VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Priyanshu Kumar (1BM22CS210)

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

Swathi Sridharan
Assistant Professor
Department of 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 **Priyanshu Kumar (1BM22CS210),** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Jyothi S Nayak Professor & HOD Department of CSE, BMSCE

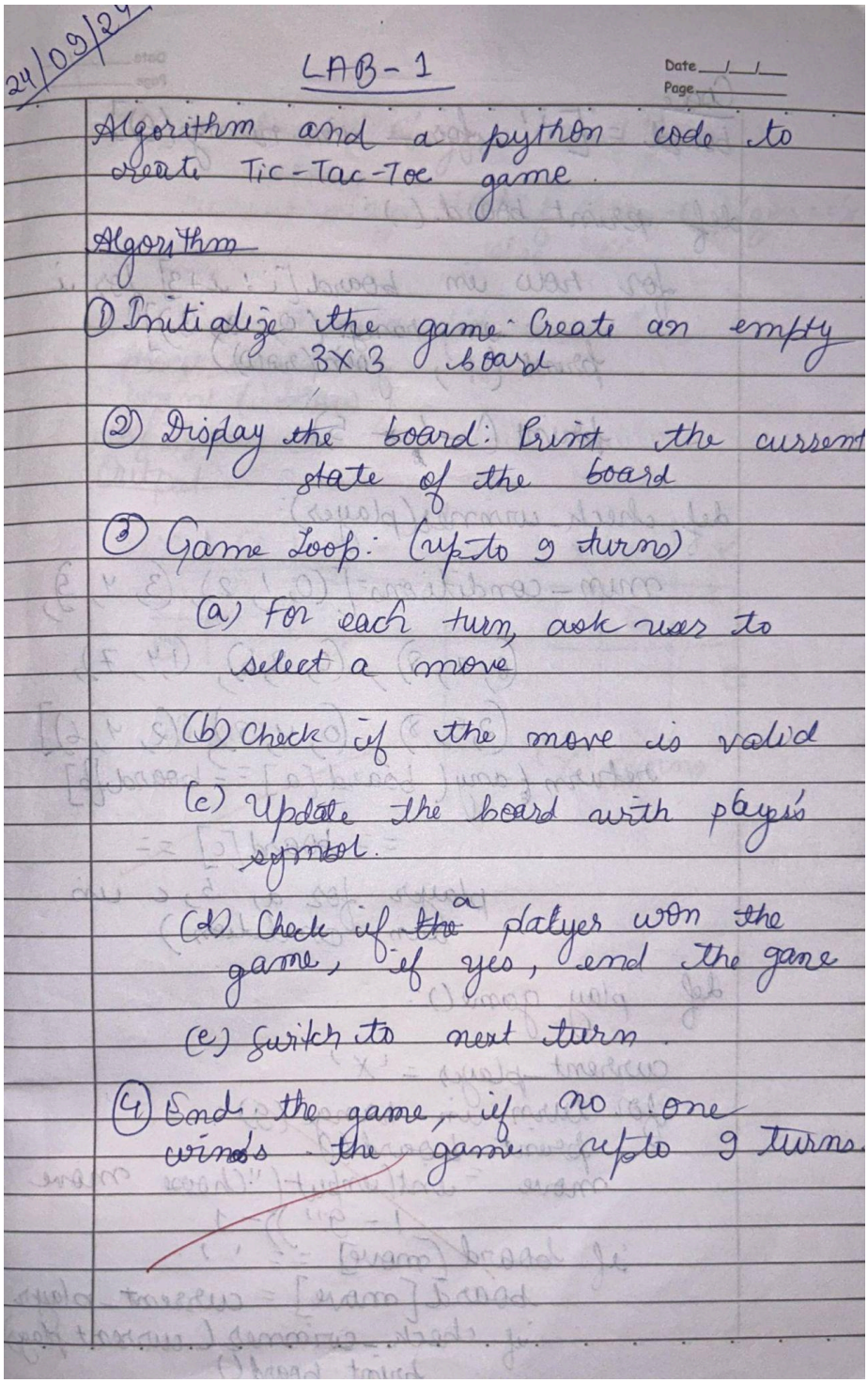
Index

Github Link: https://github.com/pkcs210/AI-Lab

Lab 1

Implement Tic-Tac-Toe Game

Screenshots:



24/09/24

LAB - 1

Date / /
Page

Algorithm and a python code to create Tic-Tac-Toe game.

Algorithm

① Initialize the game: Create an empty 3×3 board

② Display the board: Print the current state of the board

③ Game Loop: (upto 9 turns)

(a) For each turn, ask user to select a move

(b) Check if the move is valid

(c) Update the board with player's symbol.

(d) Check if a player won the game, if yes, end the game

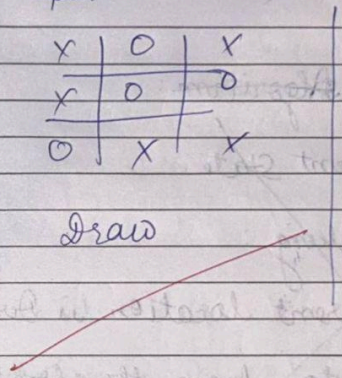
(e) Switch to next turn.

④ End the game, if no one wins the game upto 9 turns.

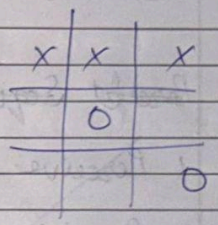
Code

```
board = [' ' for _ in range(9)]

def print_board():
    for row in board[i:i+3] for i
        in range(0, 9, 3)]:
        print('|', join(row))
        print('-' * 5)

def check_winner(player):
    win_conditions = [(0, 1, 2), (3, 4, 5),
                      (6, 7, 8), (0, 3, 6), (1, 4, 7),
                      (2, 5, 8), (0, 4, 8), (2, 4, 6)]
    return any( board[a] == board[b]
                == board[c] ==
                player for a, b, c in
                win_condition )

def play_game():
    current_player = 'X'
    for turn in range(9):
        print_board()
        move = int(input("Choose move
                1 - 9")) - 1
        if board[move] == ' ':
            board[move] = current_player
            if check_winner(current_player):
                print_board()
                print("Player wins!")
                return
            current_player = 'O' if current_player == 'X'
                             else 'X'
        else:
            print("Invalid move")
    print_board()
    print("Draw")
play_game()
```

Output:



Draw



'X' wins

Codes:

```
def print_board(board):
    print(f"{board[0]}|{board[1]}|{board[2]}")
    print("-+-+-")
    print(f"{board[3]}|{board[4]}|{board[5]}")
    print("-+-+-")
    print(f"{board[6]}|{board[7]}|{board[8]}")

def check_winner(board, player):
    win_conditions = [
        [0,1,2],[3,4,5],[6,7,8],
        [0,3,6],[1,4,7],[2,5,8],
        [0,4,8],[2,4,6]
    ]
    for condition in win_conditions:
        if all(board[i] == player for i in condition):
            return True
    return False

def main():
    board = [' ']*9
    current_player = 'X'
    for _ in range(9):
        print_board(board)
        try:
            move = int(input(f"Player {current_player}, enter your move (1-9): ")) -1
            if board[move] == ' ':
                board[move] = current_player
                if check_winner(board, current_player):
                    print_board(board)
                    print(f"Player {current_player} wins!")
                    return
                current_player = 'O' if current_player == 'X' else 'X'
            else:
                print("Invalid move. Try again.")
        except (IndexError, ValueError):
            print("Invalid input. Enter a number from 1 to 9.")
    print_board(board)
    print("It's a draw!")

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

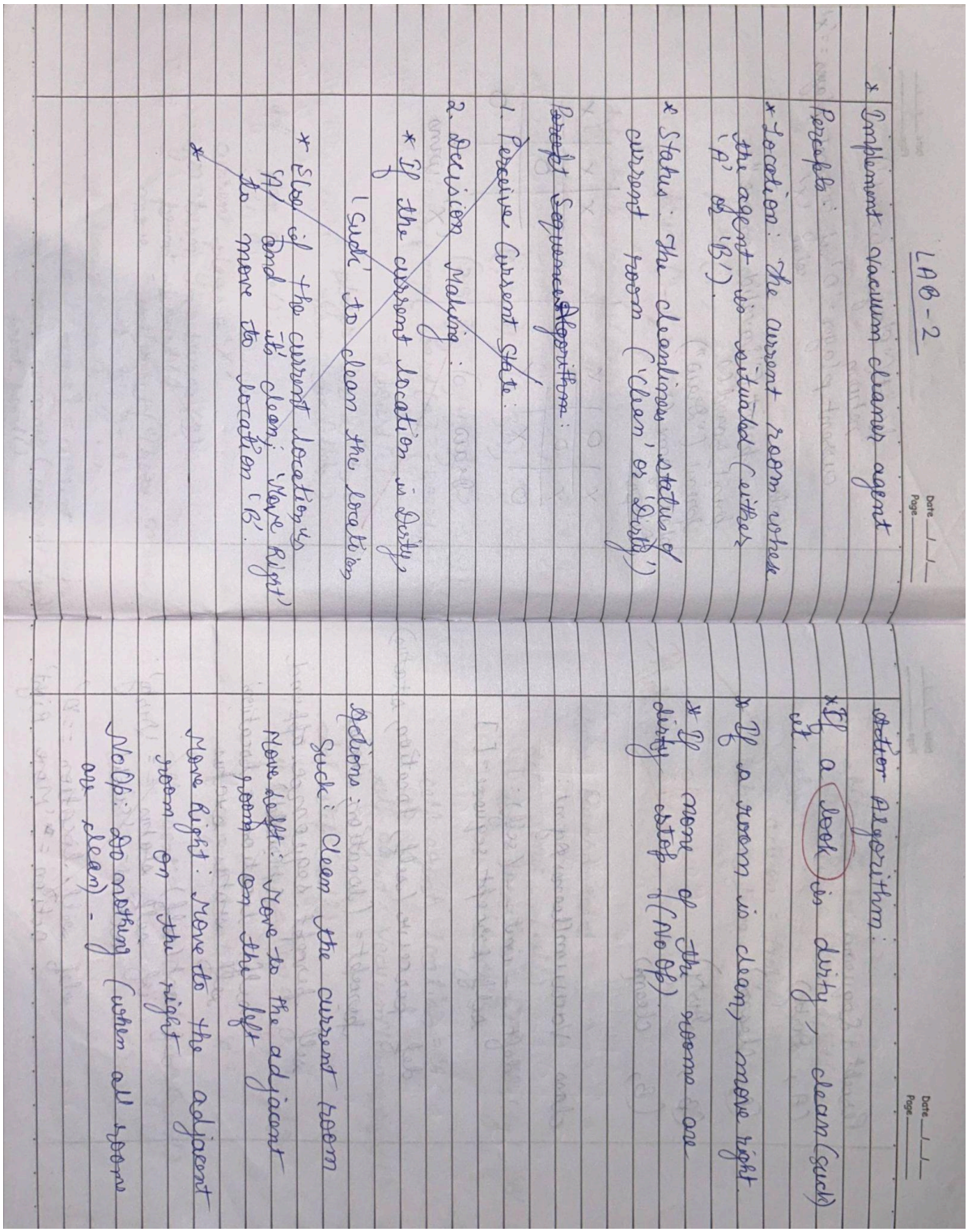
Output:

```
  | |
-+-+-
  | |
-+-+-
  | |
Player X, enter your move (1-9): 1
X| |
-+-+-
  | |
-+-+-
  | |
Player O, enter your move (1-9): 3
X| |O
-+-+-
  | |
-+-+-
  | |
Player X, enter your move (1-9):
```

Lab 2

Implement Tic-Tac-Toe Game

Algorithm:



Percept Sequence:

(A, Dirty)

(A, Clean)

(B, Dirty)

(B, Clean)

```
class VacuumCleanerAgent:
    def __init__(self):
        self.percept_sequence = []

    def perceive(self, location, status):
        percept = (location, status)
        self.percept_sequence.append(percept)
        self.location = location
        self.status = status

    def act(self):
        if self.status == 'Dirty':
            action = 'Suck'
        elif self.location == 'A':
            action = 'Move Right'
        elif self.location == 'B':
            action = 'Move Left'
        else:
            action = 'NoOp'
        return action
```

Pg 1/10/24

Q(P)

```
class Environment:
    def __init__(self):
        self.locations = {'A': 'Dirty', 'B': 'Dirty'}
        self.agent_location = 'A'
        self.agent = VacuumCleanerAgent()

    def step(self):
        self.locations = {'A': 'Dirty', 'B': 'Dirty'}
        self.agent_location = 'A'
        self.agent = VacuumCleanerAgent()
        action = self.agent.act()
```

```
        if action == 'Suck':
            self.locations[location] = 'Clean'
        elif action == 'Move Left':
            self.agent_location = 'A'
        elif action == 'Move Right':
            self.agent_location = 'B'

    def run(self):
        steps = 0

        while True:
            self.step()
            steps += 1

            if all(status == 'Clean' for
                status in self.location.
                                values()):
                print("All rooms are
                        clean")
                break
            if steps >= 10:
                print("Reached maximum
                        steps")
        print("Percept Sequence:")
        for percept in self.agent.
                            percept_sequence:
            print(percept)

if __name__ == "__main__":
    env = Environment()
    env.run()
```

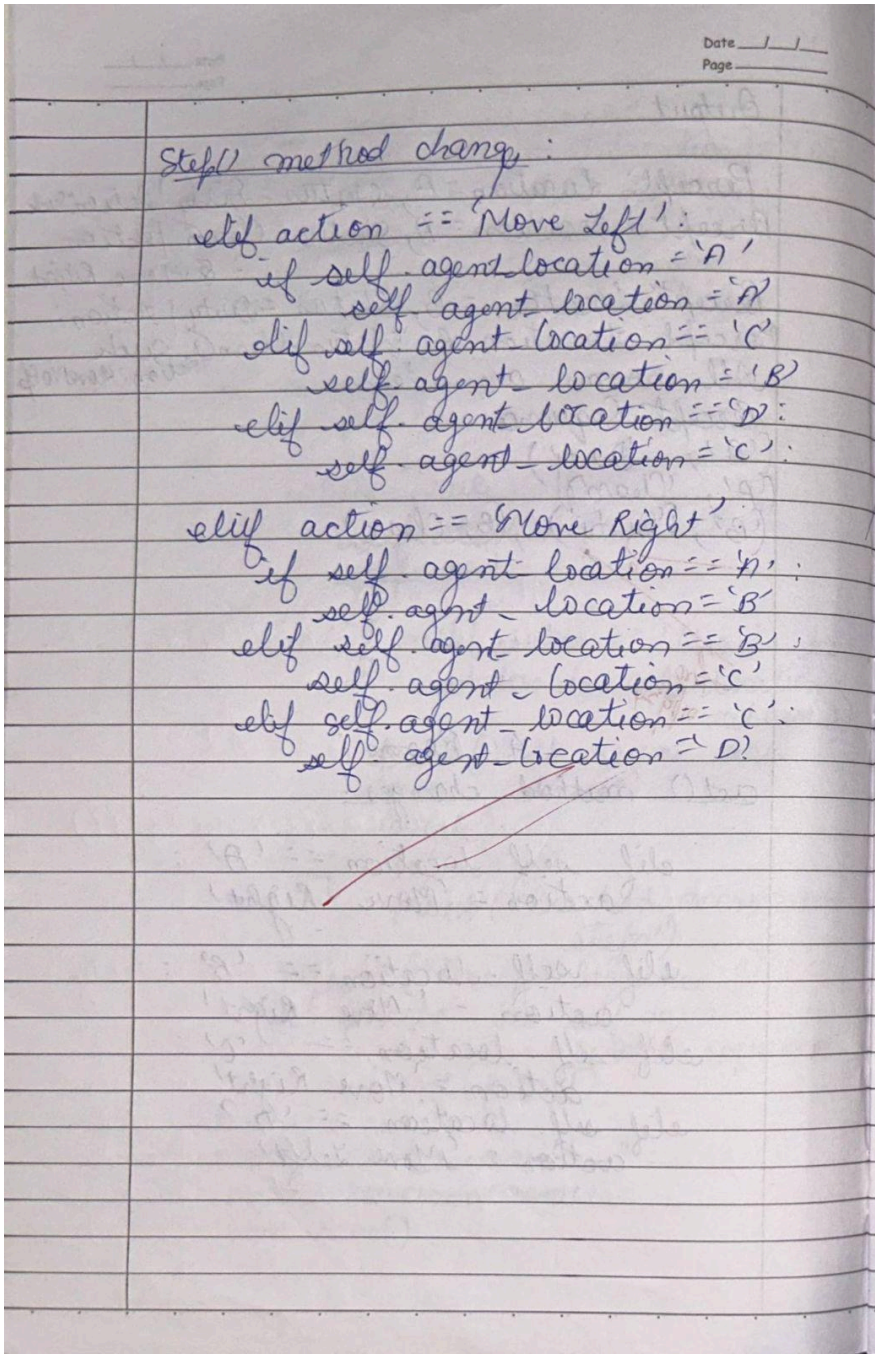
Output:

Percept: Location = A, Status = Dirty | Action: Suck
Percept: Location = A, Status = Clean | Action: Move Right
Percept: Location = B, Status = Dirty | Action: Suck
Percept: Location = B, Status = Clean | Action: NoOp
All rooms are clean
Percept Sequence:
('A', 'Dirty')
('A', 'Clean')
('B', 'Dirty'), ('B', 'Clean')

4 Rooms:

act() method changes:

```
    elif self.location == 'A':
        action = 'Move Right'

    elif self.location == 'B':
        action = 'Move Right'
    elif self.location == 'C':
        action = 'Move Right'
    elif self.location == 'D':
        action = 'Move Left'
```



Codes:

```
def vacuum_cleaner_agent():
    environment = {'A': True, 'B': True}
    position = 'A'
    actions = []
    while True:
        if environment[position]:
            actions.append('Suck')
            environment[position] = False
```

```
        else:
            actions.append('Move')
            position = 'B' if position == 'A' else 'A'
        if not any(environment.values()):
            break
    return actions

print(vacuum_cleaner_agent())
```

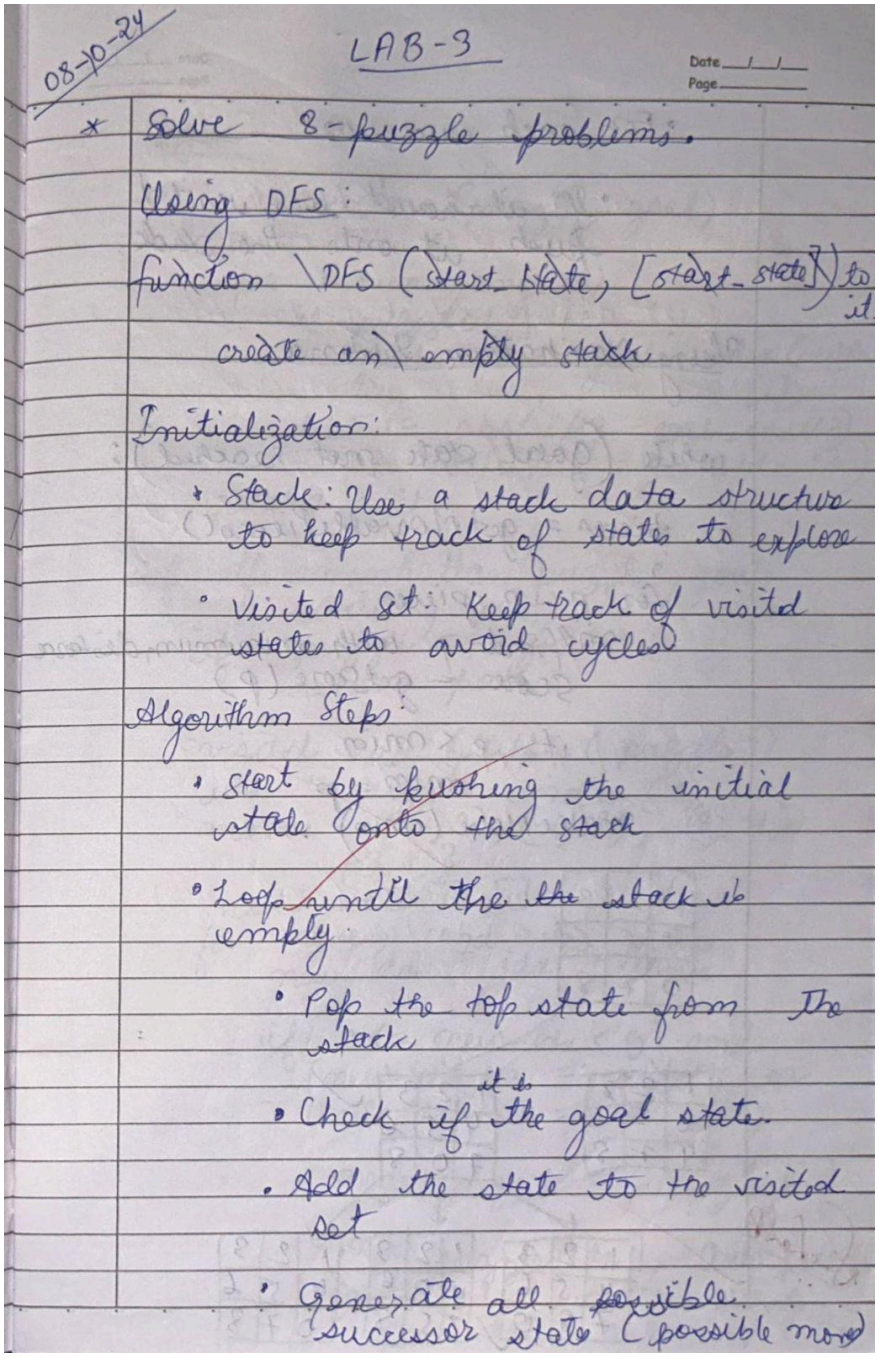
Output:

```
['Suck', 'Move', 'Suck']
```

Lab 3

Implement 8-Puzzle Problem Using Depth-First Search (DFS)

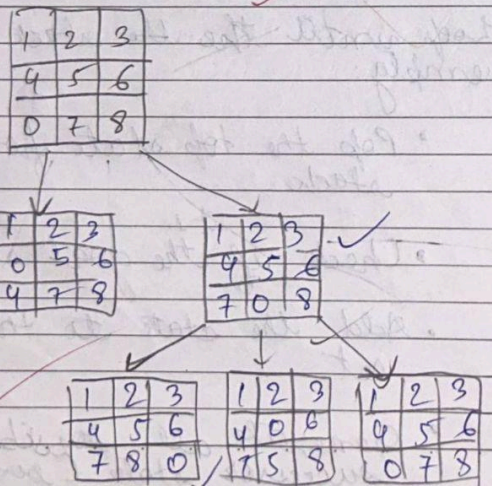
Screenshots:



- For each successor:
  - If it hasn't been visited push it onto the stack.

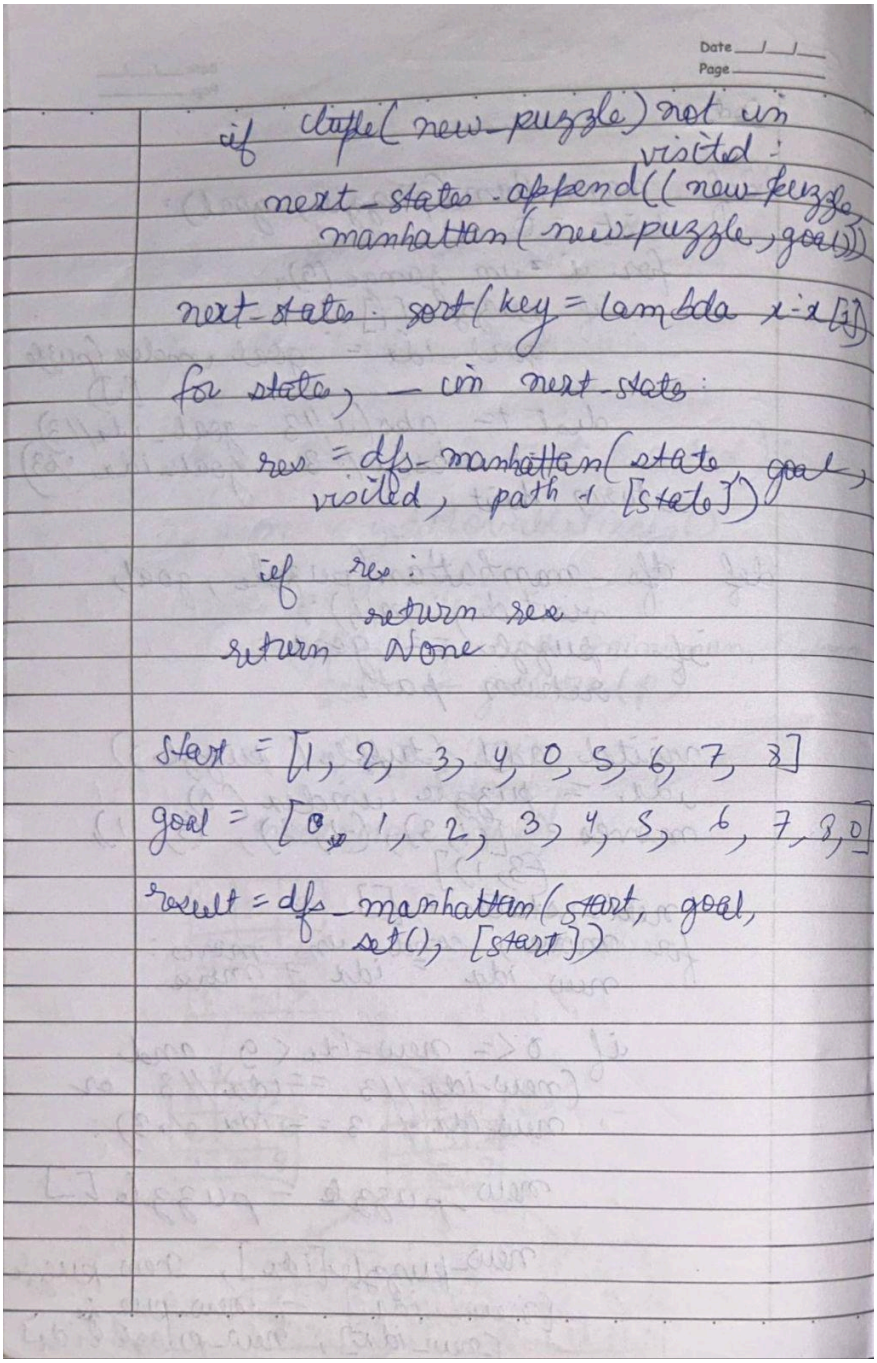
Using Manhattan Distance:

```
while (goal state not reached):
    pieces = getMovablePieces()
    for p in pieces:
        #find p with minimum distance
        score = getScore(p)
        if p < min
            min = p
    makeMove(min)
```



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):
    if puzzle == goal:
        return path

    visited.add(tuple(puzzle))
    idx = puzzle.index(0)
    moves = [(1,3), (-1,3), (3,1), (-3,1)]
    next_states = []
    for move, cond in moves:
        new_idx = idx + move

        if 0<= new_idx < 9 and
            (new_idx//3 == idx//3 or
             new_idx%3 == idx%3):

            new_puzzle = puzzle[:]

            new_puzzle[idx], new_puzzle
            [new_idx] = new_puzzle
            [new_idx], new_puzzle[idx]
```



Codes:

```
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):
    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()

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

            state_tuple = tuple(map(tuple, state))

            if state_tuple in visited:
                continue

            visited.add(state_tuple)

            if self.bhash(state) == self.bhash(self.end):
                for p in path + [state]:
                    self.display(p)
                    print("------")
                return

            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()
```

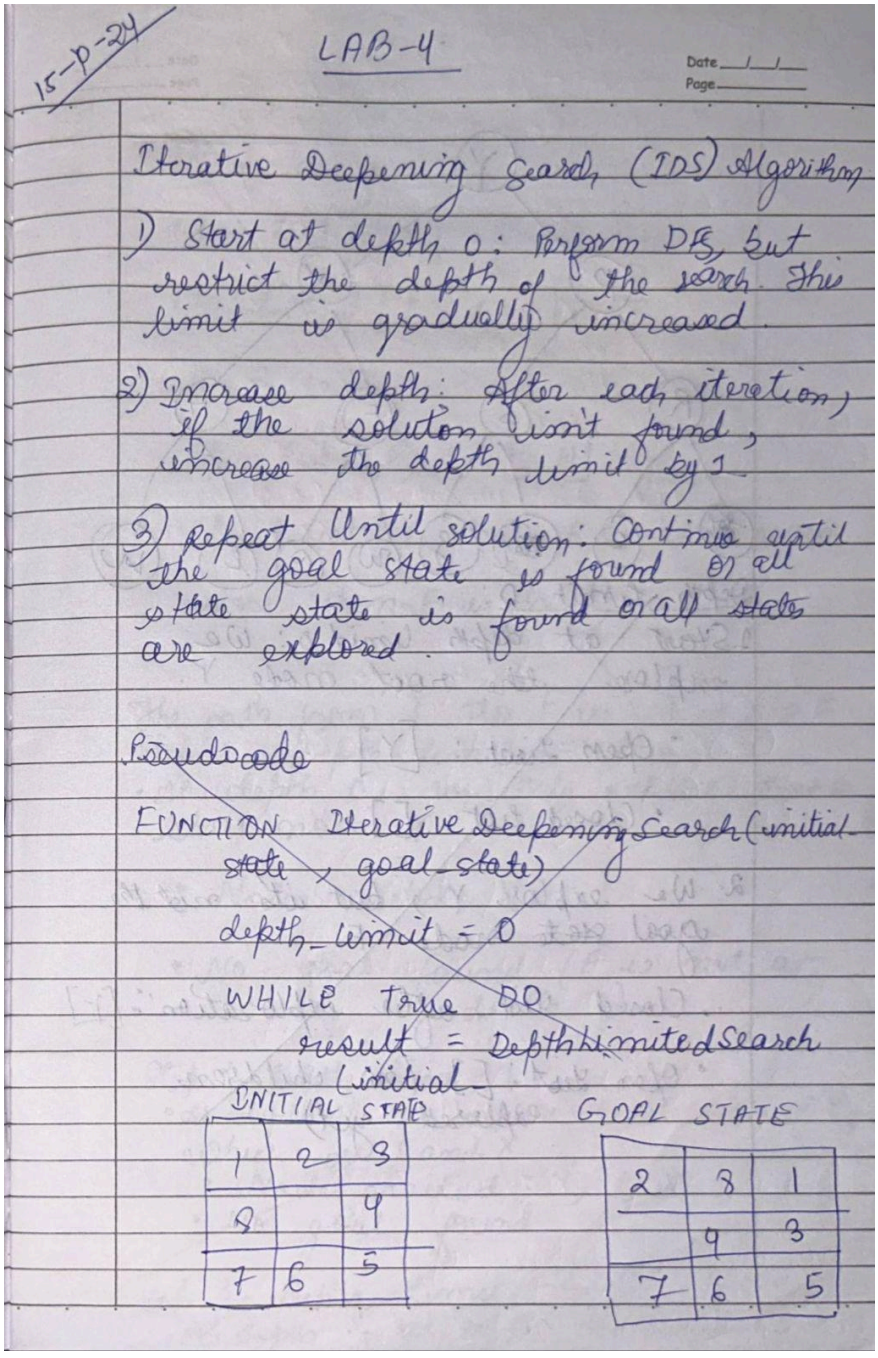
Output:

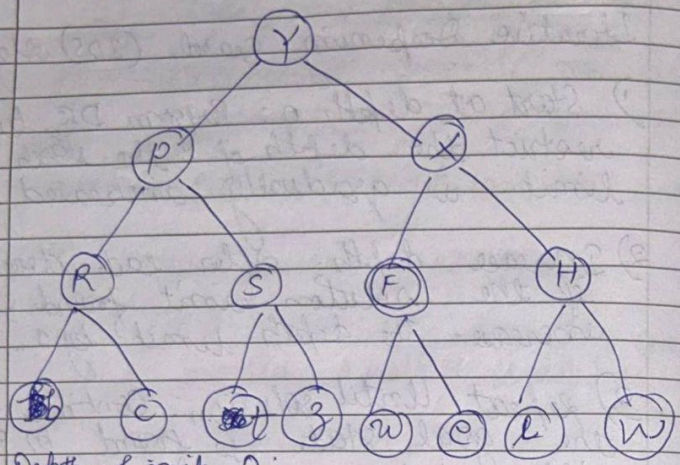
```
8 1 0
2 4 3
7 6 5
------
8 0 1
2 4 3
7 6 5
------
0 8 1
2 4 3
7 6 5
------
2 8 1
0 4 3
7 6 5
------
```

Lab 4

Implement Iterative Deepening Search Algorithm & Implement A Search Algorithm*

Screenshots:





Depth Limit 0:

1. Start at depth limit 0: We explore the root node Y.
   - Open List: [Y]
   - Closed List: []
2. We explore Y, but it's not the goal ~~state~~ node F.
   - Closed list after exploration: [Y]
   - Open List: [] (no children explored yet)

Depth 1:

Open List: [P, X]

Closed List: [P Y, P, X]

Depth 2:

Open List: [R, S, F, H]

Closed List: [Y, P, X, R, S] (the search stops when F is found)

The path from Y to F is: Y → X → F.

Step 1: Depth Limit = 0
- At depth 0, we only explore the start node Y.
  - Nodes visited Y
  - No goal found (F is not at depth 0)

Step 2: Depth Limit = 1
- At depth 1, we explore the children of Y, which are P and X.
  - Nodes visited: Y, P, X
  - No goal found

Step 3: Depth Limit = 2
- At depth 2, we explore the children of P

and X. • P's children: R, S ; • X's children: F, H
Nodes visited: Y, P, R, S, X, F
• Goal state F found at depth 2.
Thus, the solution is found at depth 2, and the path to reach the goal is $Y \rightarrow X \rightarrow F$.

The initial state of the puzzle is:

1	2	3
8		4
7	6	5

Manhattan Distance $h(n)$ for initial state:

Tile 1: Current pos (0,0)
Goal pos (0,2) Distance = 2

Tile 2: Distance = 1

Tile 3: Distance = 1

Tile 4: " 2

Tile 5: " 0

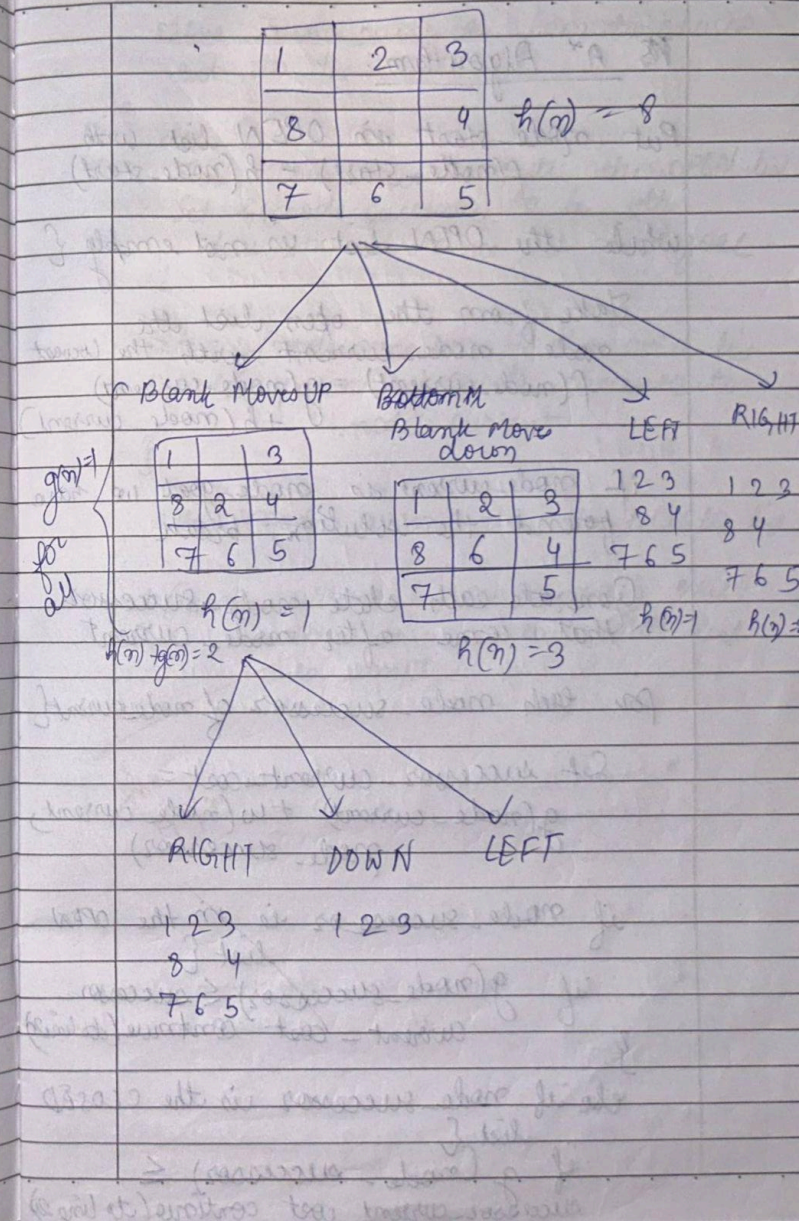
Tile 6: " 0

Tile 7: " 2

Tile 8: " 2

Total Manhattan Distance

$h(n) = 2+1+1+2+0+0+0+2 = 8$



A* Algorithm:

Put node_start in OPEN list with
f(node_start) = h(node_start)

while the OPEN list is not empty {

Take from the open list the node node_current with the lowest f(node_current) = g(node_current) + h(node_current)

if node_current is node_goal we have found the solution; break

Generate each state node_successor that come after node_current

for each node_successor of node_current {

Set successor_current_cost = g(node_current) + w(node_current, node_successor)

if node_successor is in the OPEN list {

if g(node_successor) ≤ successor_current_cost continue (do line 2)

}

else if node_successor is the CLOSED list {

if g(node_successor) ≤ successor_current_cost continue (do line 2)

Move node_successor from the CLOSED list to the OPEN list

} else {

Add node_successor to the OPEN list

Set h(node_successor) to be the heuristic distance to node_goal

}

Set g(node_successor to the OPEN list

Set the parent of node_successor to node_current

}

Add node_current to the CLOSED list

}

if (node_current != node_goal) exit with error (the OPEN list is empty)

Codes:

iterative_deepening.py

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []

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

def iddfs(root, goal):
    for d in range(100000):
        res = dls(root, goal, d)
        if res:
            return "Found"
    return "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')
node2 = TreeNode('P')
node3 = TreeNode('X')
node4 = TreeNode('R')
node5 = TreeNode('S')
node6 = TreeNode('F')
node7 = TreeNode('H')

root.add_child(node2)
root.add_child(node3)
node2.add_child(node4)
node2.add_child(node5)
node3.add_child(node6)
node3.add_child(node7)
```

```
print(iddfs(root, 'F'))
```

a_star_search.py

```
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)
```

Output:

```
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
```

Lab 5

Use Simulated Annealing to Solve the 8-Queens Problem

Screenshots:

22-10-24

LAB-5

Simulated Annealing

algorithm based on the physical annealing process.

The following pseudocode presents the simulated annealing heuristic:

1) Initialise parameters
   - Set the initial solution S
   - Set the initial temperature T
   - Define cooling rate α (0 < α < 1)
   - Set the stopping criterion

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

Acceptance Decision

- If the new state has a lower energy than the previous accept it.
- If the higher energy is found accept it with probability that it is

and have much worse the new state is,

metropolis criteria

$P(\text{accept}) = e^{-[E_{new} - E_{current}]/T}$

1) Cool down: Gradually reduce the temperature according to the cooling schedule

2) Stop: Once the temperature is low or after a set number of iterations, stop and return the best solution found

$T \leftarrow T \times \alpha$

CODE

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

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

```
def simulated_annealing(start, initial_temp, cooling_rate, max_iter):
    current_solution = start
    current_energy = rastrigin(current_solution)
    best_solution = current_solution
    best_energy = current_solution_energy
    temp = initial_temp

    energies = [current_energy]

    for i in range(max_iter):

        candidate_solution = current_solution
            + np.random.uniform(-1, 1,
                size = len(start))

        candidate_solution = np.clip(
            candidate_solution, -5.12, 5.12)

        candidate_energy = rastrigin(candidate
                _solution)

        delta_energy = candidate_energy -
                current_energy

        if delta_energ > 0:
            current_solution = candidate
                Solution
            current_energy = candidate_energy
        else:
            acceptance_prob = np.exp(delta_energ/
                temp)
            if np.random.rand() < acceptance_pro
                current_solution = candidate_soluti
                best_energy = current_energy

        temp *= cooling_rate
        energies.append(current_energy)

    return best_solution, best_energy,
            energies
```

Output:

Best solution : [-4.549..., 4.519...]

Best energy : 80.642...

Jm 22/10/24

Code:

```
import random
import math


class Annealing:
    def __init__(self) -> None:
        self.initial_sol = random.uniform(-10, 10)
        self.temp = 10
        self.cooling = 0.99
        self.final = 0.01
        self.annealing()
    def cost(self, x):
        # return x**2
        return x**4 + 5*math.sin(5*math.pi*x)
    def getNeighbors(self, sol):
        return sol + random.uniform(-1, 1)
    def annealing(self):
        current = self.initial_sol
        new = current
        best = current
        while self.temp > self.final:
            new = self.getNeighbors(current)
            dE = self.cost(new) - self.cost(current)
            print(f"Temp diff: {((self.temp - self.final)*100)/self.temp:.2f}%; Current sol:
{current}; New sol: {new}; Best: {best}")
            if dE < 0 or random.random() < math.exp(-dE/self.temp):
                current = new
            if self.cost(new) < self.cost(best):
                best = new
            self.temp *= self.cooling
        print(f"Final solution: {best}")


c = Annealing()
```

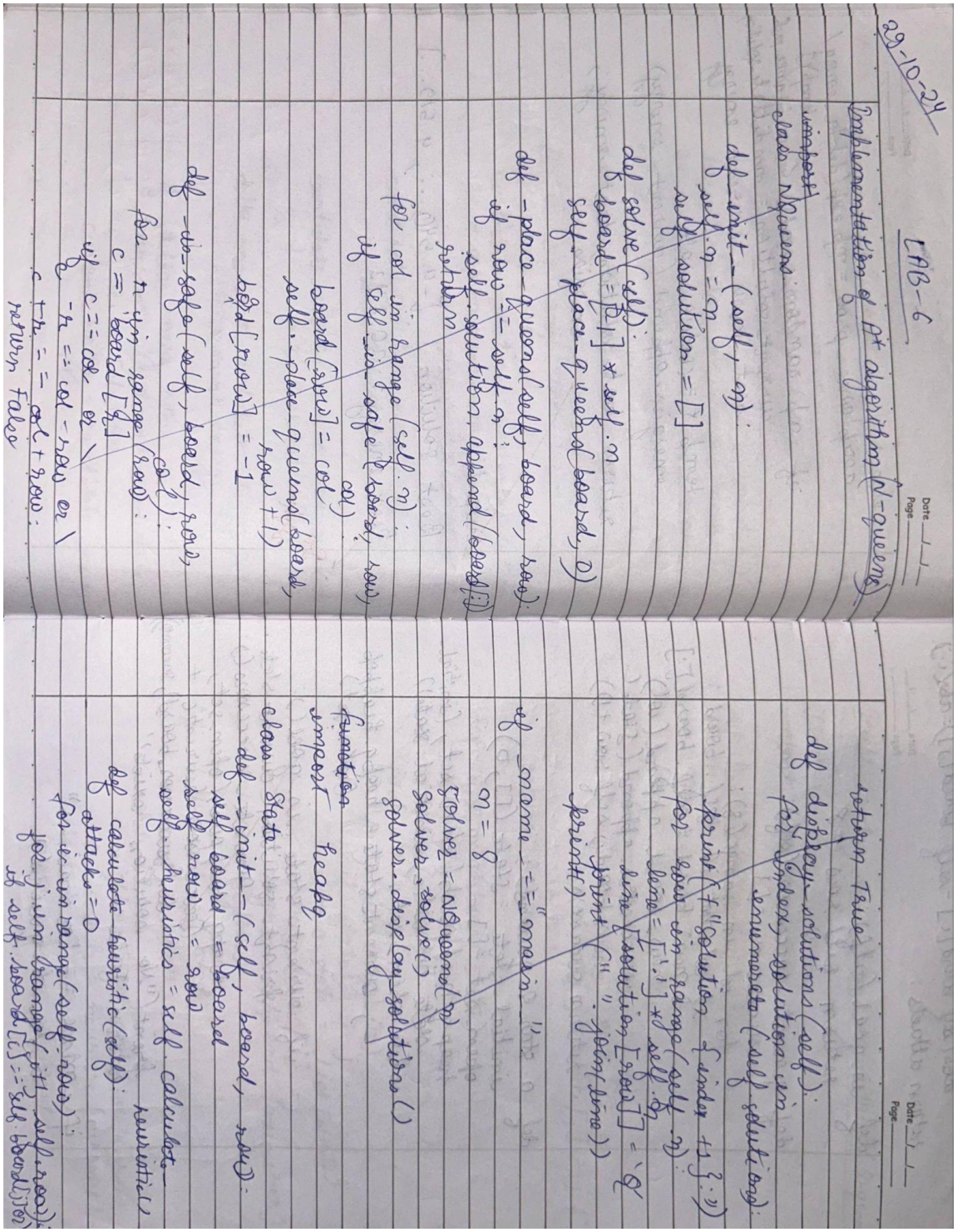
Output:

```
Temp diff: 8.02%; Current sol: -0.10524791597666172; New sol: -0.5199592973272358; Best: -0.10044379288794336
Temp diff: 7.09%; Current sol: -0.10524791597666172; New sol: -0.04562821938109707; Best: -0.10044379288794336
Temp diff: 6.15%; Current sol: -0.10524791597666172; New sol: -0.6717496506340699; Best: -0.10044379288794336
Temp diff: 5.20%; Current sol: -0.10524791597666172; New sol: -0.08008245645276957; Best: -0.10044379288794336
Temp diff: 4.24%; Current sol: -0.10524791597666172; New sol: -0.0734785559577622; Best: -0.10044379288794336
Temp diff: 3.28%; Current sol: -0.10524791597666172; New sol: 0.3864540085802399; Best: -0.10044379288794336
Temp diff: 2.30%; Current sol: -0.10524791597666172; New sol: 0.14231659835465837; Best: -0.10044379288794336
Temp diff: 1.31%; Current sol: -0.10524791597666172; New sol: -0.5267249842132364; Best: -0.10044379288794336
Temp diff: 0.32%; Current sol: -0.10524791597666172; New sol: -0.8089796006183425; Best: -0.10044379288794336
Final solution: -0.10044379288794336
```

Lab 6

Implement Hill Climbing Search Algorithm to Solve the N-Queens Problem

Screenshots:



```
            abs(self.board[i] - self.board[j]) == abs(i-j):
    return attacks:

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

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

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

        if current_state.is_goal():
            print_solution(current_state.
                        generate_successors():
                heapq.heappush(open_set,
                (successor.heuristic +
                    len(successor.board), successor))
    print("No solution exists")
    return False

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

Implementing hill climbing method (8-queen

CODE:-

```
import random

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

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

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

12-11-24

LAB - 07

Date __/__/__
Page

Knowledge Base:

1. Alice is the mother of Bob.
2. Bob is the 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 siblings.

Hypothesis:

- "Charlie is a sibling of Bob"

Entailment Process:

- Bob is a parent (from premise 2 and 3)
- Since Bob is a parent, his children (including Charlie) are siblings (from premise 6)
- Therefore, Charlie and Bob are siblings

Conclusions:

- The hypothesis is entailed by the knowledge base.

Code:

```
import math
import random
def cost(x):
   return math.sin(x)

def hill_climbing(initial_sol=0, steps=0.01, max_steps=1000):
   print(f"Initial sol: {initial_sol}; steps: {steps}")
   current = initial_sol
   best = current
   for _ in range(max_steps):
       neighbor = [current + steps, current - steps]
       # print(neighbor)
       neighbor.sort(key=lambda x : cost(x))
       current = neighbor[-1]
       if cost(best) < cost(current):
           best = current
       else:
           break
       print(f"Current: {current}, Best: {best}")
   return best
initial_sol = random.uniform(-10, 10)
steps = random.choice((00.01, 0.001))
print(hill_climbing(initial_sol, steps))
```

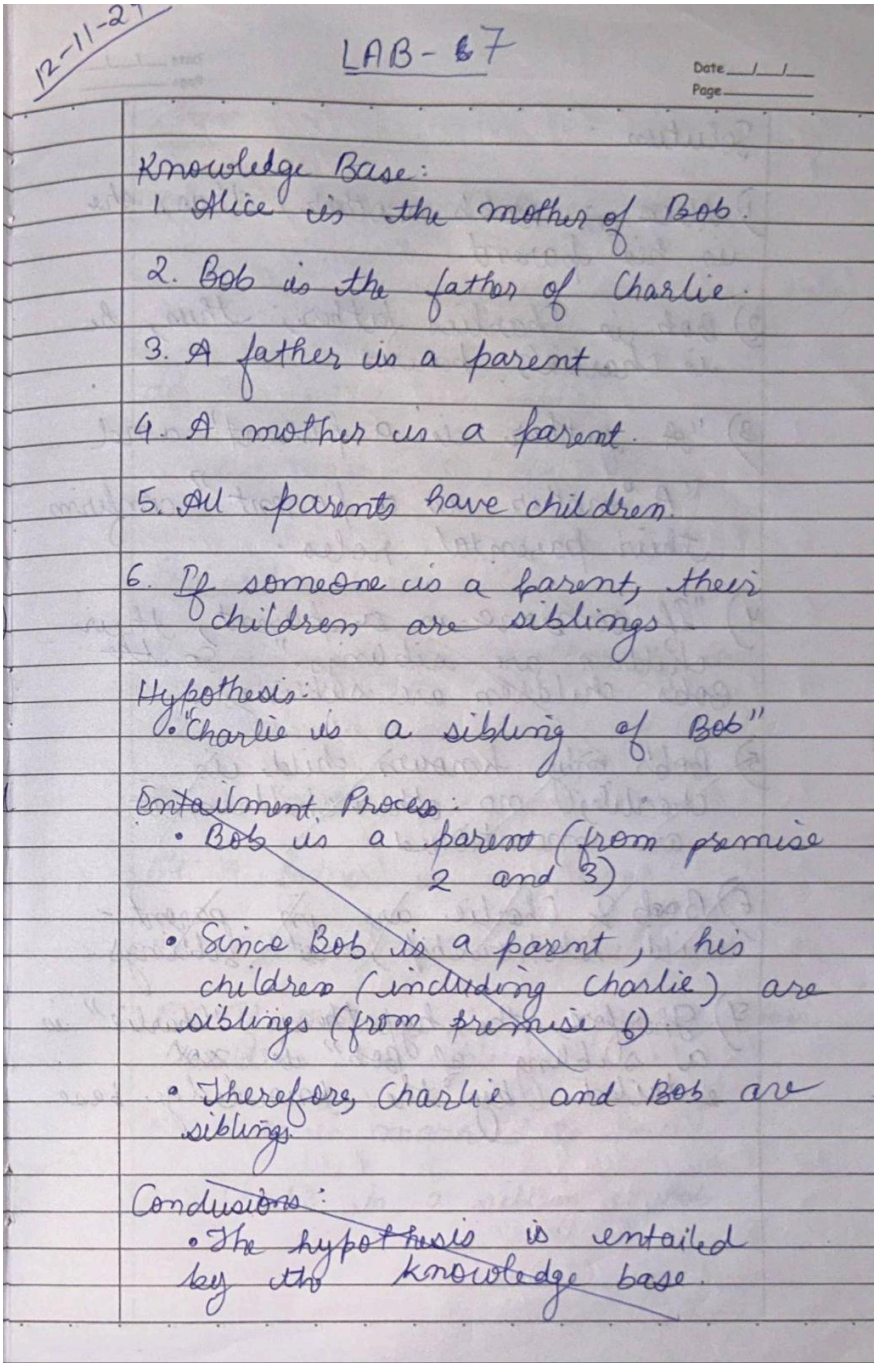
Output:

```
Current: 1.5099927602763248, Best: 1.5099927602763248
Current: 1.5199927602763248, Best: 1.5199927602763248
Current: 1.5299927602763248, Best: 1.5299927602763248
Current: 1.5399927602763248, Best: 1.5399927602763248
Current: 1.5499927602763248, Best: 1.5499927602763248
Current: 1.5599927602763248, Best: 1.5599927602763248
Current: 1.5699927602763248, Best: 1.5699927602763248
1.5699927602763248
```

Lab 7

Create a Knowledge Base Using Propositional Logic and Test Query Entailment

Screenshots:



Solution:

1) Alice is Bob's mother; thus, she is his parent.

2) Bob is Charlie's father; thus, he is Charlie's parent.

3) "A father is a parent" and "A mother is a parent" confirm their parental roles.

4) "If someone is a parent, their children are siblings." So the Bob's children are siblings.

5) Bob's only known child is Charlie; no other children are mentioned.

6) Bob & Charlie are in parent-child relationship, not siblings.

7) Therefore, the hypothesis "Charlie" is a sibling of "Bob" is not entailed by the knowledge base.

CODE:

```
def pl_true(statement, model):
    return model.get(statement, False)

def tt_entails(kb, alpha):
    symbols = list(set(kb
```

Propositional Variables

P1: Alice is the mother of Bob

P2: Bob is the father of Charlie

P3: A mother is a parent.

P4: A father is a parent.

P5: All parents have children.

P6: If someone is a parent, their children are siblings

P7: Alice is married to David.

H: Charlie is a sibling of Bob.

Step	Logical Expression	Conclusion
1.	P1	Alice is the mother of Bob.
2.	P3	A mother is a parent
3.	P1 ∧ P3 → P8	Alice is a parent (P8)
4.	P2	Bob is the father of Charlie.
5.	P4	A father is a parent
6.	P2 ∧ P4 → P9	Bob is a parent (P9)
7.	P5	All parents have children.
8.	P8 ∧ P5 → P10	Alice has children (P10)
9.	P9 ∧ P5 → P11	Bob has children (P11)
10.	P6	Parents children are siblings
11.	P9 ∧ P6 → (Bob's children are siblings)	From Bob being a parent.
12	[From step 4 & P11] Bob only known child is Charlie.	& No other children are specified.
13.	Cannot conclude Charlie has siblings via Bob	
14.	P8 ∧ P6 → (Alice's children are siblings)	From Alice being a parent
15.	[From step 2 & P10] Alice's only known child is Bob	No other children are specified. Cannot

Hence Proved.

Code:

```
def AND(a, b):
    return a and b

def OR(a, b):
    return a or b

def NOT(a):
    return not a

def IMPLIES(a, b):
    return (not a) or b

def infix_to_postfix(expr):
    precedence = {'NOT': 3, 'AND': 2, 'OR': 2, 'IMPLIES': 1}
    output = []
    stack = []
    tokens = expr.replace('(', ' ( ').replace(')', ' ) ').split()

    for token in tokens:
        if token not in precedence and token not in {'(', ')'}:
            output.append(token)
        elif token in precedence:
            while stack and stack[-1] != '(' and precedence[stack[-1]] >= precedence[token]:
                output.append(stack.pop())
            stack.append(token)
        elif token == '(':
            stack.append(token)
        elif token == ')':
            while stack and stack[-1] != '(':
                output.append(stack.pop())
            stack.pop()
    while stack:
        output.append(stack.pop())

    return ' '.join(output)

def eval_postfix(postfix_expr, assignment):
    tokens = postfix_expr.split()
    stack = []
    for token in tokens:
        if token == 'NOT':
```

```
            a = stack.pop()
            stack.append(NOT(a))
        elif token in {'AND', 'OR', 'IMPLIES'}:
            b = stack.pop()
            a = stack.pop()
            if token == 'AND':
                stack.append(AND(a, b))
            elif token == 'OR':
                stack.append(OR(a, b))
            elif token == 'IMPLIES':
                stack.append(IMPLIES(a, b))
        else:
            stack.append(assignment.get(token, False))
    return stack.pop()

def generate_assignments(propositions):
    n = len(propositions)
    for i in range(2**n):
        assignment = {}
        for j, prop in enumerate(propositions):
            assignment[prop] = (i & (1 << j)) != 0
        yield assignment

def entails(kb, query, propositions):
    kb_postfix = [infix_to_postfix(sentence) for sentence in kb]
    query_postfix = infix_to_postfix(query)

    for assignment in generate_assignments(propositions):
        kb_true = all(eval_postfix(sentence, assignment) for sentence in kb_postfix)
        if kb_true and not eval_postfix(query_postfix, assignment):
            return False
    return True

if __name__ == "__main__":
    knowledge_base = [
        "A IMPLIES B",
        "B IMPLIES C",
        "A"
    ]
    query = "C"
    propositions = set()
    for sentence in knowledge_base + [query]:
```

```
        tokens = sentence.replace('(', ' ').replace(')', ' ').split()
        for token in tokens:
            if token not in {'AND', 'OR', 'NOT', 'IMPLIES'} and token.strip():
                propositions.add(token)
    result = entails(knowledge_base, query, sorted(propositions))
    print("KB entails Query:", "Yes" if result else "No")
```

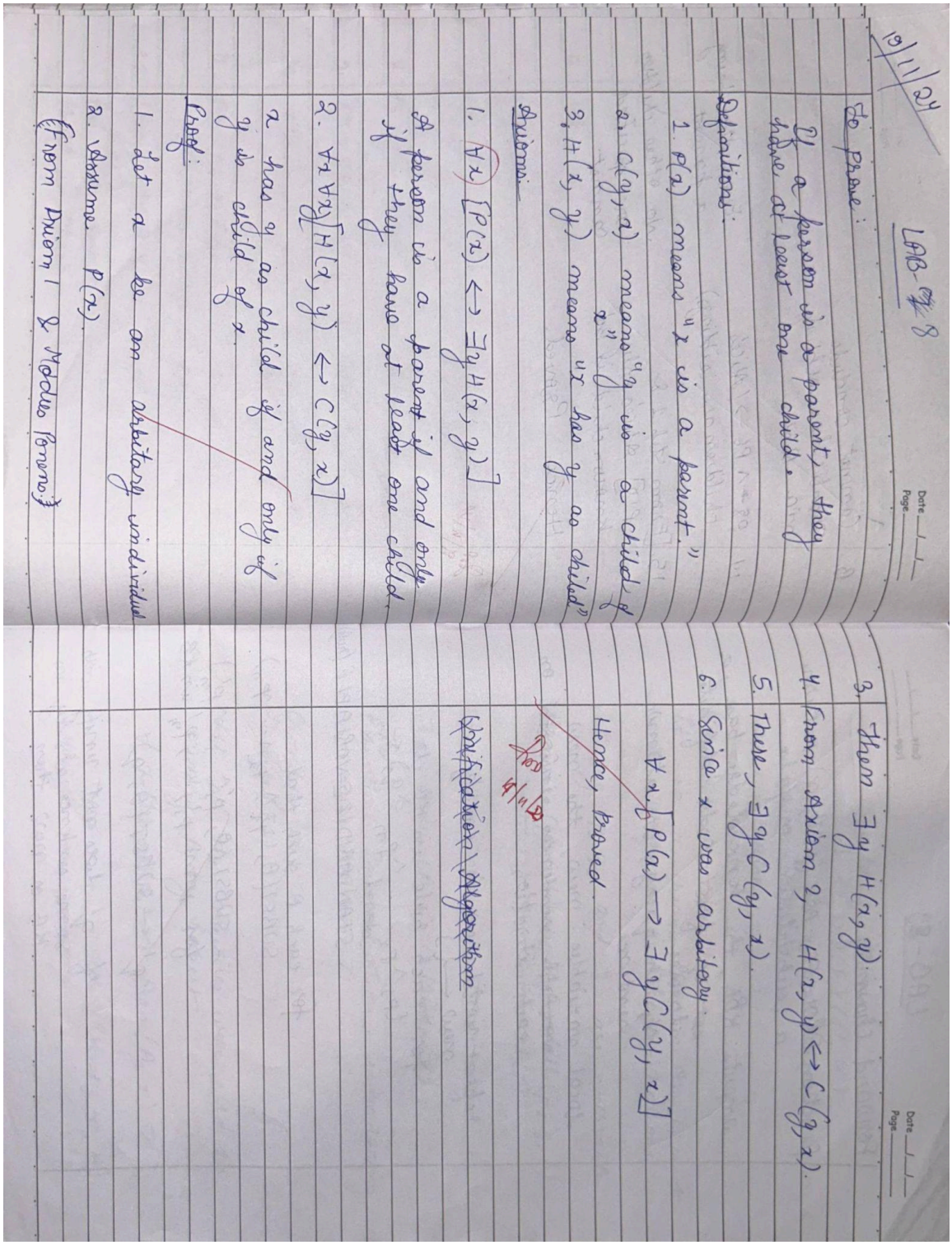
Output:

```
KB entails Query: Yes
```

Lab 8

Implement Unification in First-Order Logic

Screenshots:



Code:

```
def parse_predicate(predicate):
    name, args = predicate.split('(')
    args = args.rstrip(')').split(', ')
    return name, args

def is_variable(term):
    return term[0].islower()

def unify(x, y, subst={}):
    name1, args1 = parse_predicate(x)
    name2, args2 = parse_predicate(y)
    if name1 != name2 or len(args1) != len(args2):
        return None
    for a, b in zip(args1, args2):
        subst = unify_terms(a, b, subst)
        if subst is None:
            return None
    return subst

def unify_terms(a, b, subst):
    a = apply_substitution(a, subst)
    b = apply_substitution(b, subst)
    if a == b:
        return subst
    if is_variable(a):
        return extend_subst(a, b, subst)
    if is_variable(b):
        return extend_subst(b, a, subst)
    return None

def apply_substitution(term, subst):
    while is_variable(term) and term in subst:
        term = subst[term]
    return term

def extend_subst(var, value, subst):
    if occurs_check(var, value, subst):
        return None
    subst = subst.copy()
    subst[var] = value
```

```
    return subst

def occurs_check(var, value, subst):
    if var == value:
        return True
    if is_variable(value) and value in subst:
        return occurs_check(var, subst[value], subst)
    return False

predicate1 = "Parent(x, y)"
predicate2 = "Parent(John, Mary)"
substitution = unify(predicate1, predicate2)

if substitution is not None:
    print("Unification Successful!")
    print("Substitution:", substitution)
else:
    print("Unification Failed.")
```

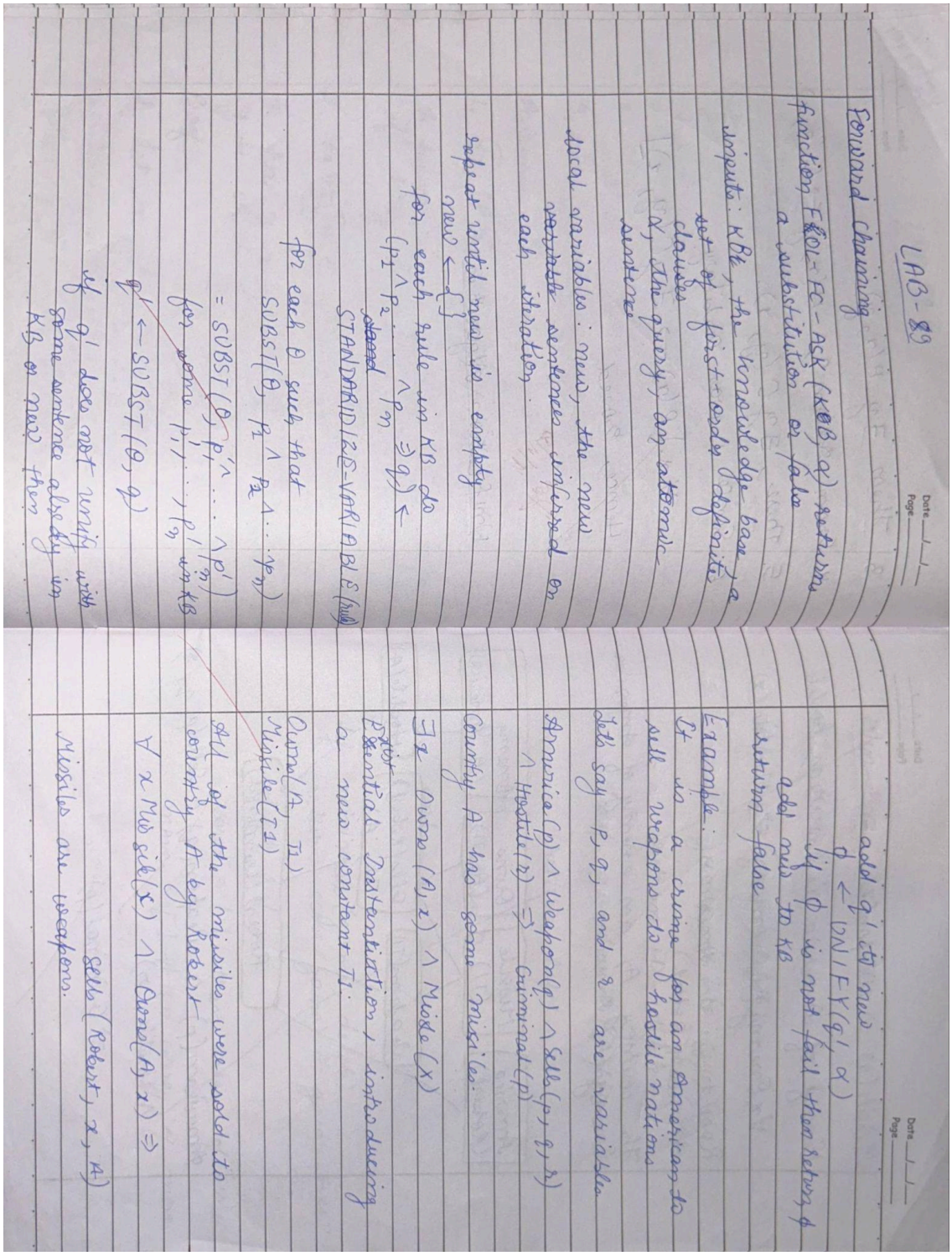
Output:

```
Unification Successful!
Substitution: {'x': 'John', 'y': 'Mary'}
```

Lab 9

Use Forward Reasoning to Prove a Query from a First-Order Logic Knowledge Base

Screenshots:



Missile(x) ⇒ Weapon(x)

Enemy of America is known as hostile

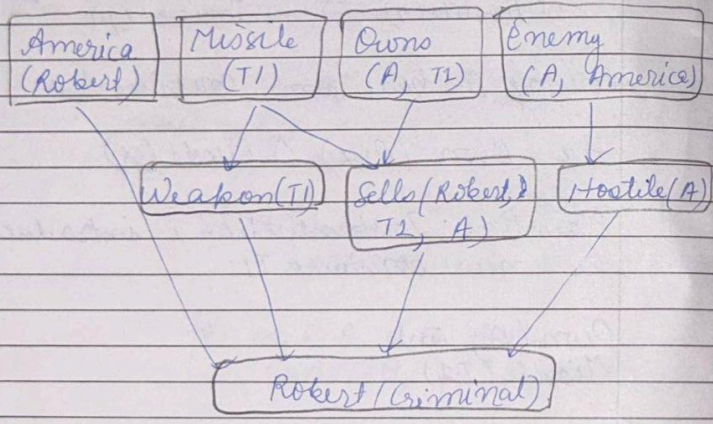
∀x Enemy(x, America) ⇒ Hostile(x)

Robert is an American

American(Robert)

The country A, an enemy of America

Enemy(A, America)



American(p) ∧ Weapon(q) ∧ Sells(p, q, r)

∧ Hostile(r)

⇒ Criminal(p)

Min-Max

Algorithm:

```
def score(game, depth)
    if game.win?(@player)
        return 10 - depth
    elsif game.win?(@opponent)
        return depth - 10
    else
        return 0
    end
end

def minimax(game, depth)
    return score(game) if game.over?
    depth += 1
    scores = [] # an array of scores
    moves = [] # an array of moves
```

Code:

```
def forward_chaining_fol(kb, facts, query):
    inferred = set(facts)
    while True:
        new_inferences = set()
        for premises, conclusion in kb:
            if premises.issubset(inferred) and conclusion not in inferred:
                new_inferences.add(conclusion)
        if query in new_inferences:
            return True
        if not new_inferences:
            break
        inferred.update(new_inferences)
    return query in inferred

knowledge_base = [
    ({"Human(John)"}, "Mortal(John)"),
    ({"Parent(John, Mary)", "Human(Mary)"}, "Human(John)"),
    ({"Father(John, Mary)"}, "Parent(John, Mary)"),
    ({"Human(Mary)"}, "Human(John)")
]

facts = {"Father(John, Mary)", "Human(Mary)"}
query = "Mortal(John)"

result = forward_chaining_fol(knowledge_base, facts, query)
print("Query Result:", "Proved" if result else "Not Proved")
```

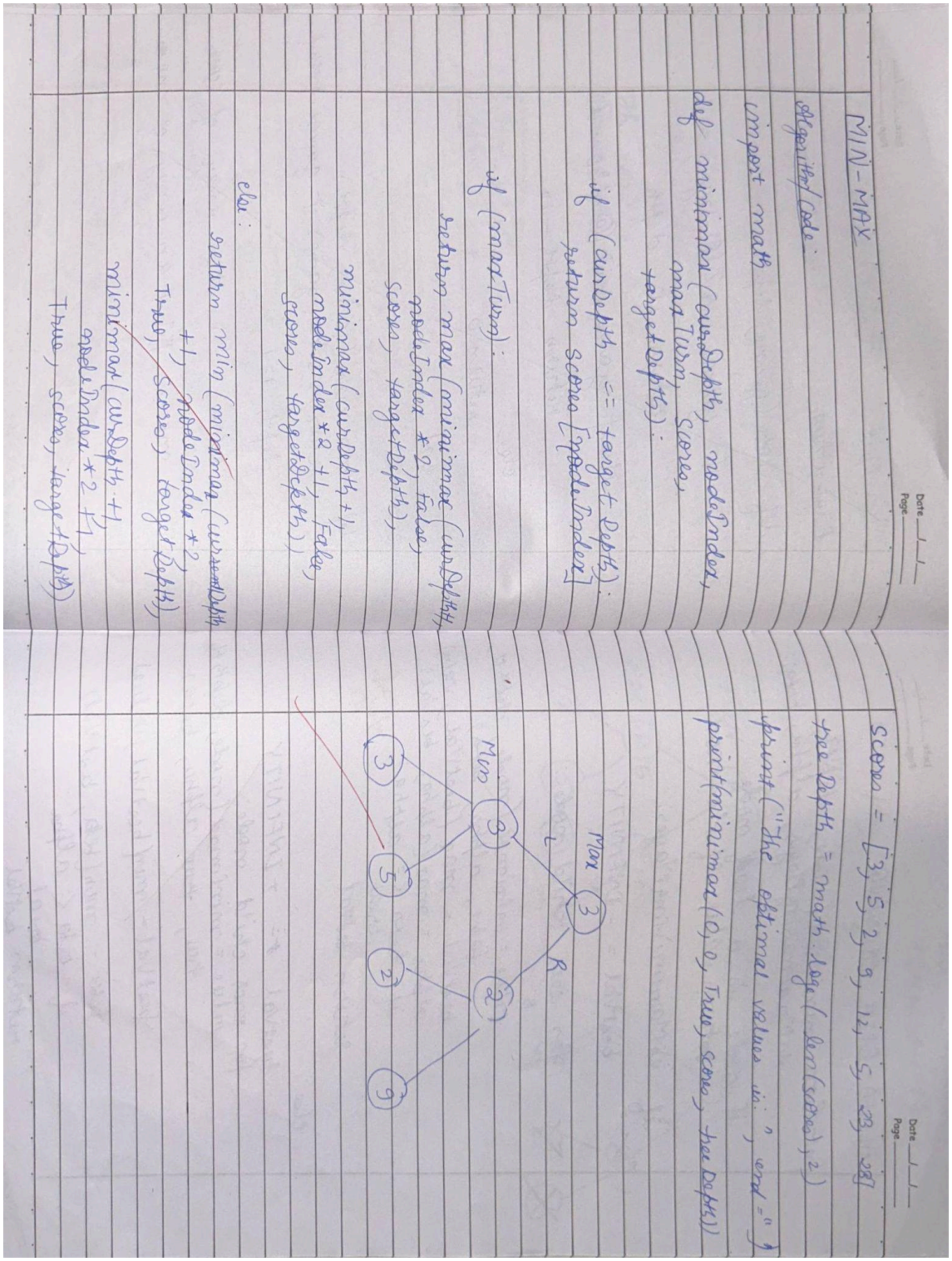
Output:

```
Query Result: Proved
```

Lab 10

Implement Alpha-Beta Pruning

Screenshots:



Date / /
Page

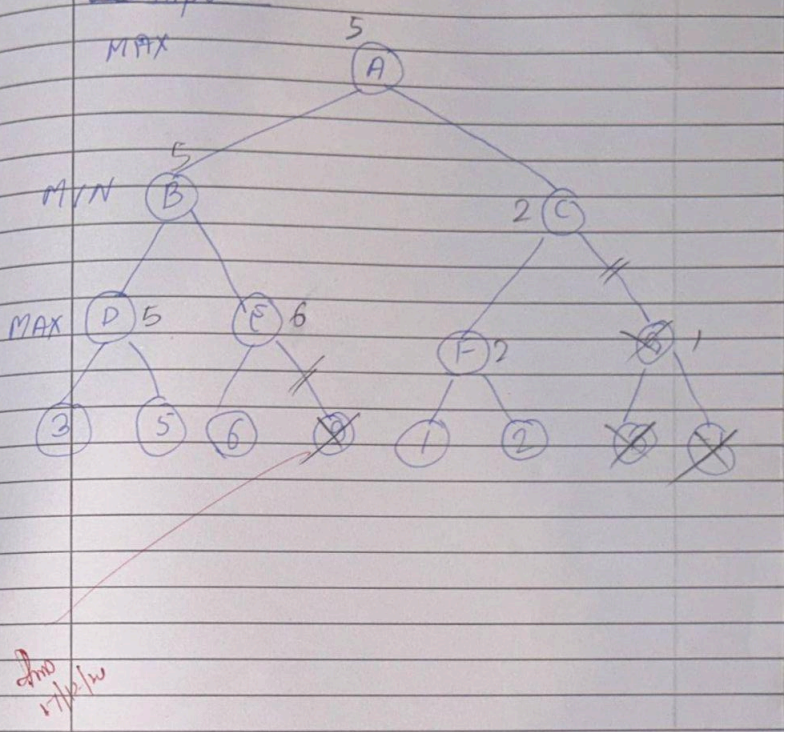
ALPHA-BETA PRUNING

```
function minimax(node, depth,
        isMaximisingPlayer, alpha, beta):

  if node is a leaf node:
      return value of the node

  if isMaximisingPlayer:
      bestVal = -INFINITY

      for each child node:
          value = minimax(node, depth+1,
                  false, alpha, beta)
          bestVal = max(bestVal, value)
          alpha = max(alpha, bestVal)
          if beta <= alpha:
              break
      return bestVal

  else:
      bestVal = +INFINITY
      for each child node:
          value = minimax(node, depth+1,
                  true, alpha, beta)
          bestVal = min(bestVal, value)
          beta = min(beta, bestVal)
          if beta <= alpha:
              break
      return bestVal
```

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

Date / /
Page

Example:



Code:

minimax.py

```
def minimax(board, depth, is_maximizing):
    winner = check_winner(board)
    if winner == "X":
        return -10
    if winner == "O":
        return 10
    if " " not in board:
        return 0

    if is_maximizing:
        best_score = -float('inf')
        for i in range(9):
            if board[i] == " ":
                board[i] = "O"
                score = minimax(board, depth + 1, False)
                board[i] = " "
                best_score = max(best_score, score)
        return best_score
    else:
        best_score = float('inf')
        for i in range(9):
            if board[i] == " ":
                board[i] = "X"
                score = minimax(board, depth + 1, True)
                board[i] = " "
                best_score = min(best_score, score)
        return best_score

def check_winner(board):
    for i in range(3):
        if board[i*3] == board[i*3+1] == board[i*3+2] != " ":
            return board[i*3]
        if board[i] == board[i+3] == board[i+6] != " ":
            return board[i]
    if board[0] == board[4] == board[8] != " " or board[2] == board[4] == board[6] != " ":
        return board[4]
    return None

def best_move(board):
    best_score = -float('inf')
    move = -1
```

```
    for i in range(9):
        if board[i] == " ":
            board[i] = "O"
            score = minimax(board, 0, False)
            board[i] = " "
            if score > best_score:
                best_score = score
                move = i
    return move

# Example usage:
board = [" " for _ in range(9)]
board[0], board[4], board[8] = "X", "O", "X"  # Sample state
move = best_move(board)
print("Best move for O:", move)
```

alpha_beta_pruning.py

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

def alpha_beta(board, row, alpha, beta, is_maximizing):
    if row == len(board):
        return 1

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

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

def solve_8_queens():
    board = [-1] * 8
    alpha = -float('inf')
    beta = float('inf')
    return alpha_beta(board, 0, alpha, beta, True)

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

Output:

```
Number of solutions for the 8 Queens problem: 6
```