe

Algorithm:


Bafna Gold
Date: Page:

Lab - 01

1. Tic Tac Toe (user v/s computer).

→ Algorithm:

Step 1: Randomize (with which starts first if computer then randomize some move and assign ← x.
else
user will pick a space instead ← x.

Step 2: Second player gets 'O'.

Step 3: Construct a play mat of 3x3 matrix.

Step 4: First player marks a space on the board.

Step 5: Every turn alternatively switches until one player gets a horizontal, diagonal or verticle rows/columns filled with the same symbol x || Y.

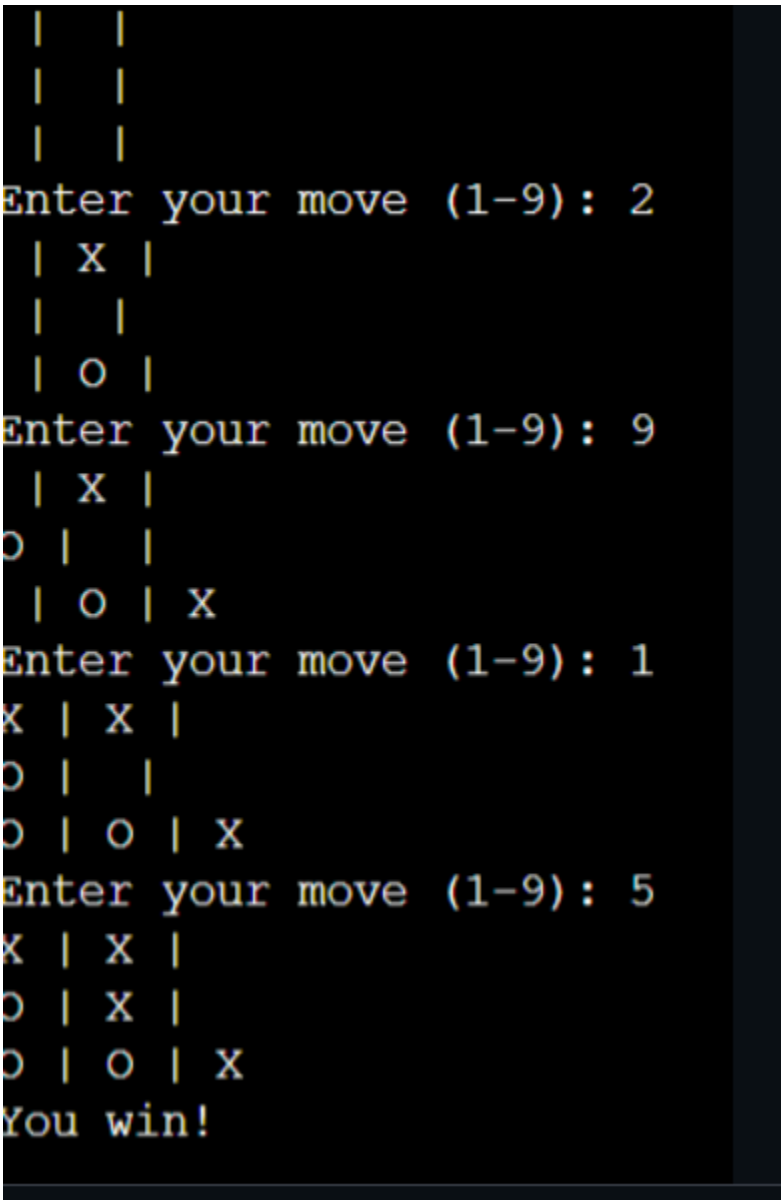
Step 6: Conditions in a tic tac toe that can occur to determine but more:
i) get three in a row - win
ii) Block an almost complete sequence - Block.
iii) Opportunity to win in two ways.

Step 7: The game ends with either of the players winning or resulting in a Tie.

Code:

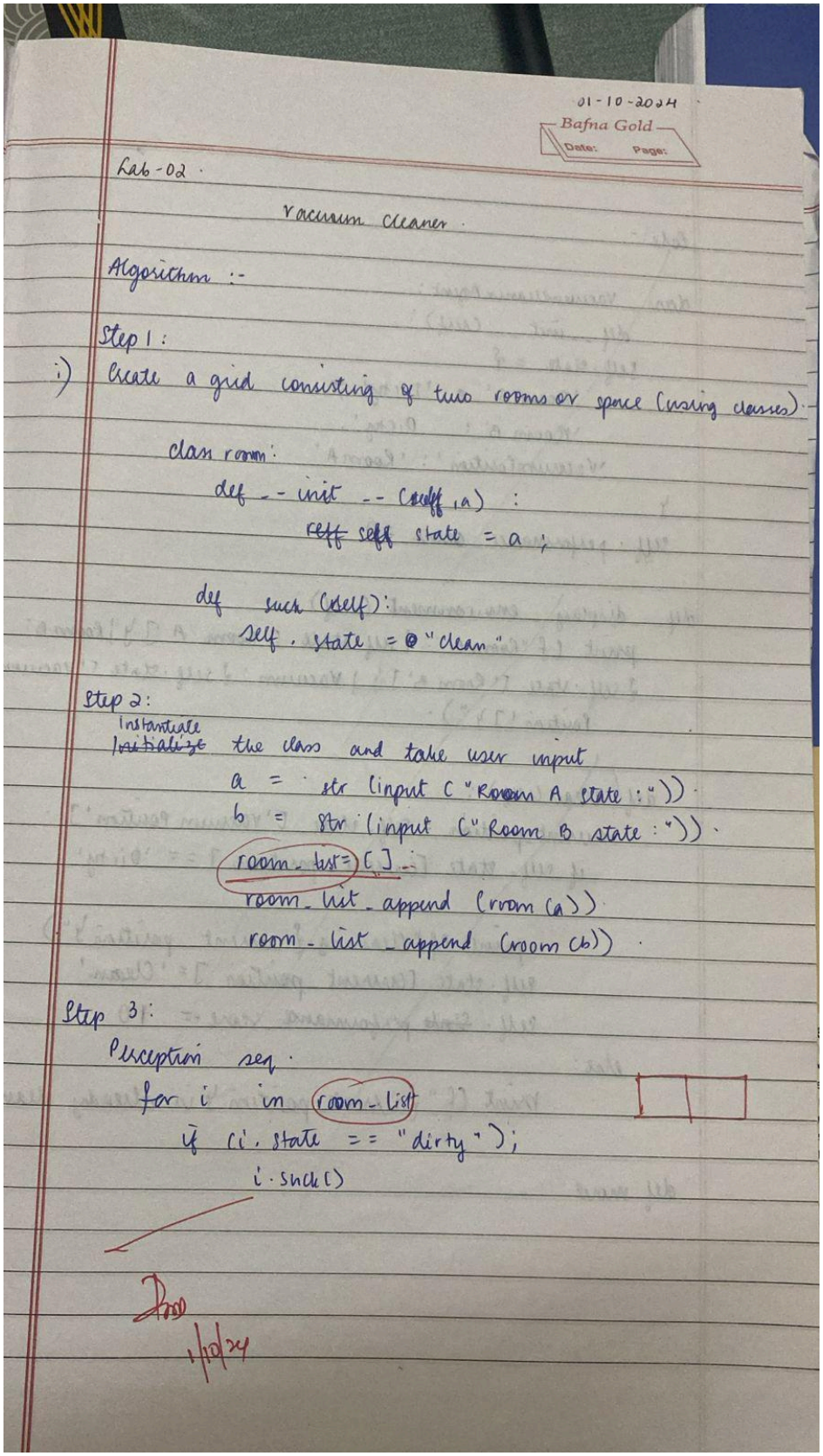
```
def check_win(board, r, c):
    if board[r - 1][c - 1] == 'X':
        ch = "O"
    else:
        ch = "X"
    if ch not in board[r - 1] and '-' not in board[r - 1]:
        return True
    elif ch not in (board[0][c - 1], board[1][c - 1], board[2][c - 1]) and '-' not in (board[0][c - 1],
board[1][c - 1], board[2][c - 1]):
        return True
    elif ch not in (board[0][0], board[1][1], board[2][2]) and '-' not in (board[0][0], board[1][1],
board[2][2]):
        return True
    elif ch not in (board[0][2], board[1][1], board[2][0]) and '-' not in (board[0][2], board[1][1],
board[2][0]):
        return True
    return False

def displayb(board):
  print(board[0])
  print(board[1])
  print(board[2])

board=[['-','-','-'],['-','-','-'],['-','-','-']]
displayb(board)
xo=1
flag=0
while '-' in board[0] or '-' in board[1] or '-' in board[2]:

  if xo==1:
    print("enter position to place X:")
    x=int(input())
    y=int(input())
    if(x>3 or y>3):
      print("invalid position")
      continue
    if(board[x-1][y-1]=='-'):
      board[x-1][y-1]='X'
      xo=0
      displayb(board)
    else:
      print("invalid position")
    continue
    if(check_win(board,x,y)):
```

```
        print("X wins")
        flag=1
        break
  else :
    print("enter position to place O:")
    x=int(input())
    y=int(input())
    if(x>3 or y>3):
      print("invalid position")
      continue
    if(board[x-1][y-1]=='-'):
      board[x-1][y-1]='O'
      xo=1
      displayb(board)
    else:
      print("invalid position")
    continue
    if(check_win(board,x,y)):
        print("0 wins")
        flag=1
        break
if flag==0:
  print("Draw")
print("Game Over")
```


| |
| |
| |
Enter your move (1-9): 2
| X |
| |
| O |
Enter your move (1-9): 9
| X |
O | |
| O | X
Enter your move (1-9): 1
X | X |
O | |
O | O | X
Enter your move (1-9): 5
X | X |
O | X |
O | O | X
You win!

Vacuum Cleaner

Algorithm:



Code:

```
count = 0
def rec(state, loc):
    global count
    if state['A'] == 0 and state['B'] == 0:
        print("Turning vacuum off")
        return

    if state[loc] == 1:
        state[loc] = 0
        count += 1
        print(f"Cleaned {loc}.")
        next_loc = 'B' if loc == 'A' else 'A'
        state[loc] = int(input(f"Is {loc} clean now? (0 if clean, 1 if dirty): "))
        if(state[next_loc]!=1):
          state[next_loc]=int(input(f"Is {next_loc} dirty? (0 if clean, 1 if dirty): "))
    if(state[loc]==1)
```

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

[0, 0]
[1, 0]

[0, 0]
[1, 0]

[0, 0]
[1, 0]

[0, 0]
[0, 0]

All rooms are cleaned!
```

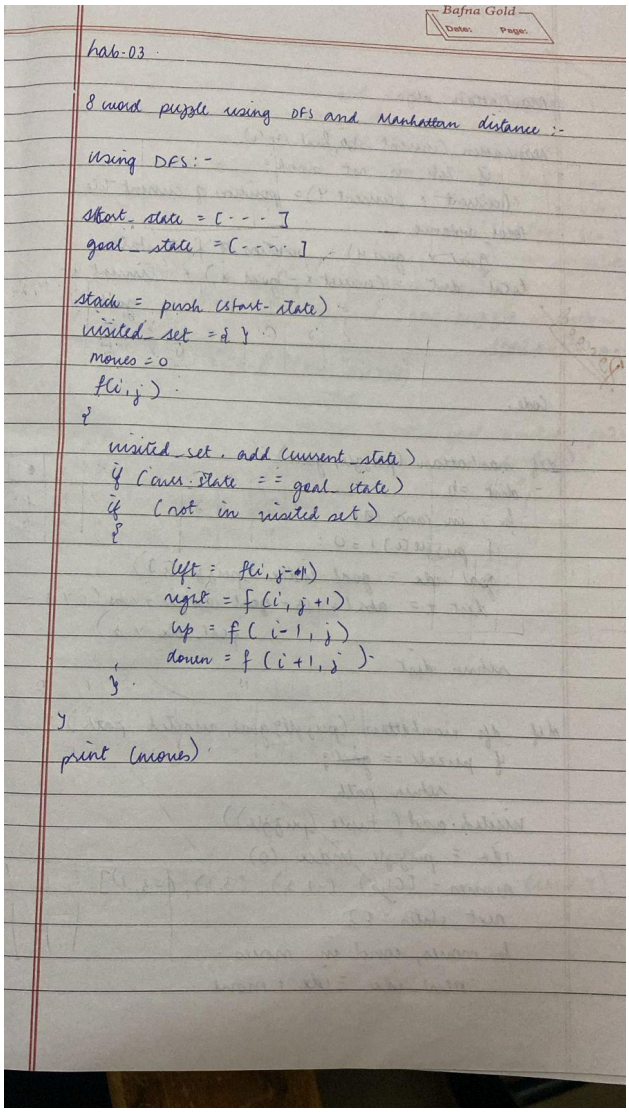
```
rec(state,loc) else:
        next_loc = 'B' if loc == 'A' else 'A'
        dire="left" if loc=="B" else "right"
        print(loc,"is clean")
        print(f"Moving vacuum {dire}")
        if state[next_loc] == 1:
            rec(state, next_loc)

  state = {}
  state['A'] = int(input("Enter state of A (0 for clean, 1 for dirty): "))
  state['B'] = int(input("Enter state of B (0 for clean, 1 for dirty): "))
  loc = input("Enter location (A or B): ")
  rec(state, loc)
  print("Cost:",count)
  print(state)
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm

8 puzzle using DFS

Algorithm:



Code:

```
import heapq

class Puzzle:
    def __init__(self):
        self.board = [
        [1, 2, 3],
        [8, 0, 4],
        [7, 6, 5]
    ]
        self.end = [
        [2, 8, 1],
        [0, 4, 3],
        [7, 6, 5]
    ]
    def getMoves(self, board):
        zero_pos = self.zero_index(board)
        moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        valid_moves = []
        for move in moves:
            if 0 <= zero_pos[0] + move[0] < 3 and 0 <= zero_pos[1] + move[1] < 3:
                valid_moves.append(move)
        return valid_moves

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

    def bhash(self, board):
      return tuple(map(tuple, board))

    def display(self, board):
        for ls in board:
            print(*ls)
```

```
    def manhattan_distance(self, state):
        """Calculate the total Manhattan distance of the state."""
        distance = 0
        for i in range(9):
            old = self.get_index(i, state)
            final = self.get_index(i, self.end)
            distance += (abs(final[0] - old[0]) + abs(final[1] - old[1]))
        return distance

    def get_index(self, el, board):
        for i in range(3):
            for j in range(3):
                if board[i][j] == el:
                    return [i, j]

    def misplaced(self, state):
        misplaced = 0
        for i in range(3):
            for j in range(3):
                if state[i][j] != self.end[i][j]:
                    misplaced += 1
        return misplaced



    def a_star(self):
        heap = []
        heapq.heappush(heap, (self.manhattan_distance(self.board) + self.misplaced(self.board), 0,
self.board, []))  # (priority, cost, current state, path)
        visited = set()  # Track visited states

        while heap:
            priority, cost, state, path = heapq.heappop(heap)

            # Convert the state to a tuple to store in a set (hashable)
            state_tuple = tuple(map(tuple, state))
```

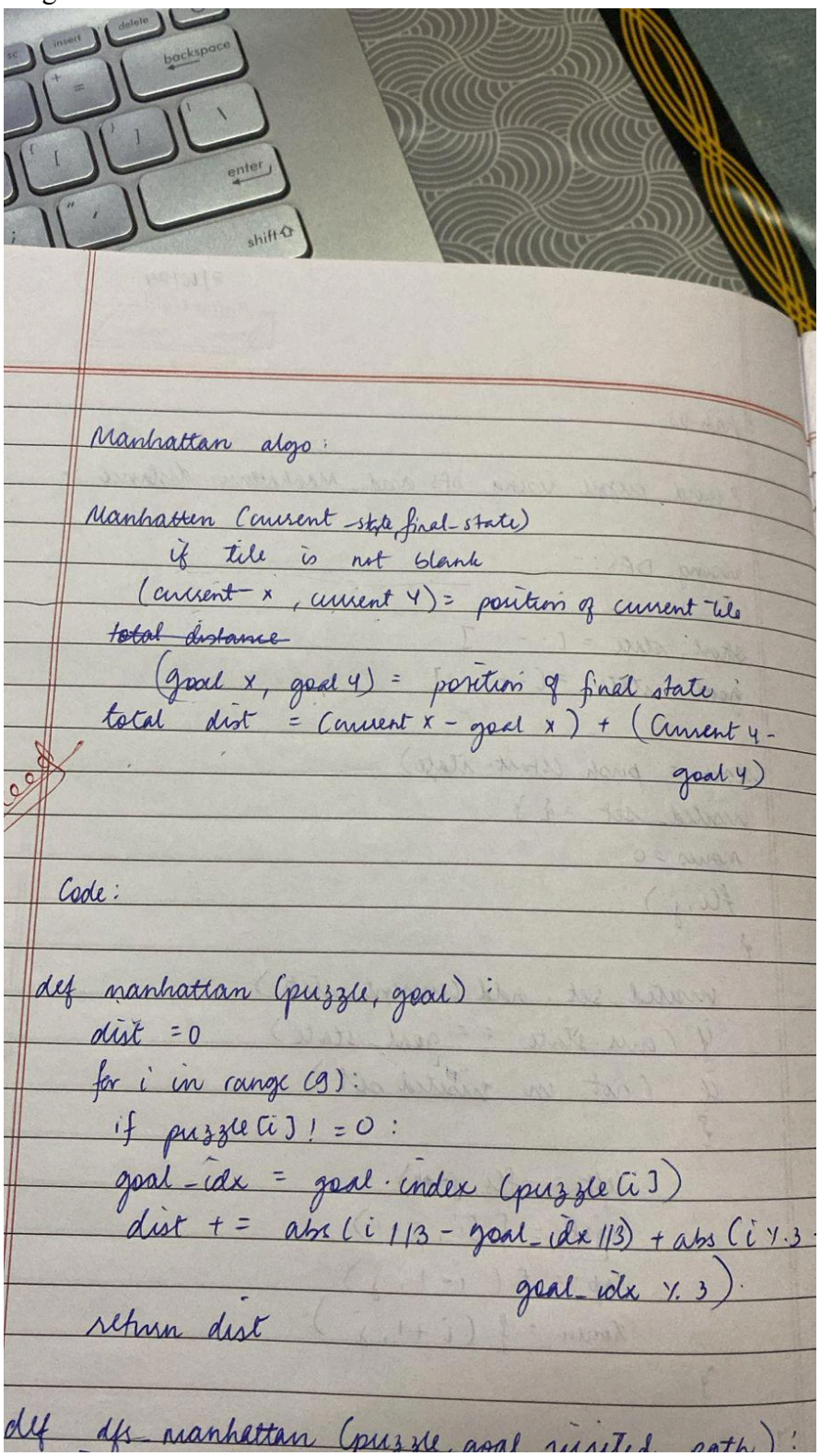
```
            if state_tuple in visited:
                continue

            visited.add(state_tuple)

            # If the current state is the goal state, return the path
            if self.bhash(state) == self.bhash(self.end):
                for p in path + [state]:
                    self.display(p)
                    print("------")
                return

            # Get all possible moves (neighbors) and add them to the heap
            for move in self.getMoves(state):
                new_board = [row[:] for row in state]
                zeroPos = self.zero_index(new_board)
                newPos = [zeroPos[0] + move[0], zeroPos[1] + move[1]]
                new_board[newPos[0]][newPos[1]], new_board[zeroPos[0]][zeroPos[1]] =
new_board[zeroPos[0]][zeroPos[1]], new_board[newPos[0]][newPos[1]]
                if tuple(map(tuple, new_board)) not in visited:
                    new_cost = cost + 1 # Each move has a cost of 1
                    priority = self.manhattan_distance(new_board) + self.misplaced(new_board)
                    heapq.heappush(heap, (priority, new_cost, new_board, path + [state]))



    def dfs(self):
        stack = []
        visited = []
        stack.append(self.board)
        visited.append(self.bhash(self.board))
        while stack:
            top = stack[-1]
            if self.bhash(top) == self.bhash(self.end):
                break
            valid_moves = self.getMoves(top)
```

```
            added = False
            # print(zeroPos, valid_moves)
            for move in valid_moves:
                new_board = [row[:] for row in top]
                zeroPos = self.zero_index(new_board)
                newPos = [zeroPos[0] + move[0], zeroPos[1] + move[1]]
                new_board[newPos[0]][newPos[1]], new_board[zeroPos[0]][zeroPos[1]] =
new_board[zeroPos[0]][zeroPos[1]], new_board[newPos[0]][newPos[1]]
                if self.bhash(new_board) not in visited:
                    stack.append(new_board)
                    visited.append(self.bhash(new_board))
                    added = True
                    break
            if not added:
                stack.pop()
        while stack:
            self.display(stack.pop(0))
            print("--------")


c = Puzzle()
print('DFS: ')
c.dfs()
print("MD: ")
c.a_star()
# print(c.zero_index("123405678"))
```

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

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

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

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

8 puzzle using Manhattan distance

Algorithm:



Manhattan algo:

Manhatten (current_state, final_state)
    if tile is not blank
    (current_x , current_y) = position of current tile
    ~~total distance~~
    (goal_x, goal_y) = position of final state
    total dist = (current_x - goal_x) + (current_y - goal_y)

Code:

```
def manhattan (puzzle, goal):
    dist = 0
    for i in range (9):
      if puzzle[i] != 0:
      goal_idx = goal.index (puzzle[i])
       dist += abs (i //3 - goal_idx //3) + abs (i %3 -
                                    goal_idx %3).
    return dist

def dfs_manhattan (puzzle, goal, visited, path):
```

Code:

```
from collections import deque

class PuzzleState:
    def __init__(self, board, zero_pos, moves=0, previous=None):
        self.board = board
        self.zero_pos = zero_pos # Position of the zero tile
        self.moves = moves          # Number of moves taken to reach this state
        self.previous = previous    # For tracking the path

    def is_goal(self, goal_state):
        return self.board == goal_state

    def get_possible_moves(self):
        moves = []
        x, y = self.zero_pos
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)] # Up, Down, Left, Right
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = [row[:] for row in self.board]
                # Swap the zero tile with the adjacent tile
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
                moves.append((new_board, (new_x, new_y)))
        return moves

def ids(initial_state, goal_state, max_depth):
    for depth in range(max_depth):
        visited = set()
        result = dls(initial_state, goal_state, depth, visited)
        if result:
            return result
    return None

def dls(state, goal_state, depth, visited):
    if state.is_goal(goal_state):
        return state
    if depth == 0:
        return None

    visited.add(tuple(map(tuple, state.board))) # Mark this state as visited
    for new_board, new_zero_pos in state.get_possible_moves():
        new_state = PuzzleState(new_board, new_zero_pos, state.moves + 1, state)
        if tuple(map(tuple, new_board)) not in visited:
```

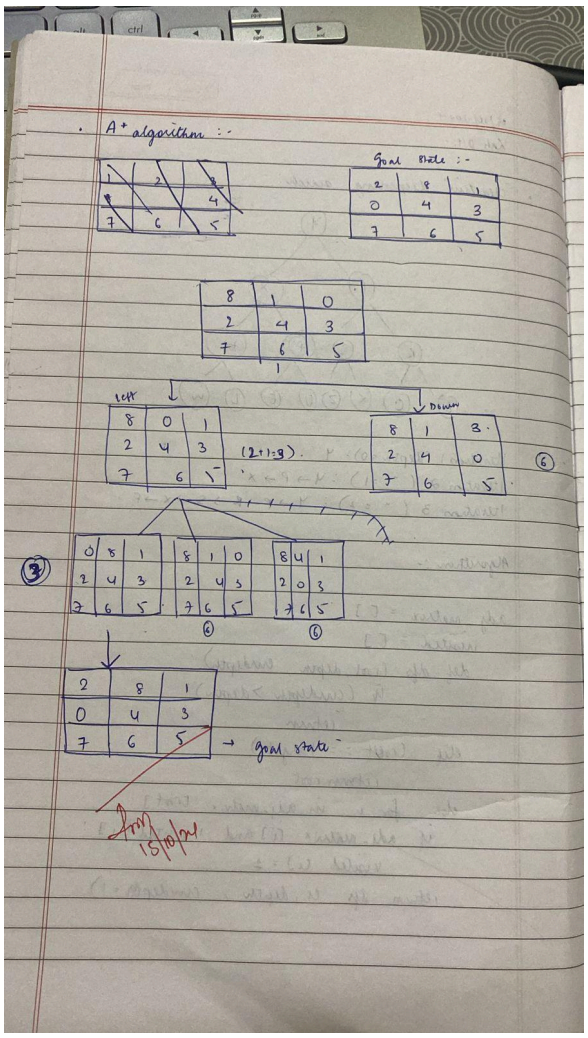
```
            result = dls(new_state, goal_state, depth - 1, visited)
            if result:
                return result
    visited.remove(tuple(map(tuple, state.board))) # Unmark this state
    return None

def print_solution(solution):
    path = []
    while solution:
        path.append(solution.board)
        solution = solution.previous
    for board in reversed(path):
        for row in board:
            print(row)
        print()

# Define the initial state and goal state
initial_state = PuzzleState(
    board=[[1, 2, 3],
           [4, 0, 5],
           [7, 8, 6]],
    zero_pos=(1, 1)
)
goal_state = [ [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]

# Perform Iterative Deepening Search
max_depth = 20 # You can adjust this value
solution = ids(initial_state, goal_state, max_depth)

if solution: print("Solution
    found:")
    print_solution(solution)
else:
    print("No solution found.")
```

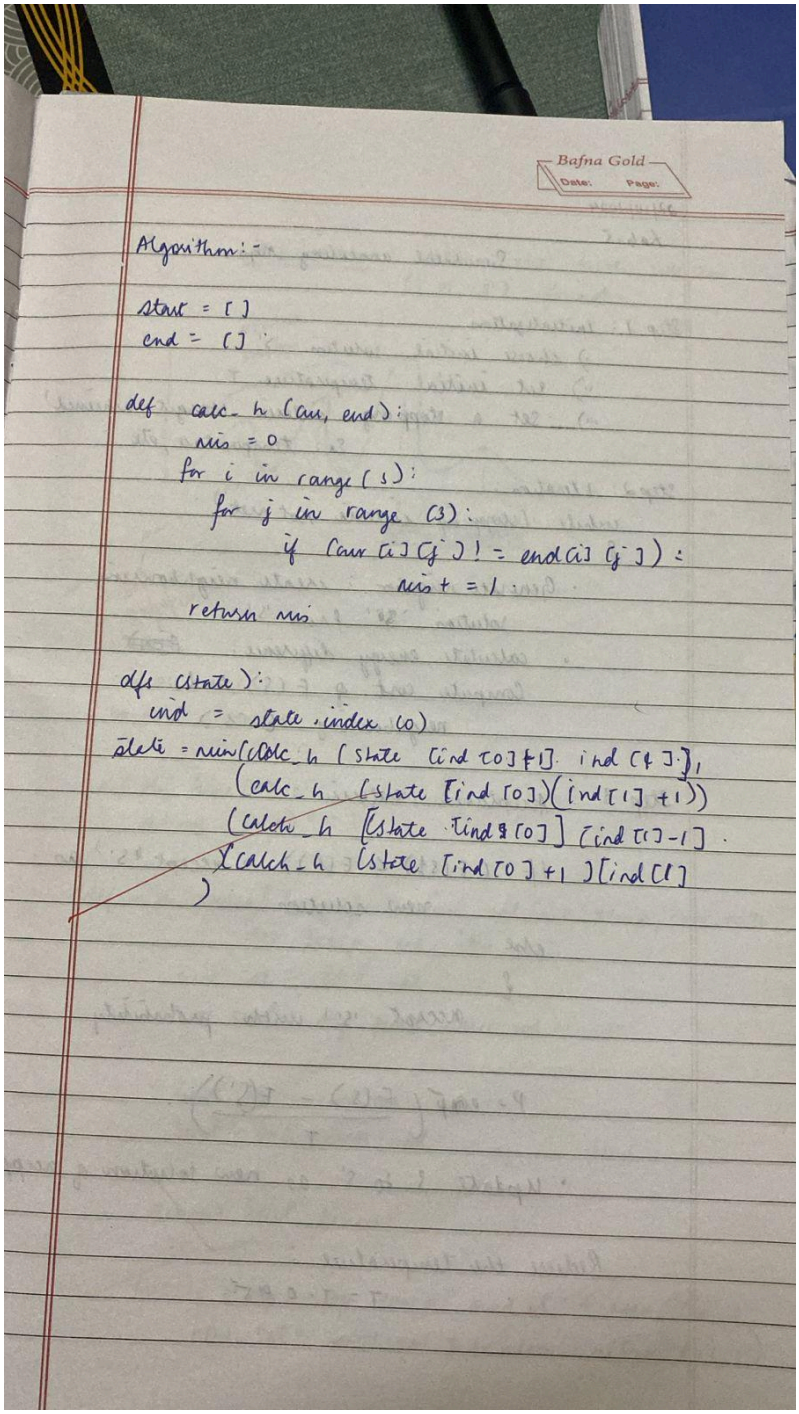
```
Solution found:
[1, 2, 3]
[4, 0, 5]
[7, 8, 6]

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

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

Program 3

Implement A* search algorithm

Algorithm:




Bafna Gold
Date: Page:
Algorithm:-
start = []
end = []
def calc_h (cur, end):
mis = 0
for i in range (3):
for j in range (3):
if (cur[i][j] != end[i][j]):
mis += 1
return mis
dfs (state):
ind = state.index (0)
sleft = min(calc_h (state [ind[0]+1][ind[1]],
(calc_h (state [ind[0]][ind[1]+1]))
(calch_h [state [ind[0]][ind[1]-1]
(calch_h (state [ind[0]+1][ind[1]]
)

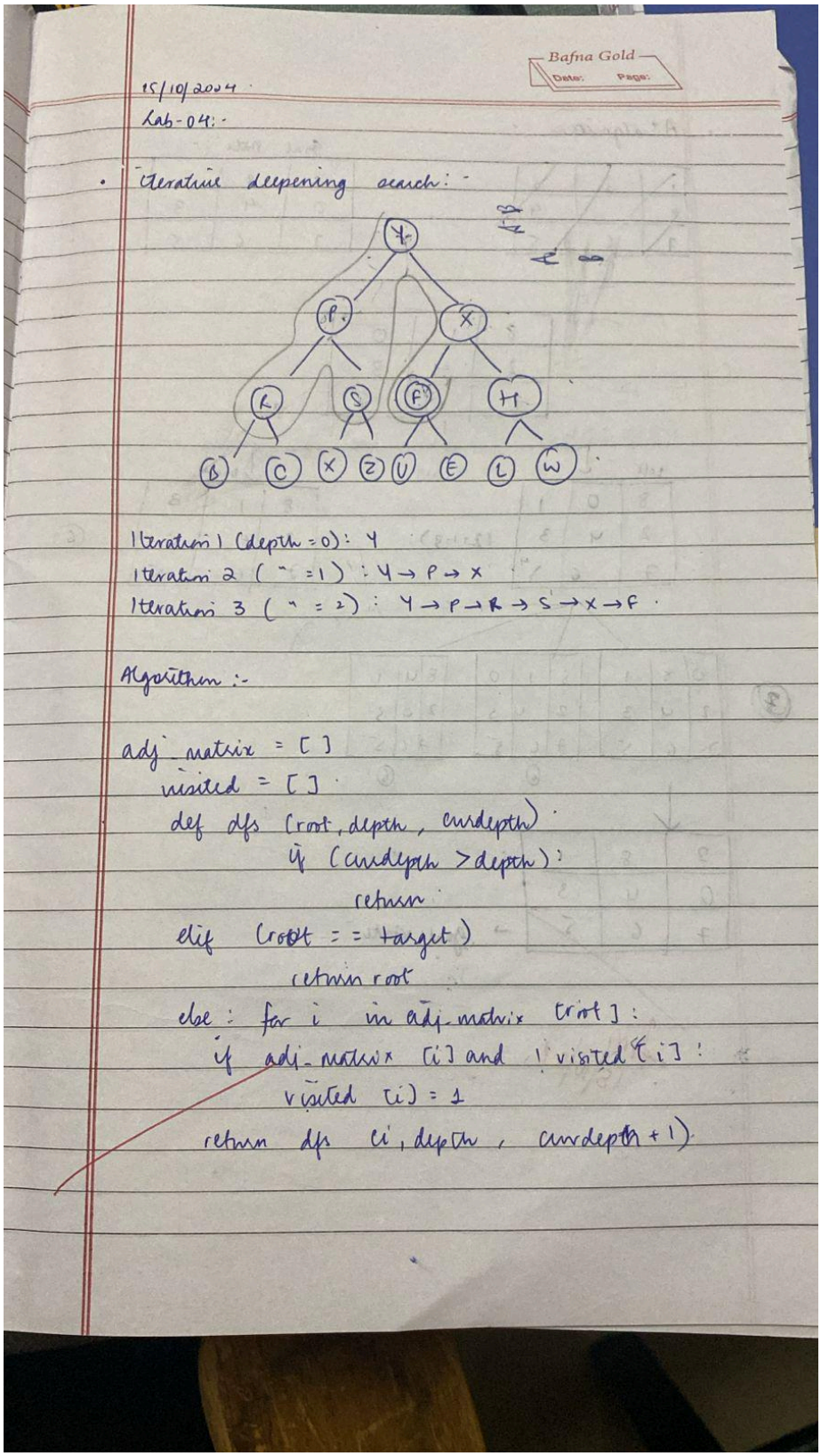
Code:

```
import heapq

def manhattan(curr, goal):
    ans = 0
    for i in range(3):
        for j in range(3):
            for k in range(3):
                for l in range(3):
                    if goal[i][j] == curr[k][l]:
                        ans += abs(i - k) + abs(j - l)
    return ans

def astar(start, goal):
    open_set = []
    heapq.heappush(open_set, (manhattan(start, goal), start))
    close_set = set()
    gscore = {}
    gscore[tuple(map(tuple, start))] = 0
    parent = {}

    while open_set:
        _, curr = heapq.heappop(open_set)
        if curr == goal:
            return path(parent, curr)
        close_set.add(tuple(map(tuple, curr)))
        for neighbour in neighbours(curr):
            if tuple(map(tuple, neighbour)) in close_set:
                continue
            new_g = gscore[tuple(map(tuple, curr))] + 1
            if tuple(map(tuple, neighbour)) not in gscore or new_g < gscore[tuple(map(tuple, neighbour))]:
                parent[tuple(map(tuple, neighbour))] = curr
                gscore[tuple(map(tuple, neighbour))] = new_g
                heapq.heappush(open_set, (new_g + manhattan(neighbour, goal), neighbour))
    return "No solution"

def neighbours(curr):
    n = []
    x, y = 0, 0
    directions = [[1, 0], [0, 1], [-1, 0], [0, -1]]
    for i in range(3):
        for j in range(3):
            if curr[i][j] == 0:
                x, y = i, j
                break
    for dx, dy in directions:
```

```
        if 0 <= x + dx < 3 and 0 <= y + dy < 3:
            new_state = [row.copy() for row in curr]
            new_state[x][y], new_state[x + dx][y + dy] = new_state[x + dx][y + dy], new_state[x][y]
            n.append(new_state)
    return n

def path(parent, curr):
    fol = [curr]
    while tuple(map(tuple, curr)) in parent:
        curr = parent[tuple(map(tuple, curr))]
        fol.append(curr)
    return list(reversed(fol))

start = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]
goal = [[2, 8, 1], [0, 4, 3], [7, 6, 5]]
result = astar(start, goal)
if result != "No solution":
    for ind,state in enumerate(result):
        print(f"Step: {ind}")
        for row in state:
            print(row)
        print()
    print("Goal Reached")
else:
    print(result)
```

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

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

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

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

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

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

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

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

Step: 9
[2, 8, 1]
[0, 4, 3]
[7, 6, 5]

Goal Reached
```

Manhattan Distance




```
def manhattan_distance(state, goal):
    distance = 0
```

```
    for i in range(3):
        for j in range(3):
            tile = state[i][j]
            if tile != 0: # Ignore the blank space (0)
                # Find the position of the tile in the goal state
                for r in range(3):
                    for c in range(3):
                        if goal[r][c] == tile:
                            target_row, target_col = r, c
                            break
                # Add the Manhattan distance (absolute difference in rows and columns)
                distance += abs(target_row - i) + abs(target_col - j)
    return distance

def findmin(open_list, goal):
    minv = float('inf')
    best_state = None
    for state in open_list:
        h = manhattan_distance(state['state'], goal) # Use Manhattan distance here
        f = state['g'] + h
        if f < minv:
            minv = f
            best_state = state
    open_list.remove(best_state)
    return best_state

def operation(state):
    next_states = []
    blank_pos = find_blank_position(state['state'])
    for move in ['up', 'down', 'left', 'right']:
        new_state = apply_move(state['state'], blank_pos, move)
        if new_state:
            next_states.append({
                'state': new_state,
                'parent': state,
                'move': move,
                'g': state['g'] + 1
            })
    return next_states

def find_blank_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
```

```
                return i, j
    return None

def apply_move(state, blank_pos, move):
    i, j = blank_pos
    new_state = [row[:] for row in state]
    if move == 'up' and i > 0:
        new_state[i][j], new_state[i - 1][j] = new_state[i - 1][j], new_state[i][j]
    elif move == 'down' and i < 2:
        new_state[i][j], new_state[i + 1][j] = new_state[i + 1][j], new_state[i][j]
    elif move == 'left' and j > 0:
        new_state[i][j], new_state[i][j - 1] = new_state[i][j - 1], new_state[i][j]
    elif move == 'right' and j < 2:
        new_state[i][j], new_state[i][j + 1] = new_state[i][j + 1], new_state[i][j]
    else:
        return None
    return new_state

def print_state(state):
    for row in state:
        print(' '.join(map(str, row)))

# Initial state and goal state
initial_state = [[2,8,3], [1,6,4], [7,0,5]]
goal_state = [[1,2,3], [8,0,4], [7,6,5]]

# Open list and visited states
open_list = [{'state': initial_state, 'parent': None, 'move': None, 'g': 0}]
visited_states = []

while open_list:
    best_state = findmin(open_list, goal_state)

    print("Current state:")
    print_state(best_state['state'])

    h = manhattan_distance(best_state['state'], goal_state) # Using Manhattan distance here
    f = best_state['g'] + h
    print(f"g(n): {best_state['g']}, h(n): {h}, f(n): {f}")

    if best_state['move'] is not None:
        print(f"Move: {best_state['move']}")
    print()
    if h == 0: # Goal is reached if h == 0
        goal_state_reached = best_state
        break
```

```
    visited_states.append(best_state['state'])
    next_states = operation(best_state)

    for state in next_states:
        if state['state'] not in visited_states:
            open_list.append(state)

# Reconstruct the path of moves
moves = []
while goal_state_reached['move'] is not None:
    moves.append(goal_state_reached['move'])
    goal_state_reached = goal_state_reached['parent']
moves.reverse()

print("\nMoves to reach the goal state:", moves)
print("\nGoal state reached:")
print_state(goal_state)
```

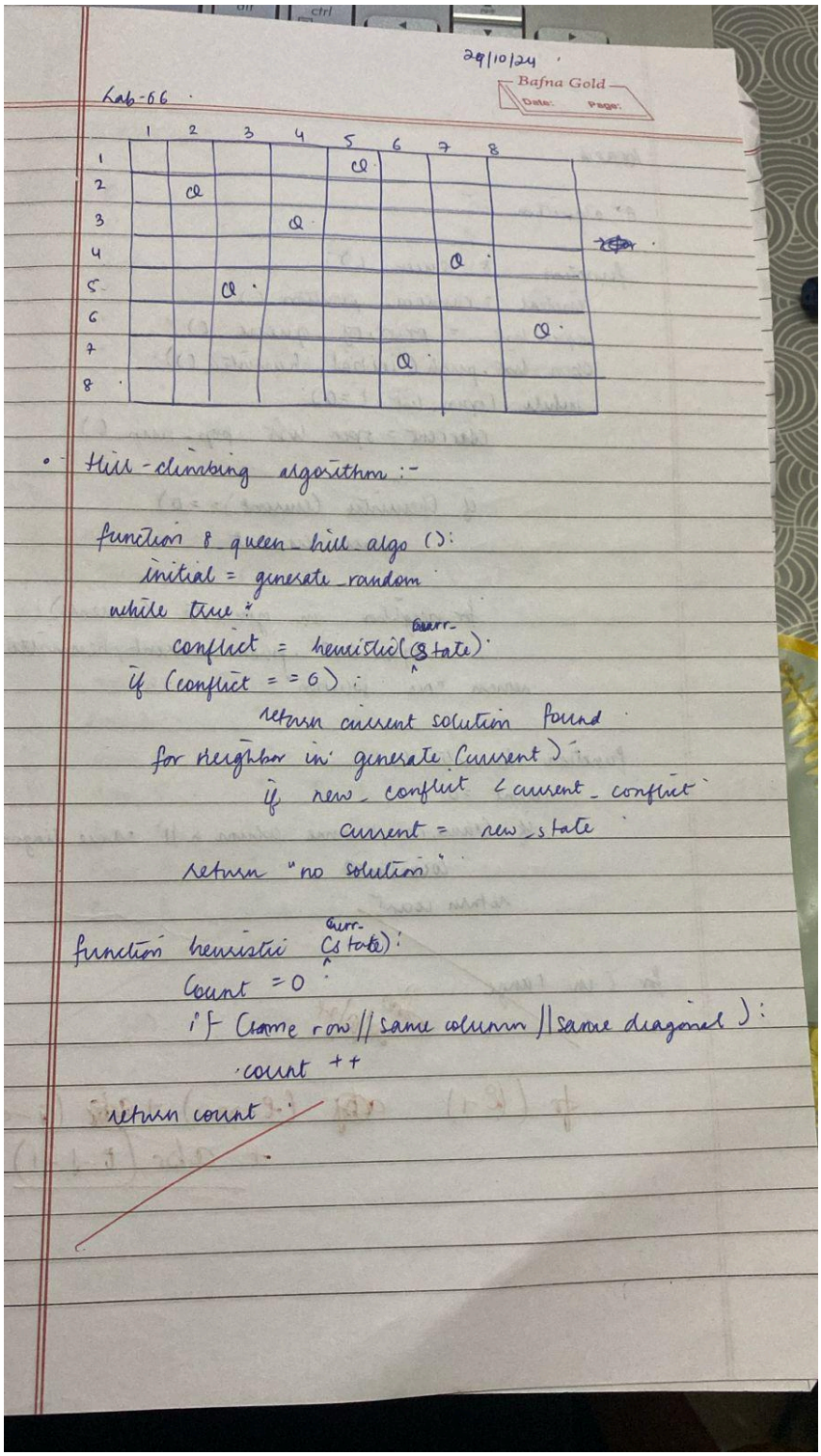
```
Current state:
2 8 3
1 6 4
7 0 5
g(n): 0, h(n): 5, f(n): 5

Current state:
2 8 3
1 0 4
7 6 5
g(n): 1, h(n): 4, f(n): 5
Move: up

Current state:
2 0 3
1 8 4
7 6 5
g(n): 2, h(n): 3, f(n): 5
Move: up

Current state:
0 2 3
1 8 4
7 6 5
g(n): 3, h(n): 2, f(n): 5
Move: left

Current state:
1 2 3
0 8 4
7 6 5
g(n): 4, h(n): 1, f(n): 5
Move: down
```

```
Current state:
1 2 3
8 0 4
7 6 5
g(n): 5, h(n): 0, f(n): 5
Move: right


Moves to reach the goal state: ['up', 'up', 'left', 'down', 'right']

Goal state reached:
1 2 3
8 0 4
7 6 5
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



29/10/24

Bafna Gold

Lab-06

	1	2	3	4	5	6	7	8
1					Q			
2		Q						
3				Q				
4							Q	
5			Q					
6								Q
7						Q		
8								

- Hill-climbing algorithm :-

```
function 8 queen_hill_algo ():
    initial = generate_random
    while true :
        conflict = heuristic(curr_state)
        if (conflict == 0) :
            return current solution found
        for neighbor in generate(current) :
            if new_conflict < current_conflict
                current = new_state
        return "no solution"

function heuristic (curr_state):
    count = 0
    if (same row // same column // same diagonal ):
        count ++
    return count
```

Lab 6:

A* algorithm :-

```
function 8_Queen ():
    initial = random_position ()
    open_list = priority_queue ()
    open_list.push (initial, heuristic ())
    while (open_list != 0):
        current = open_list.pop_min ()

        if (heuristic (current) == 0)
            return current

        for neighbor in generate (current):
            open_list.push (current, heuristic ())
    return "no solution".

function heuristic ():
    count = 0
    if (same row || same column || same diagonal):
        count++
    return count
```

for i in range

for ((i+1) (i-row) + abs (j-col)
+ abs (i-j+1)

Code:

```
import random

def calculate_conflicts(board):
    conflicts = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1
            return conflicts

def hill_climbing(n):
    cost=0
    while True:
        # Initialize a random board
        current_board = list(range(n))
        random.shuffle(current_board)
        current_conflicts = calculate_conflicts(current_board)

        while True:
            # Generate neighbors by moving each queen to a different position
            found_better = False
            for i in range(n):
                for j in range(n):
                    if j != current_board[i]: # Only consider different positions
                        neighbor_board = list(current_board)
                        neighbor_board[i] = j
                        neighbor_conflicts = calculate_conflicts(neighbor_board)
                        if neighbor_conflicts < current_conflicts:
                            print_board(current_board)
                            print(current_conflicts)
                            print_board(neighbor_board)
                            print(neighbor_conflicts)
                            current_board = neighbor_board
                            current_conflicts = neighbor_conflicts
                            cost+=1
                            found_better = True
                            break
                if found_better:
                    break

            # If no better neighbor found, stop searching
            if not found_better:
                break
```

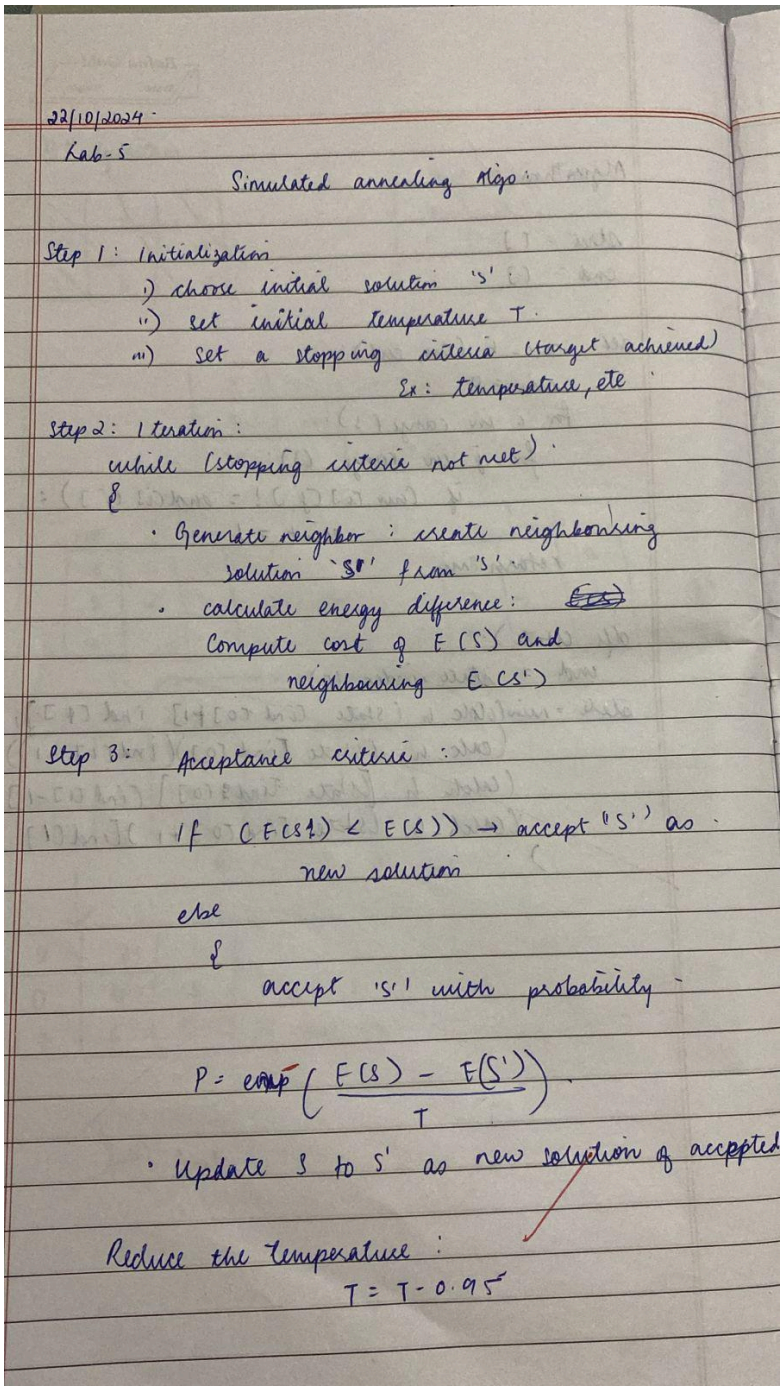
```
        # If a solution is found (zero conflicts), return the board
        if current_conflicts == 0:
            return current_board, current_conflicts, cost

def print_board(board):
    n = len(board)
    for i in range(n):
        row = ['.'] * n
        row[board[i]] = 'Q' # Place a queen
        print(' '.join(row))
    print()
    print("===============")
# Example Usage
n = 4
solution, conflicts, cost = hill_climbing(n)
print("Final Board Configuration:")
print_board(solution)
print("Number of Cost:", cost)
```

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

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



22/10/2024

Lab-5

Simulated annealing algo:

Step 1: Initialization

i) choose initial solution 'S'.

ii) set initial temperature T.

iii) Set a stopping criteria (target achieved)

Ex: temperature, etc

Step 2: Iteration:

while (stopping criteria not met).

{

- Generate neighbor: create neighbouring solution 'S'' from 'S'.
- calculate energy difference: ~~E(s)~~ Compute cost of F(S) and neighbouring E(S')

Step 3: Acceptance criteria:

If (F(S') < F(S)) → accept 'S'' as new solution

else

{

accept 'S' with probability

$$P = \exp\left(\frac{F(S) - F(S')}{T}\right)$$

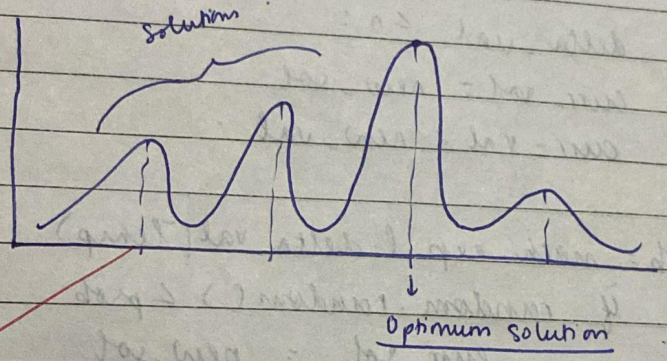
- Update S to S' as new solution of accepted

Reduce the temperature:

T = T - 0.95

Bafna Gold
Date: Page:

Step 4: Termination : Return the best solution (S or S') found.



22/12/24

Code:

```
import math

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

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

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

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

        new_energy = energy(new)

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

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

    return current

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

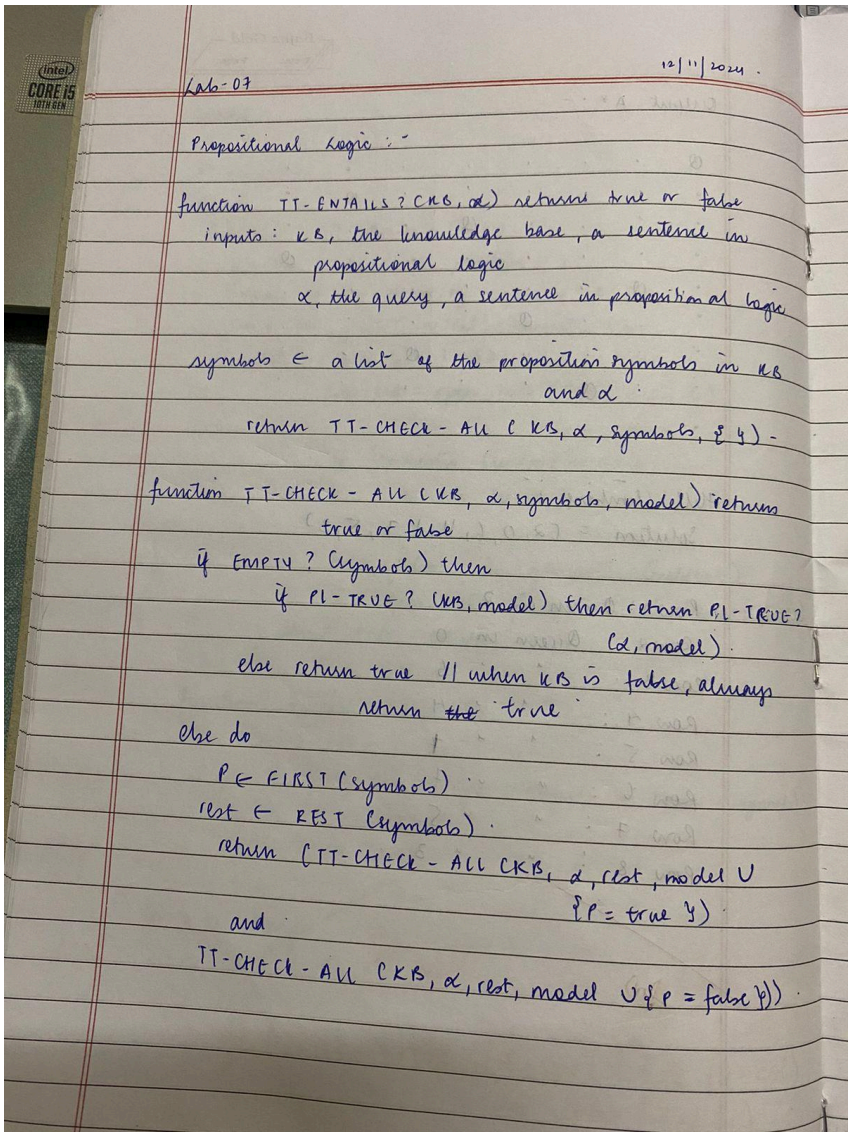
print(f"Best solution found: {best_solution}")

```
Best solution found: -0.7323104061658242
```

Program 6

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

Algorithm:



6. $\forall x \forall y \ ((P(x) \wedge child(x, y_1) \wedge child(x, y_2)) \rightarrow S(y_1, y_2))$

If x is parent of 2 children y_1 and y_2, then y_1 and y_2 are siblings.
then Charlie and Bob are siblings.

∴ It is true.

Done
21/4/24

~~Code:~~

~~import itertools~~

Output:

P	Q	R	Result
F	f	f	f
f	f	T	T
f	T	f	f
f	T	T	T
T	f	f	f
T	f	T	T
T	T	f	T
T	T	T	T

Code:

```
import itertools

def evaluate_formula(formula, valuation):
```

Code:

```
import itertools


def evaluate_formula(formula, valuation):

    formula = formula.replace('p', str(valuation['p']))
    formula = formula.replace('q', str(valuation['q']))

    return eval(formula)


def extract_variables(formula):
    variables = set()
    for char in formula:
        if char.isalpha():
            variables.add(char)
    return list(variables)


def generate_truth_table(KB, query):

    variables = extract_variables(KB) + extract_variables(query)
    variables = list(set(variables))

    print("Truth Table:")
    print(" | ".join(variables + ["KB", "Query"]))
    print("-" * (len(variables) * 4 + 12))


    entails_query = True


    for assignment in itertools.product([False, True], repeat=len(variables)):
        valuation = dict(zip(variables, assignment))

        KB_truth = evaluate_formula(KB, valuation)
        query_truth = evaluate_formula(query, valuation)


        row = [str('T' if valuation[var] else 'F') for var in variables]
        row.append(str('T' if KB_truth else 'F'))
        row.append(str('T' if query_truth else 'F'))
        print(" | ".join(row))
```

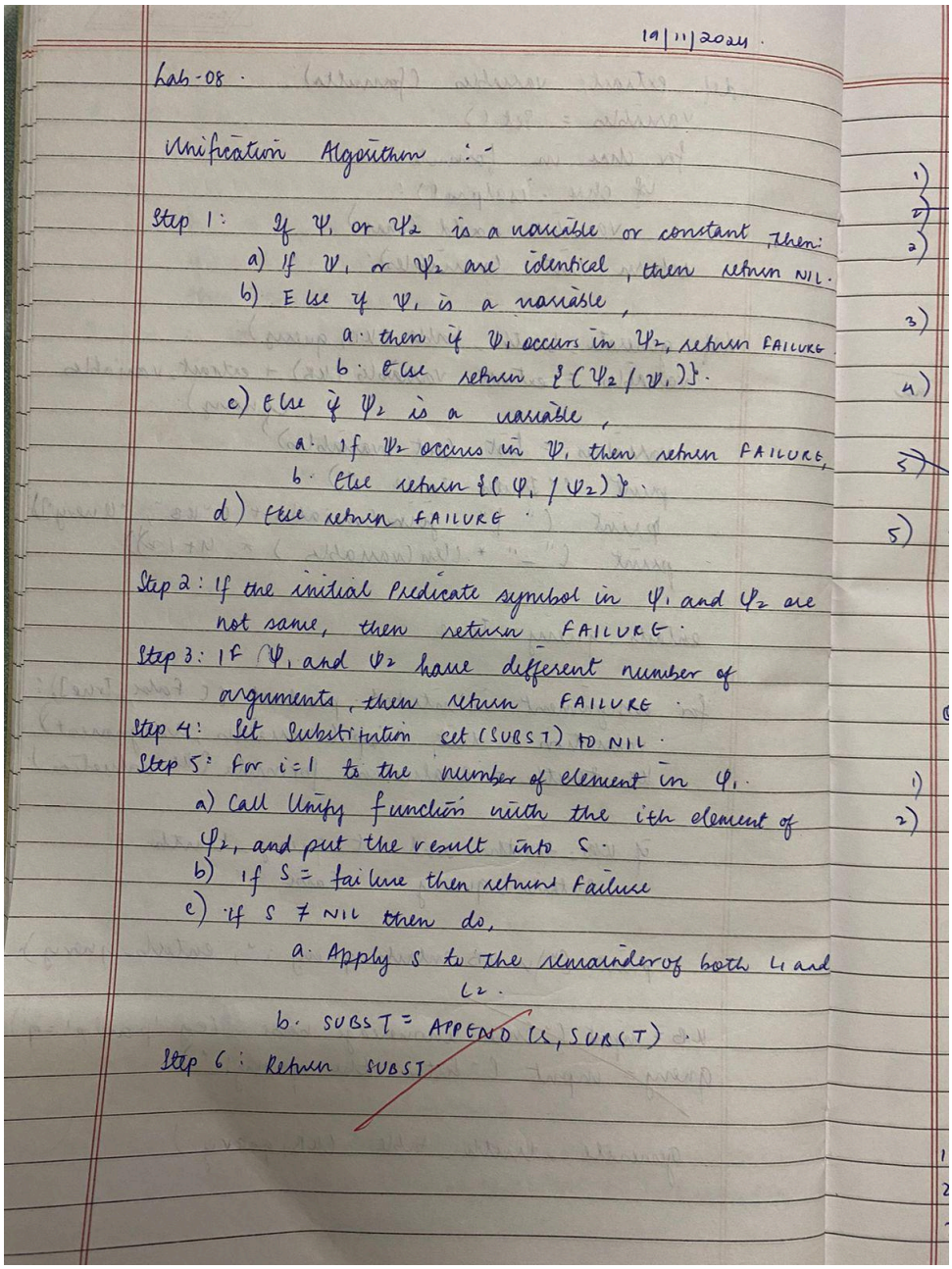
```
        if KB_truth and not query_truth:
            entails_query = False

    print("\nKB entails query:", entails_query)


KB = input("Enter the knowledge base (e.g., 'p and (p != q)'): ")
query = input("Enter the query (e.g., 'q'): ")


generate_truth_table(KB, query)
```

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

Program 7

Implement unification in first order logic

Algorithm:



19/11/2024.

Lab-08.

Unification Algorithm :-

Step 1: If Ψ_1 or Ψ_2 is a variable or constant, then:
a) If Ψ_1 or Ψ_2 are identical, then return NIL.
b) Else if Ψ_1 is a variable,
a. then if Ψ_1 occurs in Ψ_2, return FAILURE
b. Else return $\{(\Psi_2 / \Psi_1)\}$.
c) Else if Ψ_2 is a variable,
a. If Ψ_2 occurs in Ψ_1 then return FAILURE,
b. Else return $\{(\Psi_1 / \Psi_2)\}$.
d) Else return FAILURE.

Step 2: If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE.

Step 3: If Ψ_1 and Ψ_2 have different number of arguments, then return FAILURE.

Step 4: Let Substitution set (SUBST) to NIL.

Step 5: For i=1 to the number of element in Ψ_1.
a) Call Unify function with the ith element of Ψ_2, and put the result into S.
b) If S = failure then return Failure
c) If S ≠ NIL then do,
a. Apply S to the remainder of both L1 and L2.
b. SUBST = APPEND (S, SUBST).

Step 6: Return SUBST

Implementation of First Order Logic :-

1) "Every ~~mortal~~ human is mortal".
2) $\forall x$ (human (x) → mortal (x)).
2) "John is a human".
H (j).
3) ~~"Every human~~ "John loves Mary".
L (j, m).
4) "Every dog is an animal".
$\forall x$ (D(x) → A(x)).
5) ~~"If it rains, then the ground is wet"~~.
5) "A number is even if and only if it is divisible by 2".
=) $\forall x$ (Even (x) ↔ Divisible By (x, 2)).

~~Output :-~~

Unification ~~used for~~ :-

1) Loves (John, x) - =) x is a variable
2) Loves (~~John~~, Mary) =) ~~second argument is~~ ~~Mary~~ : is variable.

Substitute x with Mary and y with John.

Unified sentence : Loves (John, Mary)

∴ Loves (John, mary) = Loves (John, mary).

Output :

Enter sentence like :

1. John is a human
2. Every human is mortal
3. John loves mary.

Code:

```
class
   UnificationError(Excepti
   on):
   pass

def unify(expr1, expr2,
   substitutions=dict()):
   # if substitutions is
   None:
   #     substitutions = {}

   # If both expressions are
   identical, return current
   substitutions
   if expr1 == expr2:
      return substitutions

   # If the first expression is
   a variable
   if is_variable(expr1):
      return
   unify_variable(expr1,
   expr2, substitutions)

   # If the second
   expression is a variable
   if is_variable(expr2):
      return
   unify_variable(expr2,
   expr1, substitutions)

   # If both expressions are
   compound expressions
   if is_compound(expr1)
   and
   is_compound(expr2):
      if expr1[0] != expr2[0]
   or len(expr1[1:]) !=
   len(expr2[1:]):
         raise
   UnificationError("Expres
   sions do not match.")
      return
   unify_lists(expr1[1:],
   expr2[1:],
   unify(expr1[0], expr2[0],
```

```
    substitutions))

    # If expressions are not
    compatible
    raise
    UnificationError(f"Cann
    ot unify {expr1} and
    {expr2}.")

def unify_variable(var,
    expr, substitutions):
    if var in substitutions:
        return
    unify(substitutions[var],
    expr, substitutions)
    elif occurs_check(var,
    expr, substitutions):
        raise
    UnificationError(f"Occur
    s check failed: {var} in
    {expr}.")
    else:
        substitutions[var] =
    expr
        return substitutions

def unify_lists(list1, list2,
    substitutions):
    for expr1, expr2 in
    zip(list1, list2):
        substitutions =
    unify(expr1, expr2,
    substitutions)
    return substitutions

def is_variable(term):
    return isinstance(term,
    str) and term[0].islower()

def is_compound(term):
    return isinstance(term,
    (list, tuple)) and
    len(term) > 0

def occurs_check(var, expr,
    substitutions):
    if var == expr:
        return True
```

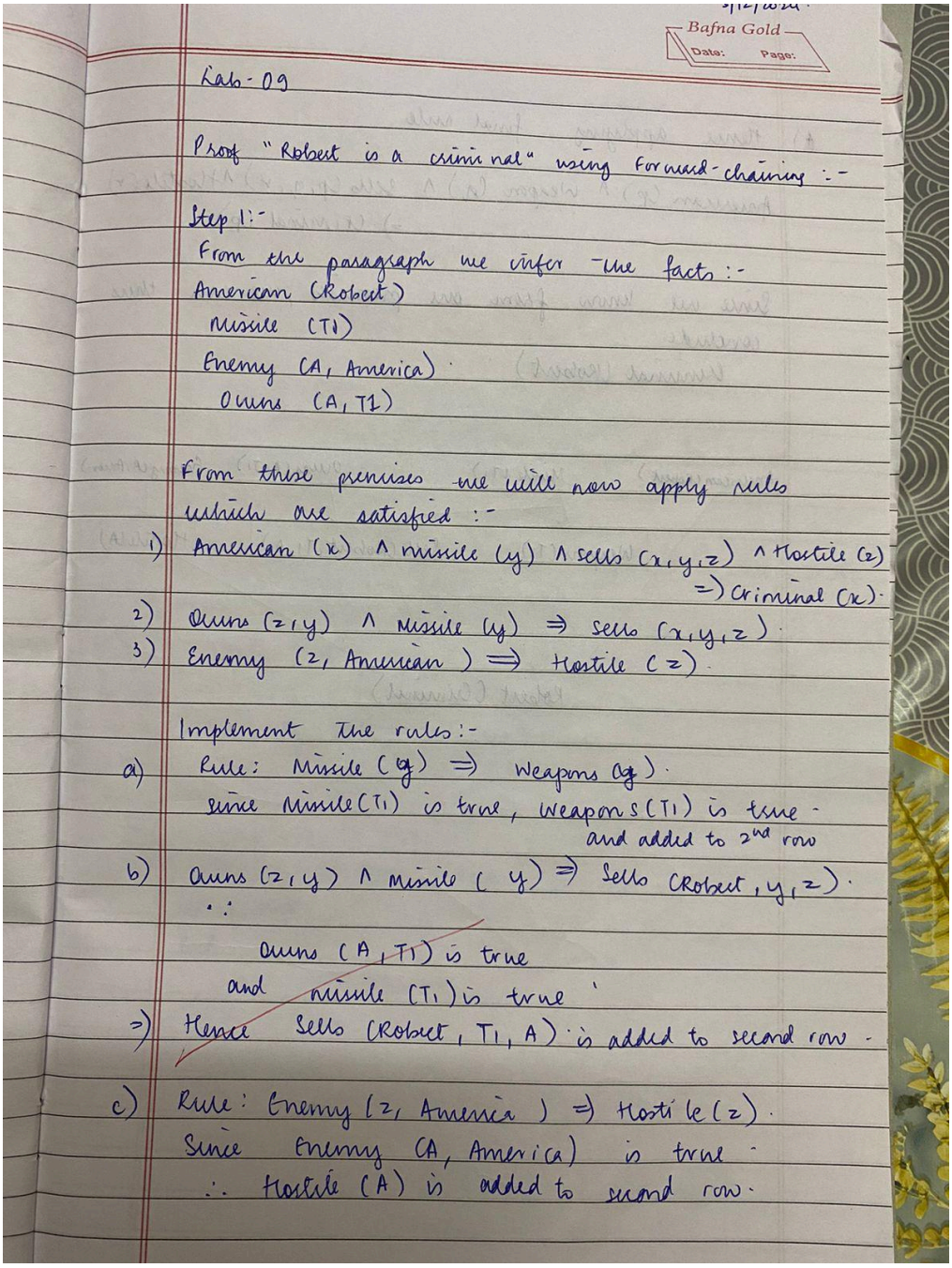
```
elif is_compound(expr):
    return
any(occurs_check(var,
sub, substitutions) for
sub in expr)
elif expr in substitutions:
    return
occurs_check(var,
substitutions[expr],
substitutions)
return False
```

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

Program 8

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

Algorithm:

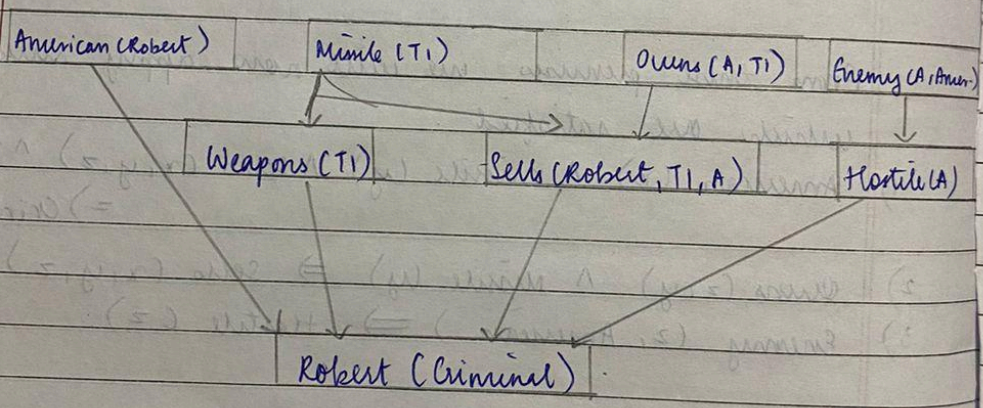


:

4) Hence applying final rule.

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

Since we know from our premises, we can thus conclude
Criminal (Robert).



Code:

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

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

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

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

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

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

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

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

# Run forward chaining to attempt to derive the conclusion
```

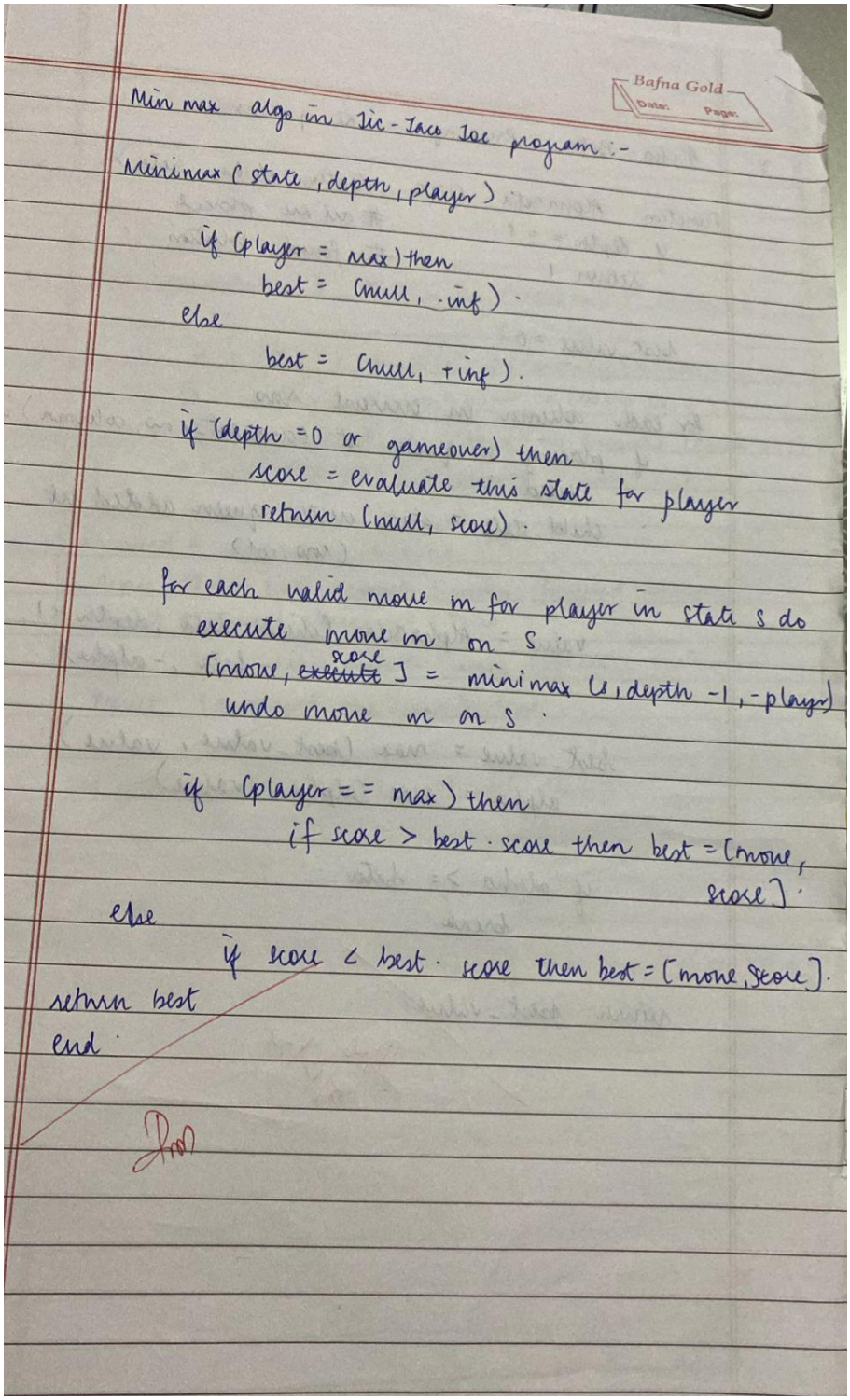
forward_chaining()

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

Program 9

Using min max algo for tic tac toe and alpha beta pruning algorithm for N-queens:

Algorithm:



Min max algo in Tic-Tac Toe program :-

```
Minimax ( state , depth , player ).

    if (player = max) then
        best = (null, -inf).
    else
        best = (null, +inf).

    if (depth = 0 or gameover) then
        score = evaluate this state for player
        return (null, score).

    for each valid move m for player in state s do
        execute move m on s.
        [move, score] = minimax (s, depth -1, -player)
        undo move m on s.

    if (player == max) then
        if score > best.score then best = [move,
                                           score].
    else
        if score < best.score then best = [move, score].
return best
end.
```

Code:

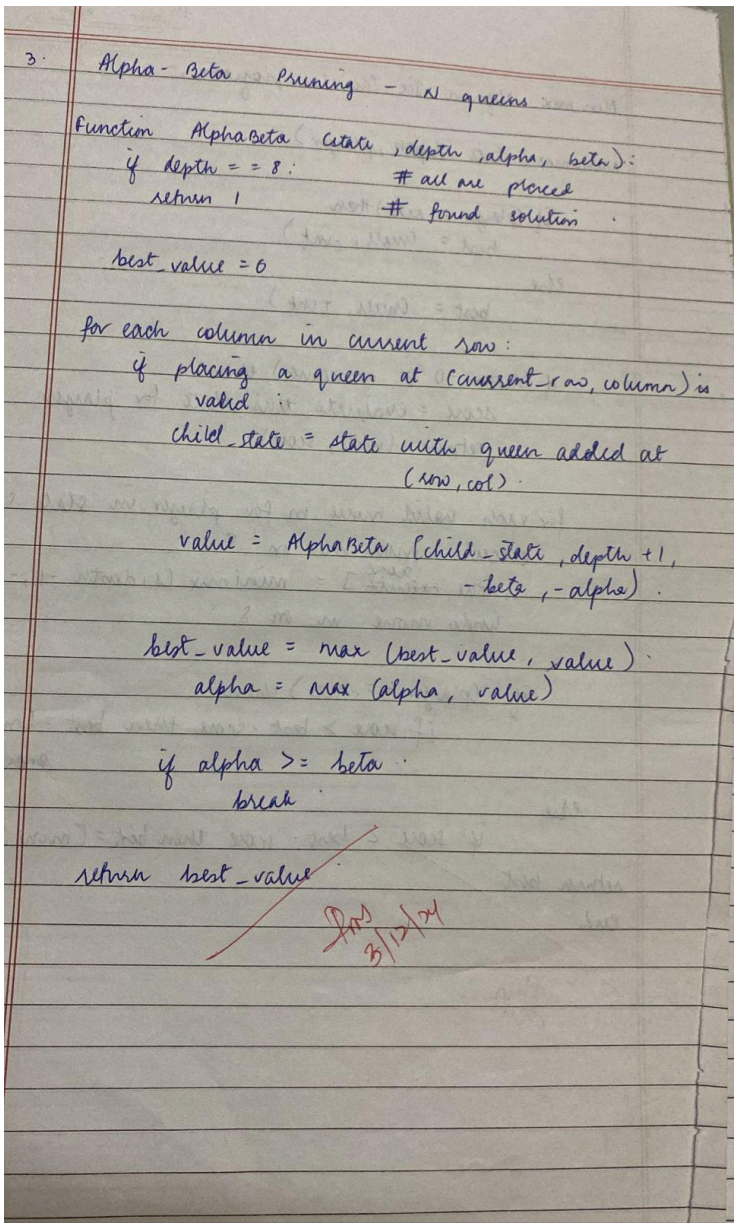
```
import math

# Constants for the players
AI = 'X'
HUMAN = 'O'
EMPTY = '_'

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

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

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

# Minimax algorithm
def minimax(board, depth, is_maximizing):
    if check_winner(board, AI):
        return 10 - depth
    if check_winner(board, HUMAN):
        return depth - 10
    if is_draw(board):
        return 0

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

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

# Function to find the best move for AI
def find_best_move(board):
    best_score = -math.inf
    move = (-1, -1)
    for i in range(3):
        for j in range(3):
            if board[i][j] == EMPTY:
                board[i][j] = AI
                score = minimax(board, 0, False)
                board[i][j] = EMPTY
                if score > best_score:
                    best_score = score
                    move = (i, j)
    return move

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

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

```
Current Board:
X O X
O X O

_ _ _

The best move for AI is: (2, 0)
```

Implement Alpha-Beta Pruning.

Algorithm:



3. Alpha - Beta Pruning - N queens

```
Function AlphaBeta (state, depth, alpha, beta):
    if depth == 8:                  # all are placed
        return 1                    # found solution

    best_value = 0

    for each column in current row:
        if placing a queen at (current_row, column) is valid:
            child_state = state with queen added at (row, col)

            value = AlphaBeta (child_state, depth + 1,
                               - beta, - alpha)

            best_value = max (best_value, value)
            alpha = max (alpha, value)

            if alpha >= beta
                break

    return best_value
```

Done 3/12/24

Code:

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

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

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

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

        return True

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

```
    if maximizing_player:
        max_eval =
float('-inf')
        best_board = None
        for row in
range(self.size):
            if
self.is_safe(board, row,
col):

board[row][col] = 1
                eval_score,
potential_board =
self.alpha_beta_search(b
oard, col + 1, alpha, beta,
False)

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

board[row][col] = 1
                eval_score,
potential_board =
self.alpha_beta_search(b
oard, col + 1, alpha, beta,
True)
```

```
    board[row][col] = 0
                if eval_score <
    min_eval:
                    min_eval =
    eval_score
                    best_board =
    potential_board
                beta =
    min(beta, eval_score)
                if beta <=
    alpha:  # Alpha cutoff
                    break
            return min_eval,
    best_board

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

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


if __name__ ==
    "__main__":
    game = EightQueens()
    solution = game.solve()
    if solution:
        print("Solution
    found:")

    game.print_board(solutio
    n)
    else:
```

```
    print("No solution
exists.")
```

```
Enter numbers for the game tree (space-separated): 10 9 14 18 5 4 50 3
Final Result of Alpha-Beta Pruning: 50
```