VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Vignesh B (1BM22CS326)

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 **Vignesh B (1BM22CS326),** 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.

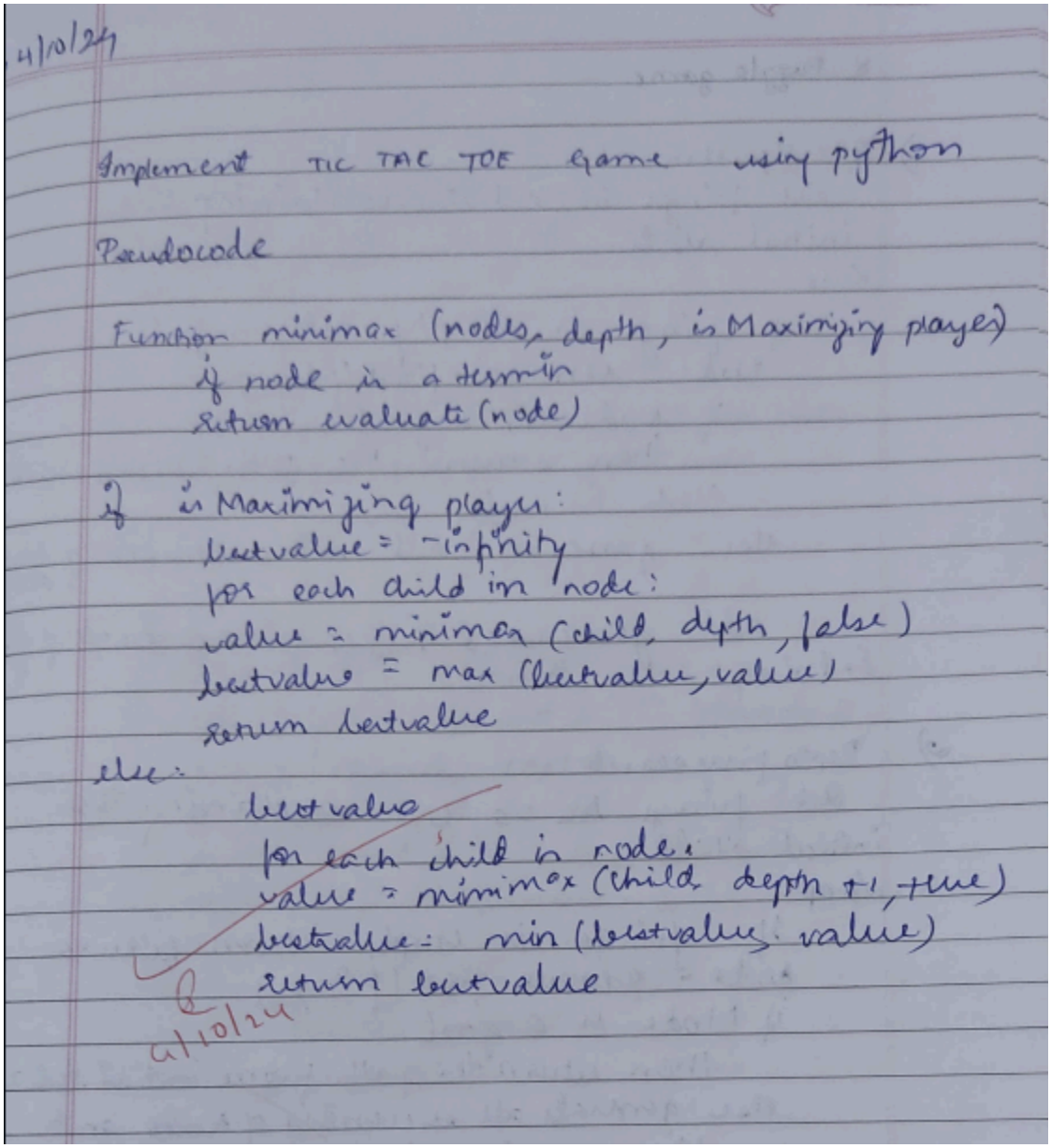
Dr. Rashmi H	Dr. Kavitha Sooda
Assistant Professor	Professor & HOD
Department of CSE, BMSCE	Department of CSE, BMSCE

Index

Github Link:
AI Lab Github Link

1. a) Implement Tic-Tac-Toe Game.

Algorithm:



Code:

```
board = {1: ' ', 2: ' ', 3: ' ',
        4: ' ', 5: ' ', 6: ' ',
        7: ' ', 8: ' ', 9: ' '}

output_printed = False

def printBoard(board):
```

```
    global output_printed
    if not output_printed:
        print('Output: 1BM22CS290')
        output_printed = True
    print(board[1] + '|' + board[2] + '|' + board[3])
    print('-+-+-')
    print(board[4] + '|' + board[5] + '|' + board[6])
    print('-+-+-')
    print(board[7] + '|' + board[8] + '|' + board[9])
    print('\n')

def spaceFree(pos):
    return board[pos] == ' '

def checkWin():
    win_conditions = [(1, 2, 3), (4, 5, 6), (7, 8, 9),  # Horizontal
                      (1, 4, 7), (2, 5, 8), (3, 6, 9),  # Vertical
                      (1, 5, 9), (3, 5, 7)]             # Diagonal
    for a, b, c in win_conditions:
        if board[a] == board[b] == board[c] and board[a] != ' ':
            return True
    return False

def checkMoveForWin(move):
    win_conditions = [(1, 2, 3), (4, 5, 6), (7, 8, 9),  # Horizontal
                      (1, 4, 7), (2, 5, 8), (3, 6, 9),  # Vertical
                      (1, 5, 9), (3, 5, 7)]             # Diagonal
    for a, b, c in win_conditions:
        if board[a] == board[b] == board[c] and board[a] == move:
            return True
    return False

def checkDraw():
    return all(board[key] != ' ' for key in board.keys())

def insertLetter(letter, position):
    if spaceFree(position):
        board[position] = letter
        printBoard(board)

        if checkWin():
            if letter == 'X':
                print('Bot wins!')
            else:
                print('You win!')
            return True
```

```
        elif checkDraw():
            print('Draw!')
            return True
    else:
        print('Position taken, please pick a different position.')
        position = int(input('Enter new position: '))
        return insertLetter(letter, position)

    return False

player = 'O'
bot = 'X'

def playerMove():
    position = int(input('Enter position for O: '))
    return insertLetter(player, position)

def compMove():
    bestScore = -1000
    bestMove = 0
    for key in board.keys():
        if board[key] == ' ':
            board[key] = bot
            score = minimax(board, False)
            board[key] = ' '
            if score > bestScore:
                bestScore = score
                bestMove = key

    return insertLetter(bot, bestMove)

def minimax(board, isMaximizing):
    if checkMoveForWin(bot):
        return 1
    elif checkMoveForWin(player):
        return -1
    elif checkDraw():
        return 0

    if isMaximizing:
        bestScore = -1000
        for key in board.keys():
            if board[key] == ' ':
                board[key] = bot
                score = minimax(board, False)
                board[key] = ' '
```

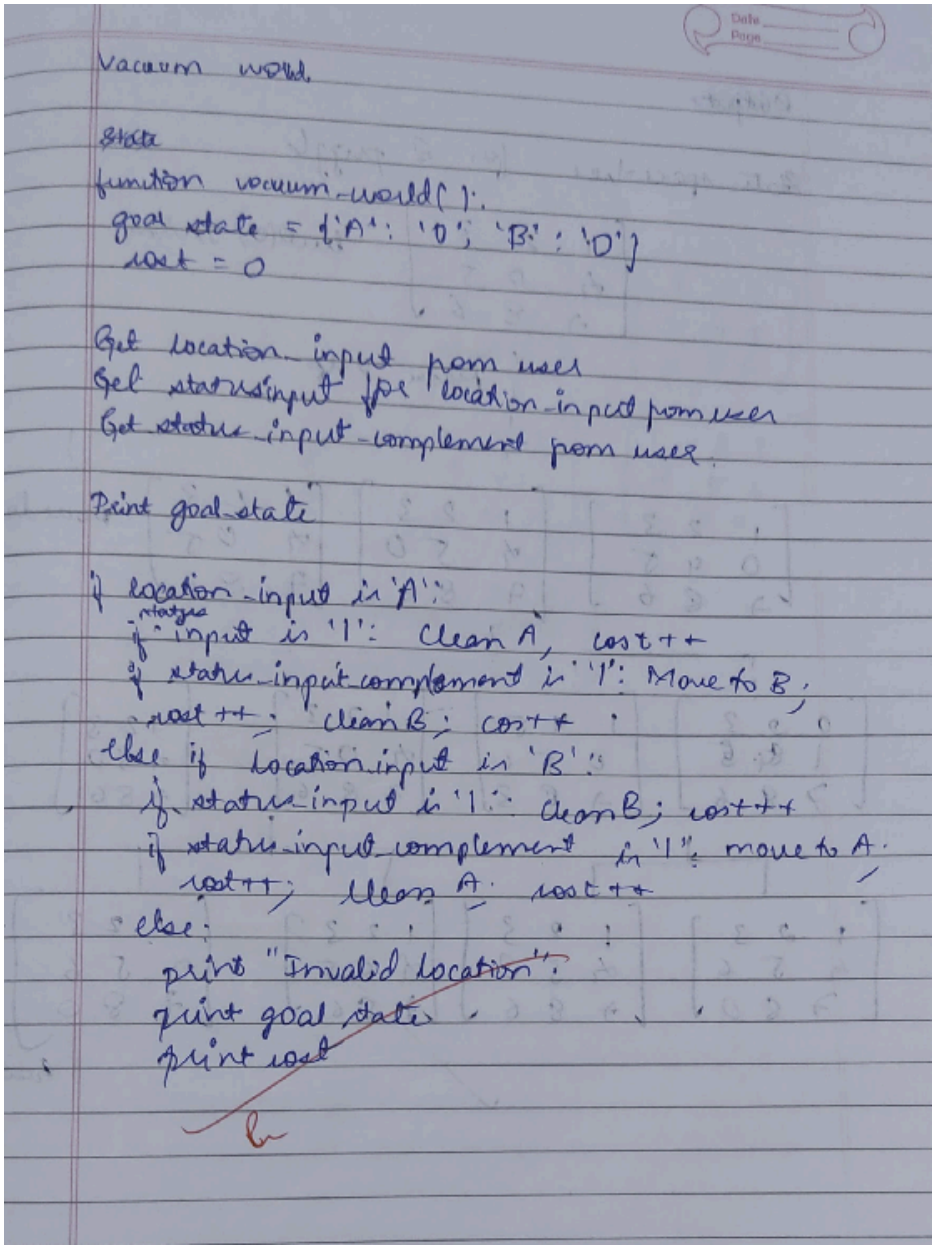
```
                bestScore = max(score, bestScore)
        return bestScore
    else:
        bestScore = 1000
        for key in board.keys():
            if board[key] == ' ':
                board[key] = player
                score = minimax(board, True)
                board[key] = ' '
                bestScore = min(score, bestScore)
        return bestScore

game_over = False
while not game_over:
    game_over = compMove()
    if not game_over:
        game_over = playerMove()
```

Output:

```
Output: 1BM22CS290
X| |
-+-+-
 | |
-+-+-
 | |

Enter position for O: 5
X| |
-+-+-
 |O|
-+-+-
 | |

X|X|
-+-+-
 |O|
-+-+-
 | |

Enter position for O: 3
X|X|O
-+-+-
 |O|
-+-+-
 | |

X|X|O
-+-+-
 |O|
-+-+-
X| |

Enter position for O: 4
X|X|O
-+-+-
O|O|
-+-+-
X| |

X|X|O
-+-+-
O|O|X
-+-+-
X| |

Enter position for O: 7
Position taken, please pick a different position.
Enter new position: 8
X|X|O
-+-+-
O|O|X
-+-+-
X|O|

X|X|O
-+-+-
O|O|X
-+-+-
X|O|X

Draw!
```

b) Implement Vacuum Cleaner Agent.

Algorithm:



Code:

```
def vacuum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter Location of Vacuum (A or B): ").strip().upper()
    status_input = input(f"Enter status of A (0 for Clean, 1 for Dirty): ").strip()
    status_input_complement = input("Enter status of B (0 for Clean, 1 for Dirty): ").strip()
```

```
print("Initial Location Condition: " + str(goal_state))
if location_input == 'A':
    print("Vacuum is placed in Location A")
    if status_input == '1':
        print("Location A is Dirty.")
        goal_state['A'] = '0'
        cost += 1
        print("Cost for cleaning A: " + str(cost))
        print("Location A has been Cleaned.")

        if status_input_complement == '1':
            print("Location B is Dirty.")
            print("Moving right to Location B.")
            cost += 1
            print("Cost for moving RIGHT: " + str(cost))
            goal_state['B'] = '0'
            cost += 1
            print("Cost for suck: " + str(cost))
            print("Location B has been Cleaned.")
        else:
            print("Location B is already clean.")
    else:
        print("Location A is already clean.")
        if status_input_complement == '1':
            print("Location B is Dirty.")
            print("Moving RIGHT to Location B.")
            cost += 1
            print("Cost for moving RIGHT: " + str(cost))
            goal_state['B'] = '0'
            cost += 1
            print("Cost for suck: " + str(cost))
            print("Location B has been Cleaned.")
        else:
            print("Location B is already clean.")

elif location_input == 'B':
    print("Vacuum is placed in Location B")
    if status_input == '1':
        print("Location B is Dirty.")
        goal_state['B'] = '0'  # Clean B
        cost += 1  # Cost for sucking
        print("Cost for cleaning B: " + str(cost))
        print("Location B has been Cleaned.")

        if status_input_complement == '1':
```

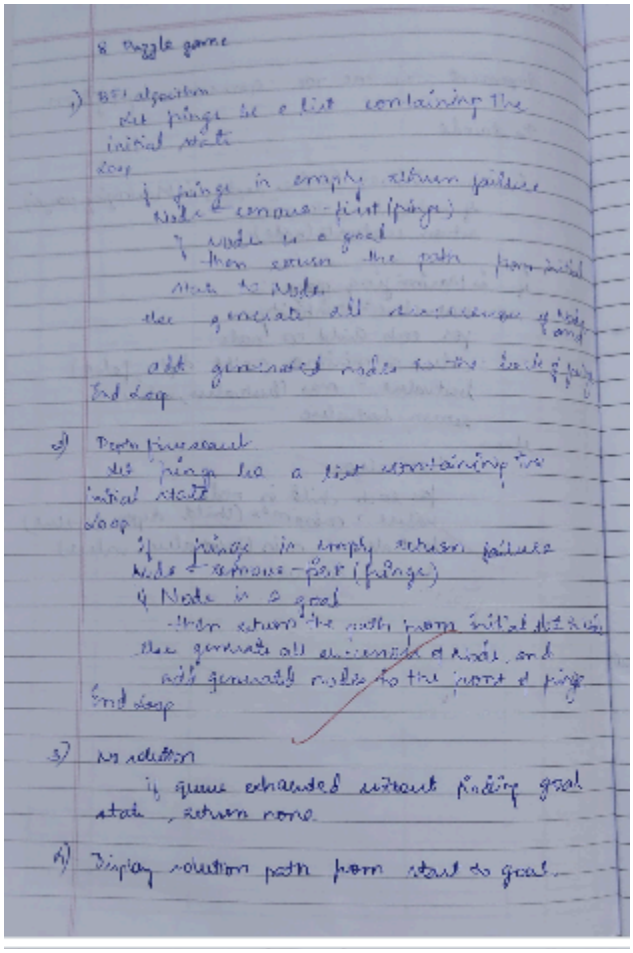
```
            print("Location A is Dirty.")
            print("Moving LEFT to Location A.")
            cost += 1  # Cost for moving left
            print("Cost for moving LEFT: " + str(cost))
            goal_state['A'] = '0'
            cost += 1
            print("Cost for suck: " + str(cost))
            print("Location A has been Cleaned.")
        else:
            print("Location A is already clean.")
    else:
        print("Location B is already clean.")
        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("Moving LEFT to Location A.")
            cost += 1
            print("Cost for moving LEFT: " + str(cost))
            goal_state['A'] = '0'
            cost += 1
            print("Cost for suck: " + str(cost))
            print("Location A has been Cleaned.")
        else:
            print("Location A is already clean.")
  print("GOAL STATE: ")
  print(goal_state)
  print("Performance Measurement: " + str(cost))
vacuum_world()
print("-------------------------")
print("Output: 1BM22CS290")
```

Output:

```
Enter Location of Vacuum (A or B): A
Enter status of A (0 for Clean, 1 for Dirty): 0
Enter status of B (0 for Clean, 1 for Dirty): 1
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is already clean.
Location B is Dirty.
Moving RIGHT to Location B.
Cost for moving RIGHT: 1
Cost for suck: 2
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 2
-------------------------
Output: 1BM22CS290
```

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

Algorithm:



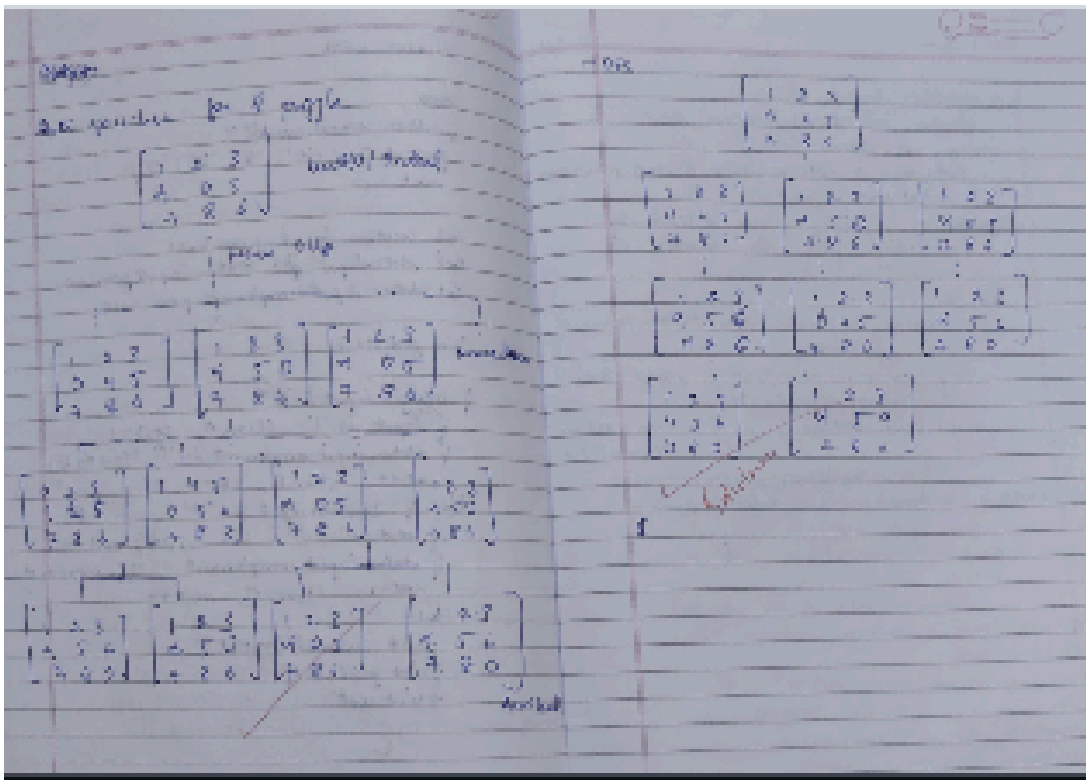
Code:

```
from collections import deque

def is_solvable(state):
    inv_count = 0
    state_flat = [tile for row in state for tile in row if tile != 0]
    for i in range(len(state_flat)):
        for j in range(i + 1, len(state_flat)):
            if state_flat[i] > state_flat[j]:
                inv_count += 1
    return inv_count % 2 == 0

def find_blank(state):
    for i, row in enumerate(state):
        for j, val in enumerate(row):
```

```
            if val == 0:
                return i, j

def get_neighbors(state):
    neighbors = []
    blank_i, blank_j = find_blank(state)
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
    for di, dj in directions:
        new_i, new_j = blank_i + di, blank_j + dj
        if 0 <= new_i < 3 and 0 <= new_j < 3:
            new_state = [row[:] for row in state]
                new_state[blank_i][blank_j], new_state[new_i][new_j] = new_state[new_i][new_j],
new_state[blank_i][blank_j]
            neighbors.append(new_state)
    return neighbors

def dfs(initial_state, goal_state):
    stack = [(initial_state, [])]
    visited = set()
    while stack:
        state, path = stack.pop()
        state_tuple = tuple(tuple(row) for row in state)
        if state_tuple in visited:
            continue
        visited.add(state_tuple)
        if state == goal_state:
            return path + [state]
        for neighbor in get_neighbors(state):
            stack.append((neighbor, path + [state]))
    return None

def bfs(initial_state, goal_state):
    queue = deque([(initial_state, [])])
    visited = set()
    while queue:
        state, path = queue.popleft()
        state_tuple = tuple(tuple(row) for row in state)
        if state_tuple in visited:
            continue
        visited.add(state_tuple)
        if state == goal_state:
            return path + [state]
        for neighbor in get_neighbors(state):
            queue.append((neighbor, path + [state]))
    return None
```

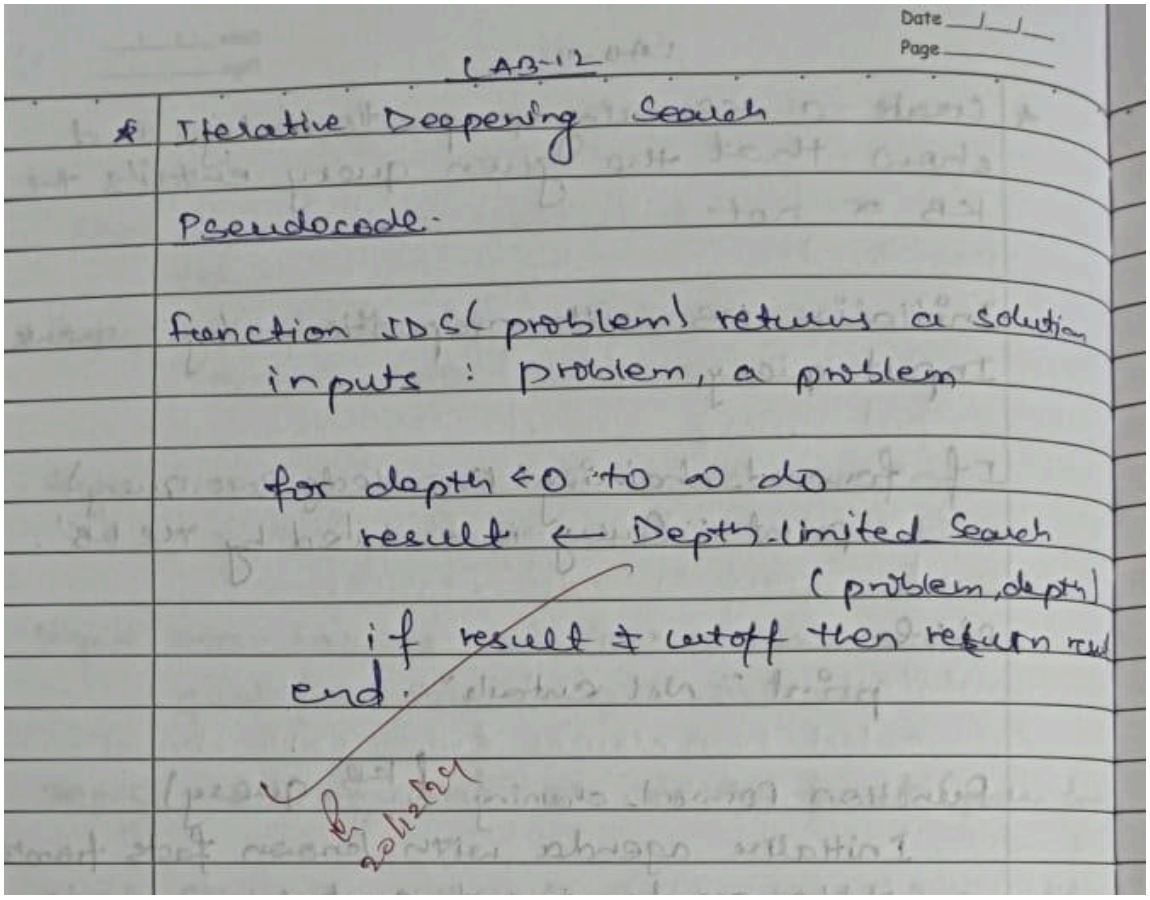
```
def display_path(path):
    for step, state in enumerate(path):
        print(f"Step {step}:")
        for row in state:
            print(row)
        print()

if __name__ == "__main__":
    print("Output: 1BM22CS290")
    print("Enter the initial state (3x3 grid, 0 for blank):")
    initial_state = [list(map(int, input().split())) for _ in range(3)]
    print("Enter the goal state (3x3 grid, 0 for blank):")
    goal_state = [list(map(int, input().split())) for _ in range(3)]
    if not is_solvable(initial_state):
        print("The given puzzle is not solvable.")
    else:
        print("Choose the method to solve the puzzle:")
        print("1. Depth-First Search (DFS)")
        print("2. Breadth-First Search (BFS)")
        choice = int(input("Enter your choice (1 or 2): "))
        match choice:
            case 1:
                print("Solving using DFS...")
                dfs_solution = dfs(initial_state, goal_state)
                if dfs_solution:
                    display_path(dfs_solution)
                else:
                    print("No solution found using DFS.")

            case 2:
                print("Solving using BFS...")
                bfs_solution = bfs(initial_state, goal_state)
                if bfs_solution:
                    display_path(bfs_solution)
                else:
                    print("No solution found using BFS.")
            case _:
                print("Invalid choice. Please select 1 or 2.")
```

Output:



b) Implement Iterative deepening search algorithm.

Algorithm:



Code:

```
from collections import defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        """Add an edge to the graph."""
        self.graph[u].append(v)

    def dls(self, node, target, depth):
        """
        Perform Depth-Limited Search (DLS) from the current node.
```

```
        :param node: Current node
        :param target: Target node
        :param depth: Maximum depth to explore
        :return: True if target is found, False otherwise
        """
        if depth == 0:
            return node == target
        if depth > 0:
            for neighbor in self.graph[node]:
                if self.dls(neighbor, target, depth - 1):
                    return True
        return False

    def iddfs(self, start, target, max_depth):
        """
        Perform Iterative Deepening Depth-First Search (IDDFS).

        :param start: Starting node
        :param target: Target node to search for
        :param max_depth: Maximum depth limit for IDDFS
        :return: True if target is found, False otherwise
        """
        for depth in range(max_depth + 1):
            print(f"Searching at depth: {depth}")
            if self.dls(start, target, depth):
                return True
        return False


# Example Usage
if __name__ == "__main__":
    g = Graph()
    # Construct the graph
    g.add_edge(0, 1)
    g.add_edge(0, 2)
```

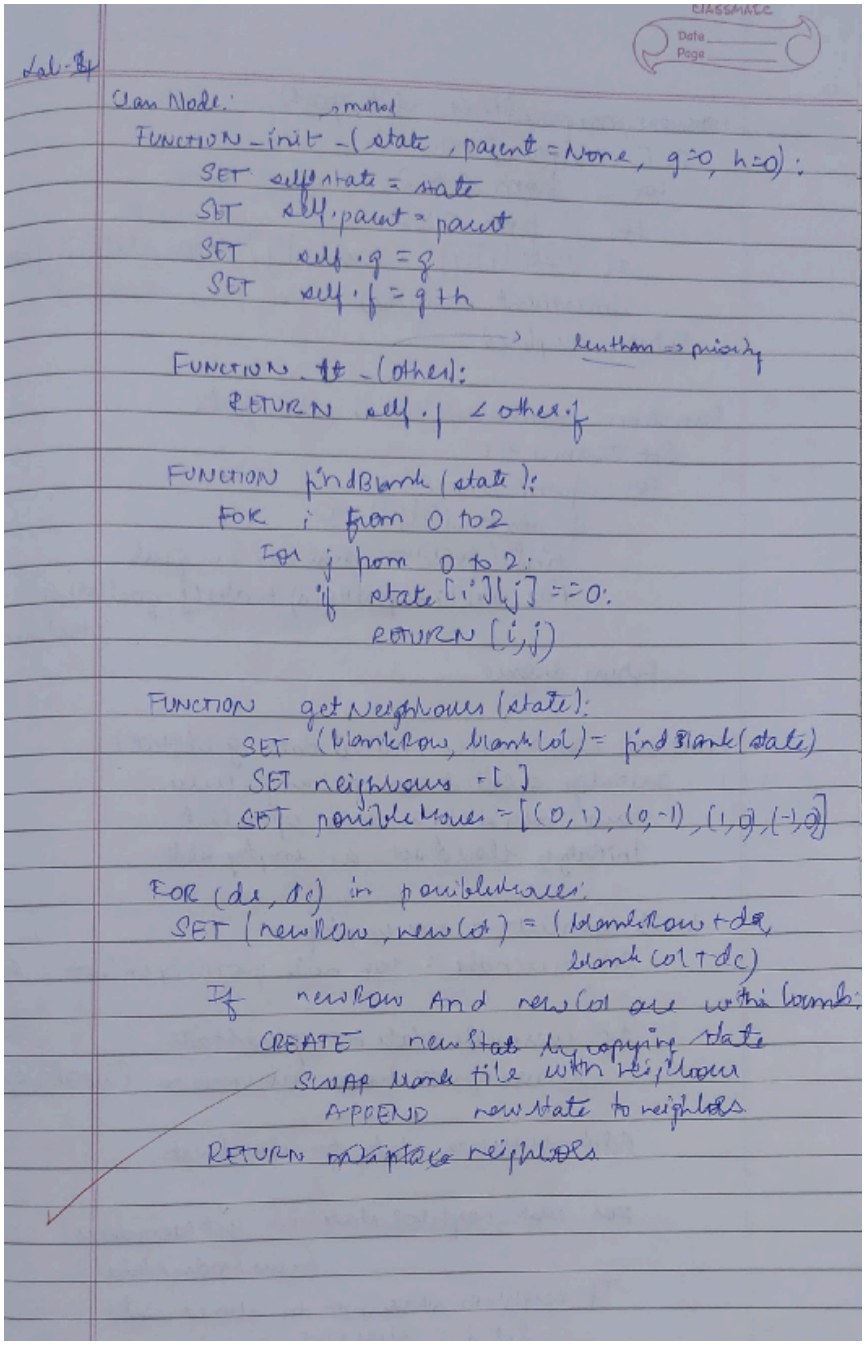
```
    g.add_edge(1, 3)
    g.add_edge(1, 4)
    g.add_edge(2, 5)
    g.add_edge(2, 6)

    start_node = 0
    target_node = 5
    max_depth = 3

    # Perform IDDFS
    if g.iddfs(start_node, target_node, max_depth):
        print(f"Target node {target_node} found within depth {max_depth}")
    else:
        print(f"Target node {target_node} NOT found within depth {max_depth}")
```

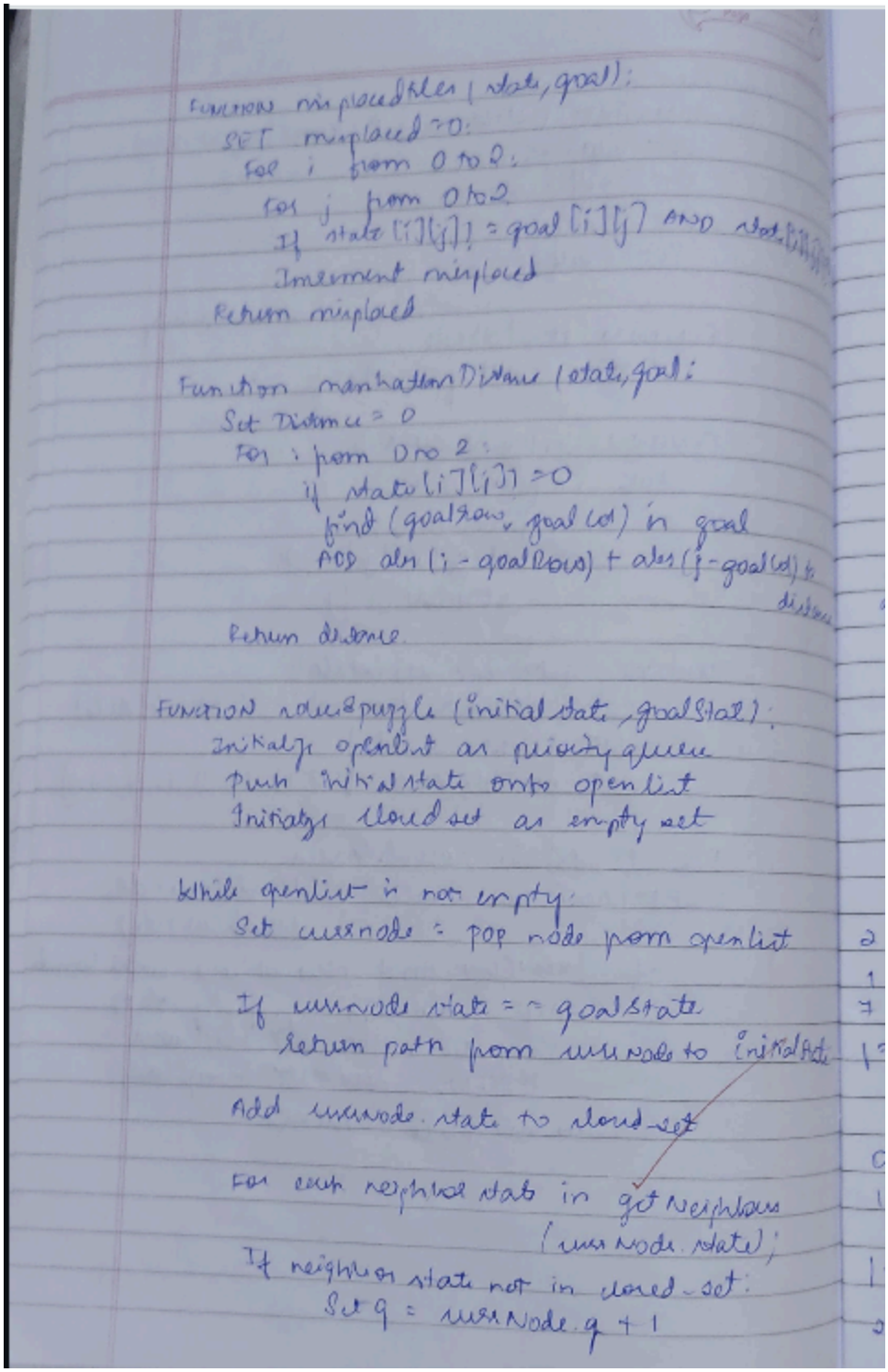
Output:

```
Searching at depth: 0
Searching at depth: 1
Searching at depth: 2
Target node 5 found within depth 3
```

3. Implement A* search algorithm.

Algorithm:





Code:

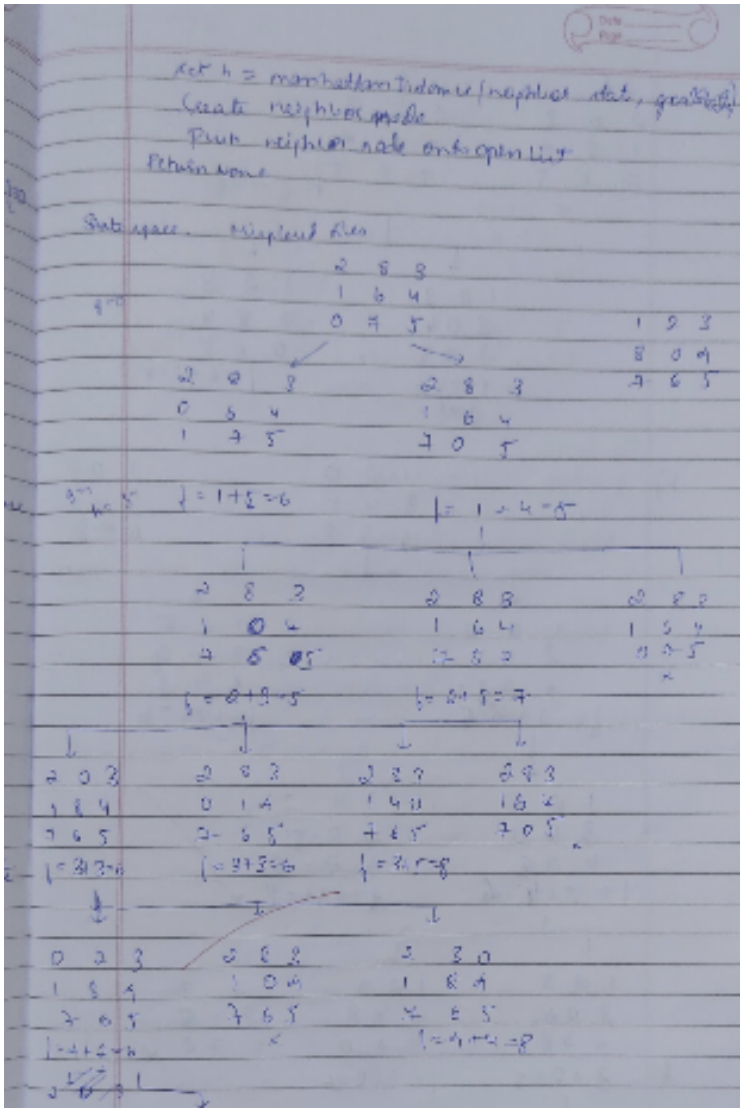
```
import heapq

class Node:
```

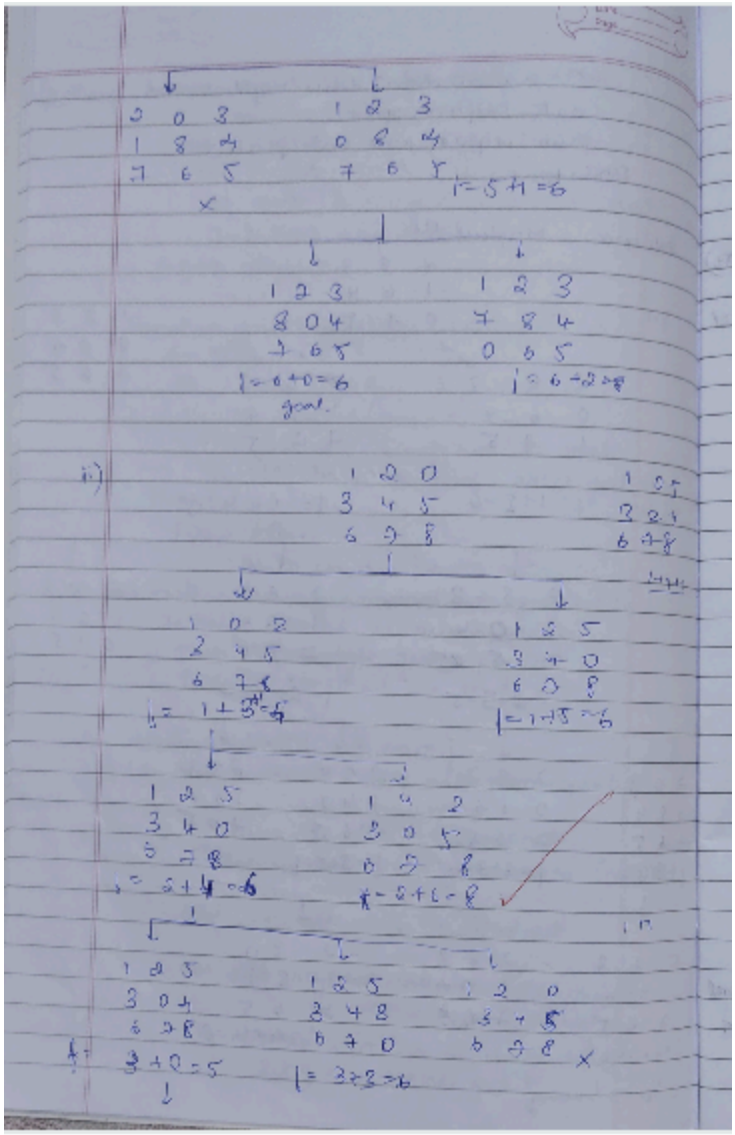
```
    def __init__(self, state, parent=None, g=0, h=0):
        self.state = state
        self.parent = parent
        self.g = g
        self.h = h
        self.f = g + h

    def __lt__(self, other):
        return self.f < other.f

def findBlank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def getNeighbors(state):
    blankRow, blankCol = findBlank(state)
    neighbors = []
    possibleMoves = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    for dr, dc in possibleMoves:
        newRow, newCol = blankRow + dr, blankCol + dc
        if 0 <= newRow < 3 and 0 <= newCol < 3:
            newState = [row[:] for row in state]
                                    newState[blankRow][blankCol],   newState[newRow][newCol]   =
newState[newRow][newCol], newState[blankRow][blankCol]
            neighbors.append(newState)
    return neighbors

def misplacedTiles(state, goal):
    misplaced = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != goal[i][j] and state[i][j] != 0:
                misplaced += 1
    return misplaced

def manhattanDistance(state, goal):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                goalRow, goalCol = -1, -1
                for x in range(3):
                    for y in range(3):
                        if state[i][j] == goal[x][y]:
```

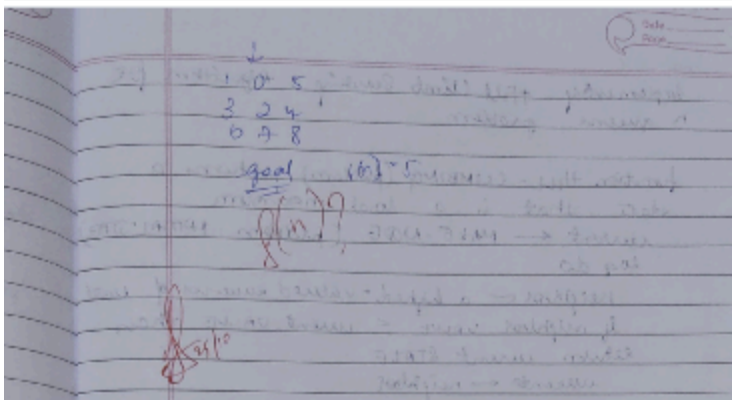
```
                    goalRow, goalCol = x, y
                    break
            distance += abs(i - goalRow) + abs(j - goalCol)
    return distance

def solve8puzzle(initialState, goalState, heuristic):
    openList = []
    if heuristic == "manhattan":
        h = manhattanDistance(initialState, goalState)
    else:
        h = misplacedTiles(initialState, goalState)


    heapq.heappush(openList, Node(initialState, None, 0, h))
    closed_set = set()

    while openList:
        currNode = heapq.heappop(openList)

        if tuple(map(tuple, currNode.state)) == tuple(map(tuple, goalState)):
            path = []
            while currNode:
                path.append(currNode.state)
                currNode = currNode.parent
            return path[::-1]

        closed_set.add(tuple(map(tuple, currNode.state)))

        for neighbor_state in getNeighbors(currNode.state):
            if tuple(map(tuple, neighbor_state)) not in closed_set:
                g = currNode.g + 1
                if heuristic == "manhattan":
                    h = manhattanDistance(neighbor_state, goalState)
                else:
                    h = misplacedTiles(neighbor_state, goalState)
                neighbor_node = Node(neighbor_state, currNode, g, h)
                heapq.heappush(openList, neighbor_node)

    return None

initialState = []
goalState = []

print("Output: 1BM22CS290")
print("Enter the initial state (3x3 matrix, use 0 for the blank tile):")
for i in range(3):
```

```
    row = list(map(int, input().split()))
    initialState.append(row)

print("Enter the goal state (3x3 matrix, use 0 for the blank tile):")
for i in range(3):
    row = list(map(int, input().split()))
    goalState.append(row)

# Prompt the user to choose the heuristic
heuristic = input("Choose a heuristic (1 for Misplaced Tiles, 2 for Manhattan Distance): ")
if heuristic == '1':
    heuristic = "misplaced"
elif heuristic == '2':
    heuristic = "manhattan"
else:
    print("Invalid choice. Defaulting to Manhattan Distance.")
    heuristic = "manhattan"

path = solve8puzzle(initialState, goalState, heuristic)

if path:
    print("Solution found!")
    for i, state in enumerate(path):
        print(f"Step {i}:")
        for row in state:
            print(row)
    print("Number of moves:", len(path) - 1)
else:
    print("No solution found.")
```

Output:

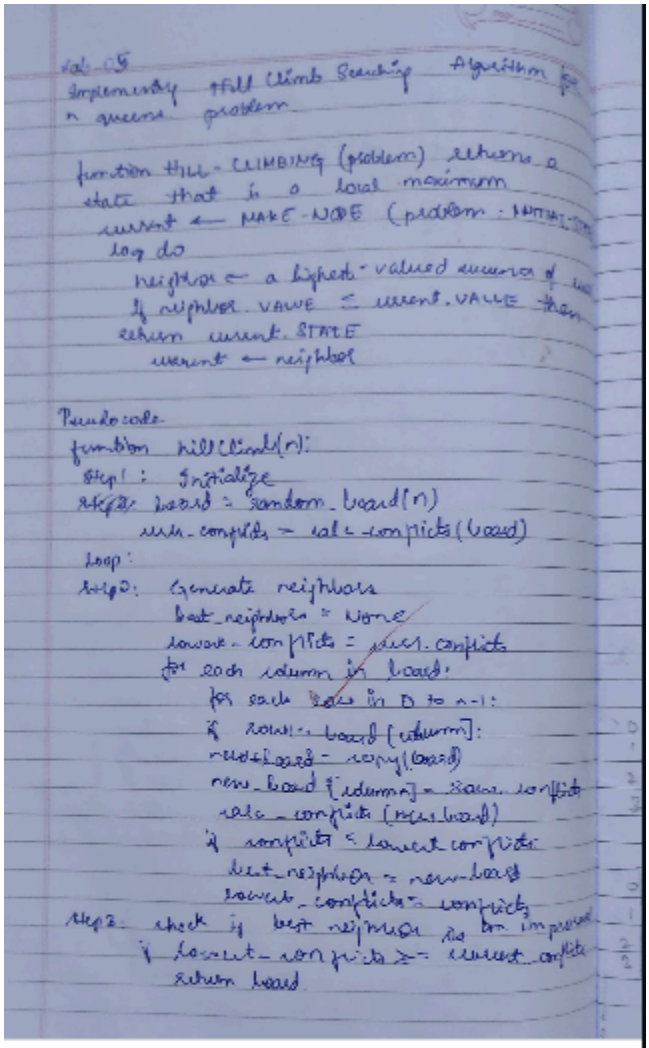


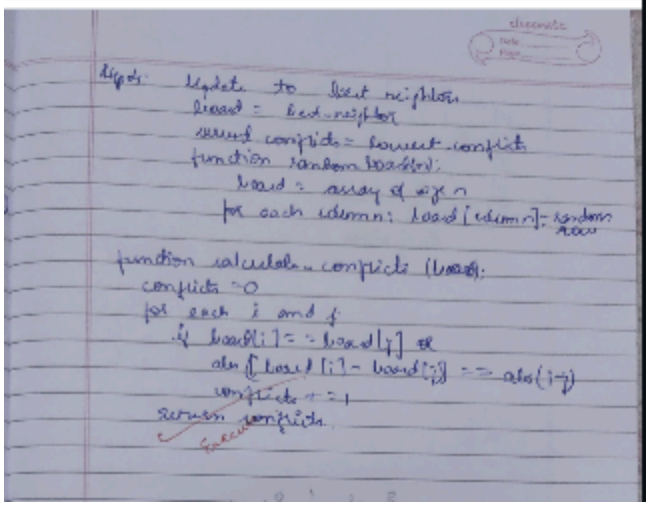




4. Implement Hill Climbing search algorithm to solve N-Queens problem.

Algorithm:



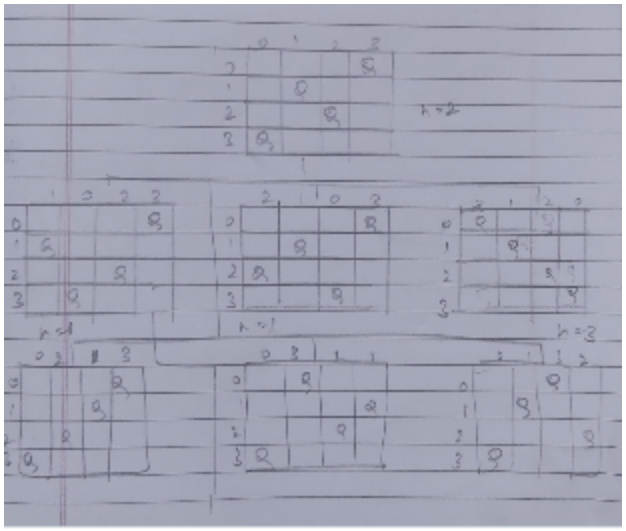


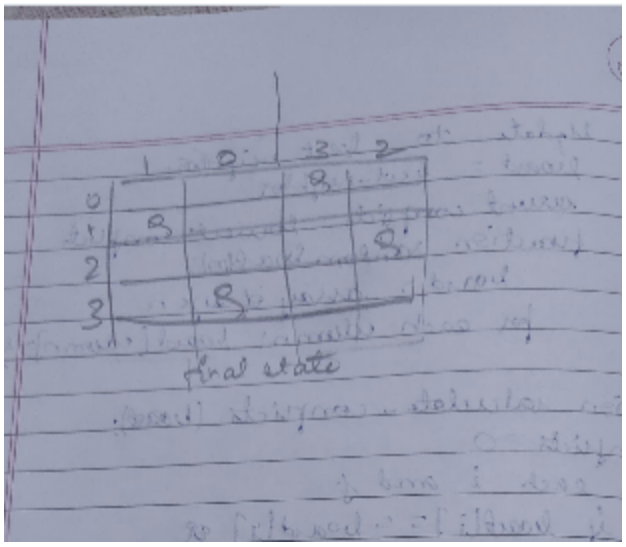
Code:

```
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:
        current_board = list(range(n))
        random.shuffle(current_board)
        current_conflicts = calculate_conflicts(current_board)

        while True:
            found_better = False
            for i in range(n):
                for j in range(n):
                    if j != current_board[i]:
                        neighbor_board = list(current_board)
                        neighbor_board[i] = j
                        neighbor_conflicts = calculate_conflicts(neighbor_board)
                        if neighbor_conflicts < current_conflicts:
                            print("Current Board:")
                            print_board(current_board)
                            print(f"Current Conflicts: {current_conflicts}")
                            print("Neighbor Board:")
                            print_board(neighbor_board)
                            print(f"Neighbor Conflicts: {neighbor_conflicts}")
                            current_board = neighbor_board
                            current_conflicts = neighbor_conflicts
                            cost += 1
                            found_better = True
                            break
                if found_better:
                    break

            if not found_better:
                break

        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'
        print(' '.join(row))
    print()

print("Output: 1BM22CS290")
n = 4
solution, conflicts, cost = hill_climbing(n)
print("Final Board Configuration:")
print_board(solution)
print("Number of Cost:", cost)
```

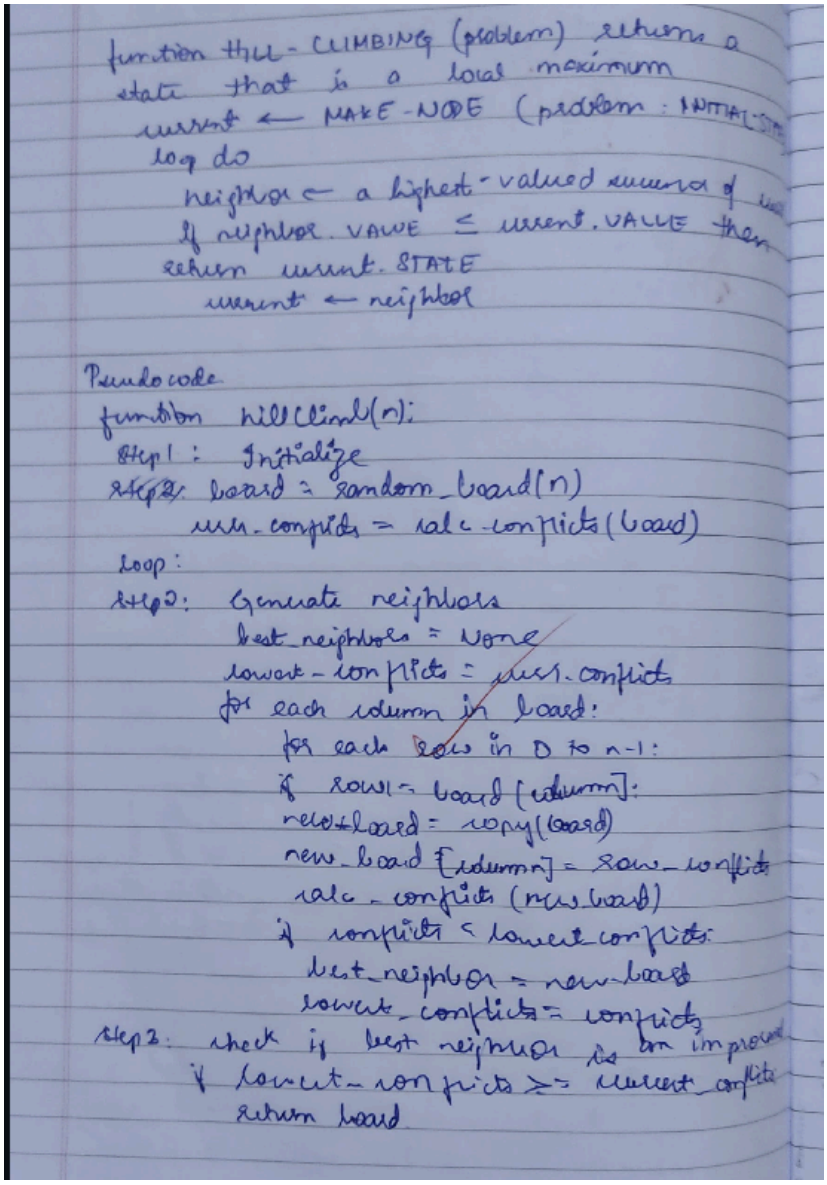
Output:





5. Implement Simulated Annealing to Solve 8-Queens problem.

Algorithm:



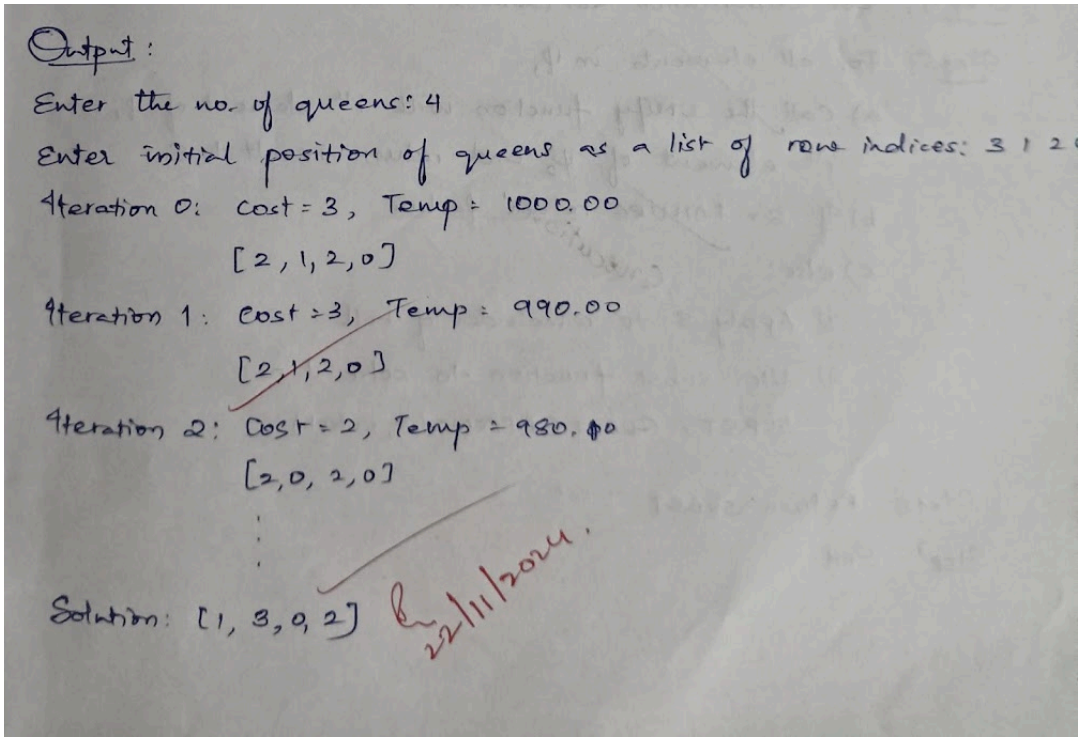
Code:

```
import random
import math

def count_conflicts(state):
    conflicts = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j]:
                conflicts += 1
            if abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
```

```
    return conflicts

def generate_neighbors(state):
    neighbors = []
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            neighbor = state[:]
            neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
            neighbors.append(neighbor)
    return neighbors

def acceptance_probability(old_cost, new_cost, temperature):
    if new_cost < old_cost:
        return 1.0
    return math.exp((old_cost - new_cost) / temperature)

def simulated_annealing(n, initial_state, initial_temp, cooling_rate, max_iterations):
    state = initial_state
    current_cost = count_conflicts(state)
    temperature = initial_temp

    for iteration in range(max_iterations):
        neighbors = generate_neighbors(state)
        random_neighbor = random.choice(neighbors)
        new_cost = count_conflicts(random_neighbor)

        if acceptance_probability(current_cost, new_cost, temperature) > random.random():
            state = random_neighbor
            current_cost = new_cost

        temperature *= cooling_rate

        if current_cost == 0:
            return state
    return None

def get_user_input(n):
    while True:
        try:
            print("Output: 1BM22CS290")
              user_input = input(f"Enter the column positions for the queens (space-separated integers
between 0 and {n-1}): ")
            initial_state = list(map(int, user_input.split()))
            for row in range(n):
              board = ['Q' if col == initial_state[row] else '.' for col in range(n)]
```

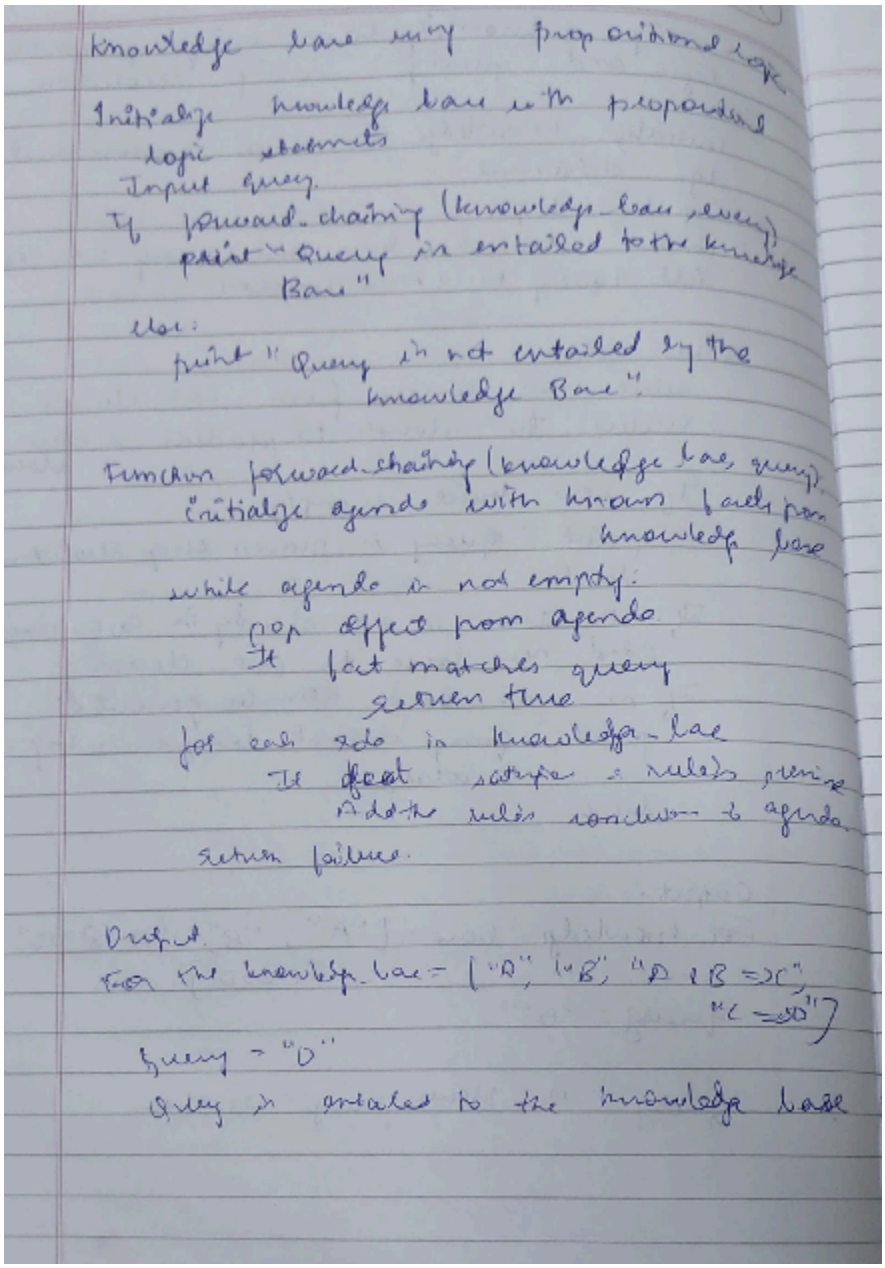
```
            print(' '.join(board))
        if len(initial_state) != n or any(x < 0 or x >= n for x in initial_state):
            print(f"Invalid input. Please enter exactly {n} integers between 0 and {n-1}.")
            continue
        return initial_state
    except ValueError:
        print(f"Invalid input. Please enter a list of {n} integers.")

n = 8
initial_state = get_user_input(n)

initial_temp = 1000
cooling_rate = 0.99
max_iterations = 10000

solution = simulated_annealing(n, initial_state, initial_temp, cooling_rate, max_iterations)
if solution:
    print("Solution found!")
    for row in range(n):
        board = ['Q' if col == solution[row] else '.' for col in range(n)]
        print(' '.join(board))
else:
    print("No solution found within the given iterations.")
```

Output:



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

Algorithm:



Code:

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

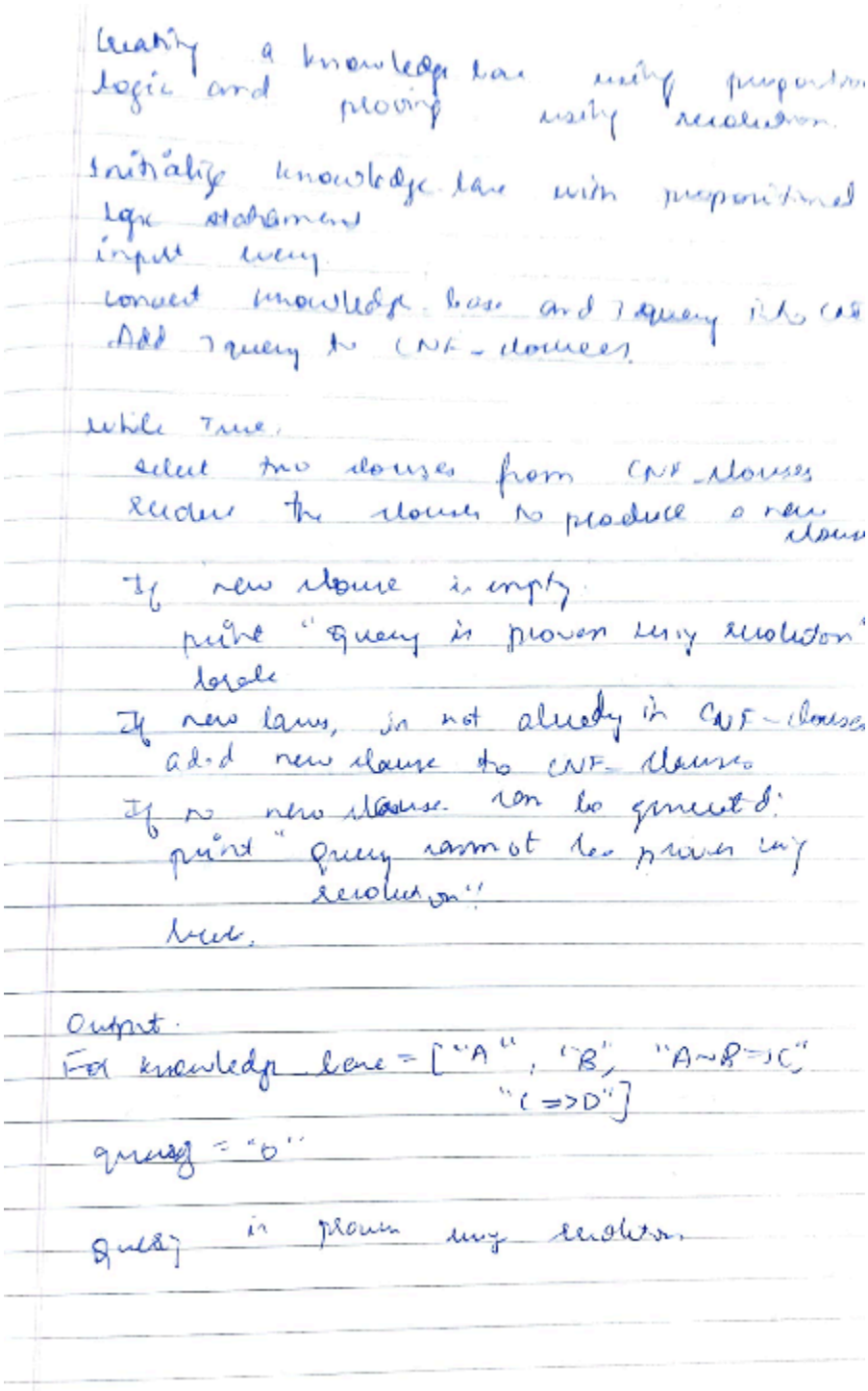
```
def is_entailment(kb, query):

    # Negate the query
    negated_query = Not(query)

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

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

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


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

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

Output:

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

7. Implement unification in first order logic.

Algorithm:



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
```

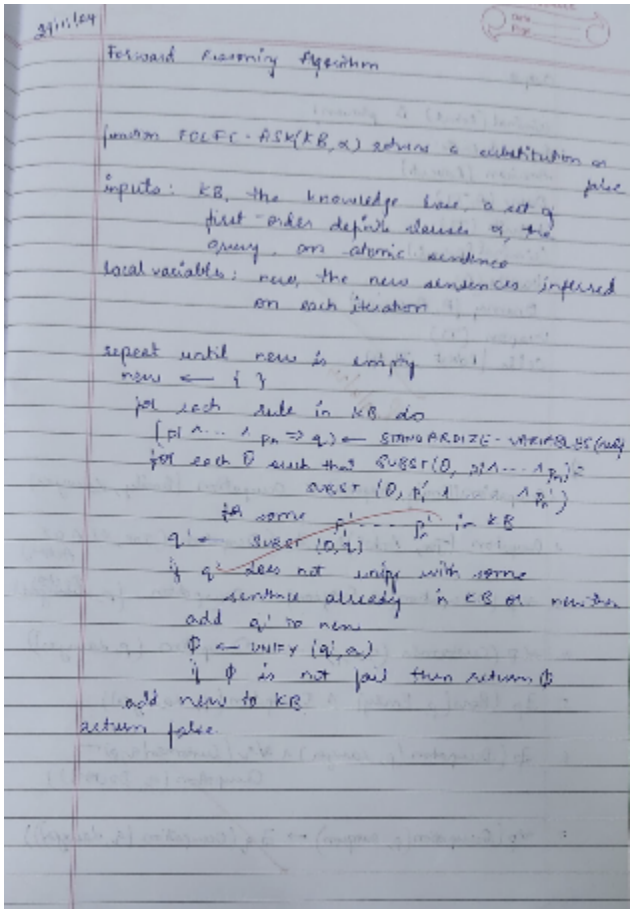
```
def display_result(expr1, expr2, is_unified, subst):
    print("Expression 1:", expr1)
    print("Expression 2:", expr2)
    if not is_unified:
        print("Result: Unification Failed")
    else:
        print("Result: Unification Successful")
        print("Substitutions:", {k: list(v) if isinstance(v, 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:
        print("Output: 1BM22CS290")
        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: 1BM22CS290
Enter the first expression (e.g., p(x, f(y))): p(x,y,z)
Enter the second expression (e.g., p(a, f(z))): p(a,b)
Expression 1: ['p', '(x', 'y', 'z)']
Expression 2: ['p', '(a', 'b)']
Result: Unification Failed
Do you want to test another pair of expressions? (yes/no): yes
Output: 1BM22CS290
Enter the first expression (e.g., p(x, f(y))): p(a,b)
Enter the second expression (e.g., p(a, f(z))): p(q,r)
Expression 1: ['p', '(a', 'b)']
Expression 2: ['p', '(q', 'r)']
Result: Unification Successful
Substitutions: {'(a': '(q', 'b)': 'r)'}
Do you want to test another pair of expressions? (yes/no): no
```

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

Algorithm:

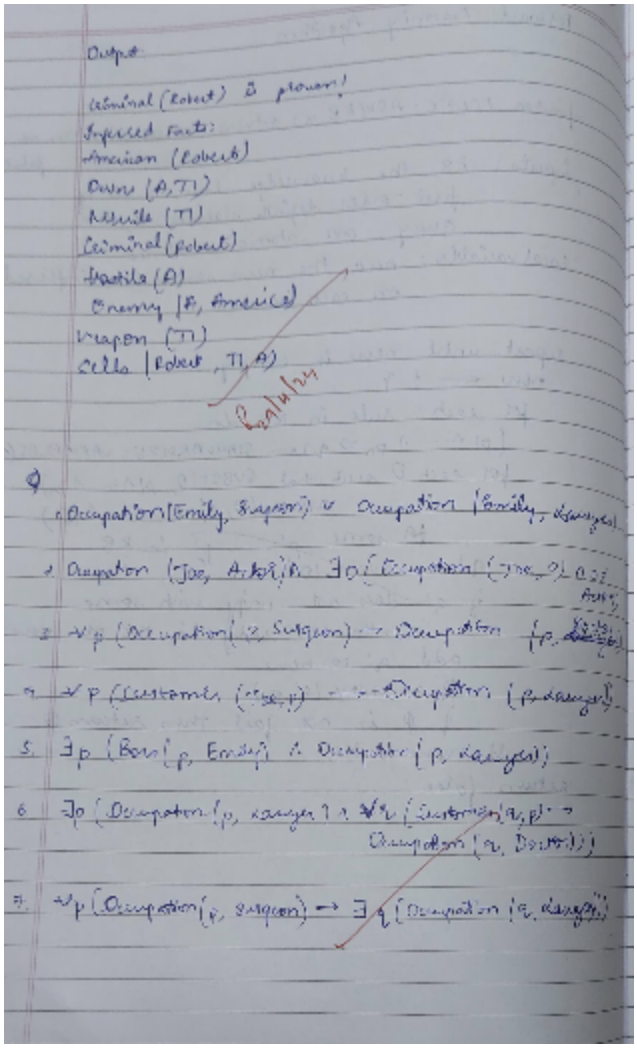


Code:

```
# Define initial facts and rules
facts = {"InAmerica(West)", "SoldWeapons(West, Nono)", "Enemy(Nono, America)"}
rules = [
    {
        "conditions": ["InAmerica(x)", "SoldWeapons(x, y)", "Enemy(y, America)"],
        "conclusion": "Criminal(x)",
    },
    {
        "conditions": ["Enemy(y, America)"],
        "conclusion": "Dangerous(y)",
    },
]

# Forward chaining function
```

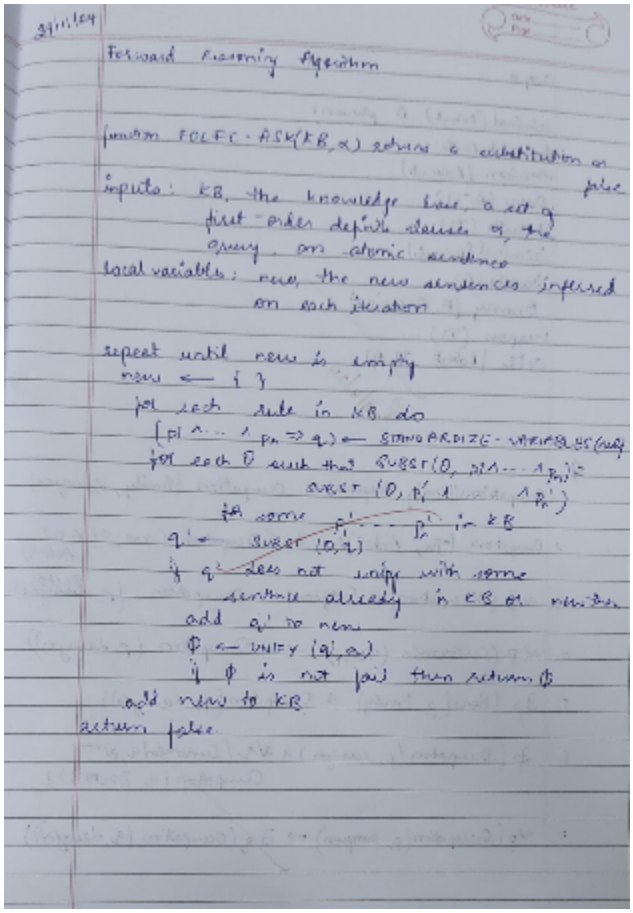
```
def forward_chaining(facts, rules):
    derived_facts = set(facts)  # Initialize derived facts
    while True:
        new_fact_found = False

        for rule in rules:
            # Substitute variables and check if conditions are met
            for fact in derived_facts:
                if "x" in rule["conditions"][0]:
                    # Substitute variables (x, y) with specific instances
                    for condition in rule["conditions"]:
                        if "x" in condition or "y" in condition:
                            x = "West"  # Hardcoded substitution for simplicity
                            y = "Nono"
                            conditions = [
                                cond.replace("x", x).replace("y", y)
                                for cond in rule["conditions"]
                            ]
                            conclusion = (
                                rule["conclusion"].replace("x", x).replace("y", y)
                            )

                            # Check if all conditions are satisfied
                                    if all(cond in derived_facts for cond in conditions) and conclusion not in
derived_facts:
                                derived_facts.add(conclusion)
                                print(f"New fact derived: {conclusion}")
                                new_fact_found = True

        # Exit loop if no new fact is found
        if not new_fact_found:
            break

    return derived_facts

# Run forward chaining
final_facts = forward_chaining(facts, rules)
print("Output: 1BM22CS290")
print("\nFinal derived facts:")
for fact in final_facts:
    print(fact)
```

Output:


Output
Criminal (Robert) ? proven!
Injected facts:
American (Robert)
Owns (A, T1)
Missile (T1)
Criminal (Robert)
Hostile (A)
Enemy (A, America)
Weapon (T1)
Sells (Robert, T1, A)
Evaluate
Q
1. Occupation (Emily, Surgeon) ∨ Occupation (Emily, Lawyer)
2. Occupation (Joe, Actor) ∧ ∃p (Occupation (p, Doctor) ∧ Age
3. ∀p (Occupation (p, Surgeon) → Occupation (p,
4. ∀p (Customer (p, p) → Occupation (p, Lawyer))
5. ∃p (Born (p, Emily) ∧ Occupation (p, Lawyer))
6. ∃p (Occupation (p, Lawyer) ∧ ∀q (Customer (q, p) →
Occupation (q, Doctor)))
7. ∀p (Occupation (p, Surgeon) → ∃q (Occupation (q, Lawyer)))

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

Algorithm:



Code:

```
# Define the knowledge base (KB)
KB = {
    "food(Apple)": True,
    "food(vegetables)": True,
    "eats(Anil, Peanuts)": True,
    "alive(Anil)": True,
    "likes(John, X)": "food(X)",  # Rule: John likes all food
    "food(X)": "eats(Y, X) and not killed(Y)",  # Rule: Anything eaten and not killed is food
    "eats(Harry, X)": "eats(Anil, X)",  # Rule: Harry eats what Anil eats
    "alive(X)": "not killed(X)",  # Rule: Alive implies not killed
    "not killed(X)": "alive(X)",  # Rule: Not killed implies alive
}

# Function to evaluate if a predicate is true based on the KB
```

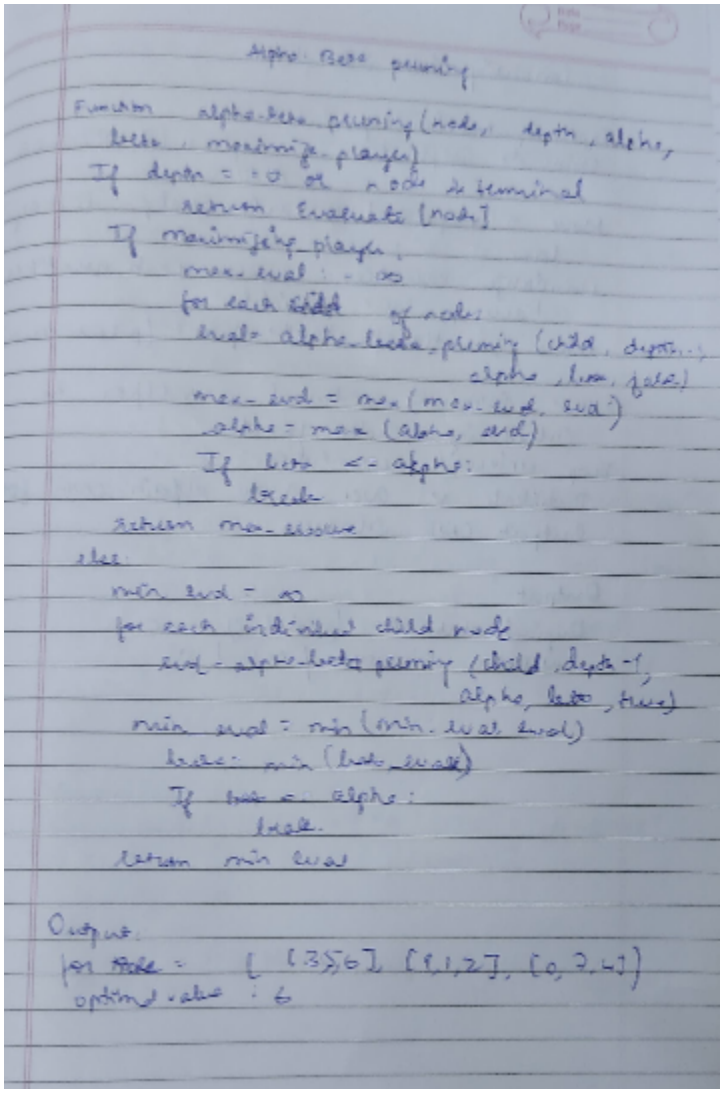
```
def resolve(predicate):
    # If it's a direct fact in KB
    if predicate in KB and isinstance(KB[predicate], bool):
        return KB[predicate]

    # 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 is a specific query (e.g., likes(John, Peanuts))
    if "(" in predicate:
        func, args = predicate.split("(")
        args = args.strip(")").split(", ")
        if func == "food" and args[0] == "Peanuts":
            return resolve("eats(Anil, Peanuts)") and not resolve("killed(Anil)")
        if func == "likes" and args[0] == "John" and args[1] == "Peanuts":
            return resolve("food(Peanuts)")

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

# Query to prove: John likes Peanuts
query = "likes(John, Peanuts)"
result = resolve(query)

# Print the result
print("Output: 1BM22CS290")
print(f"Does John like peanuts? {'Yes' if result else 'No'}")
```

Output:

```
Output: 1BM22CS290
Does John like peanuts? Yes
```

10. Implement Alpha-Beta Pruning.

Algorithm:



Code:

```
def alpha_beta_pruning(node, alpha, beta, maximizing_player):
    if type(node) is int:
        return node

    if maximizing_player:
        max_eval = -float('inf')
        for child in node:
            eval = alpha_beta_pruning(child, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
```

```
            break
        return max_eval
    else:
        min_eval = float('inf')
        for child in node:
            eval = alpha_beta_pruning(child, alpha, beta, True)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return min_eval

def build_tree(numbers):
    current_level = [[n] for n in numbers]

    while len(current_level) > 1:
        next_level = []
        for i in range(0, len(current_level), 2):
            if i + 1 < len(current_level):
                next_level.append(current_level[i] + current_level[i + 1])
            else:
                next_level.append(current_level[i])
        current_level = next_level

    return current_level[0]

def main():
    print("Output: 1BM22CS290")
    numbers = list(map(int, input("Enter numbers for the game tree (space-separated): ").split()))
    tree = build_tree(numbers)

    alpha = -float('inf')
    beta = float('inf')
    maximizing_player = True

    result = alpha_beta_pruning(tree, alpha, beta, maximizing_player)
    print("Final Result of Alpha-Beta Pruning:", result)

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

Output:

```
Output: 1BM22CS290
Enter numbers for the game tree (space-separated): 10 9 14 18 5 4 50 3
Final Result of Alpha-Beta Pruning: 50
```