VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Artificial Intelligence (23CS5PCAIN)

Submitted by

Vignesh Bhat(1BM22CS327)

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 Bhat(1BM22CS327),** 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 of the above mentioned subject and the work prescribed for the said degree.

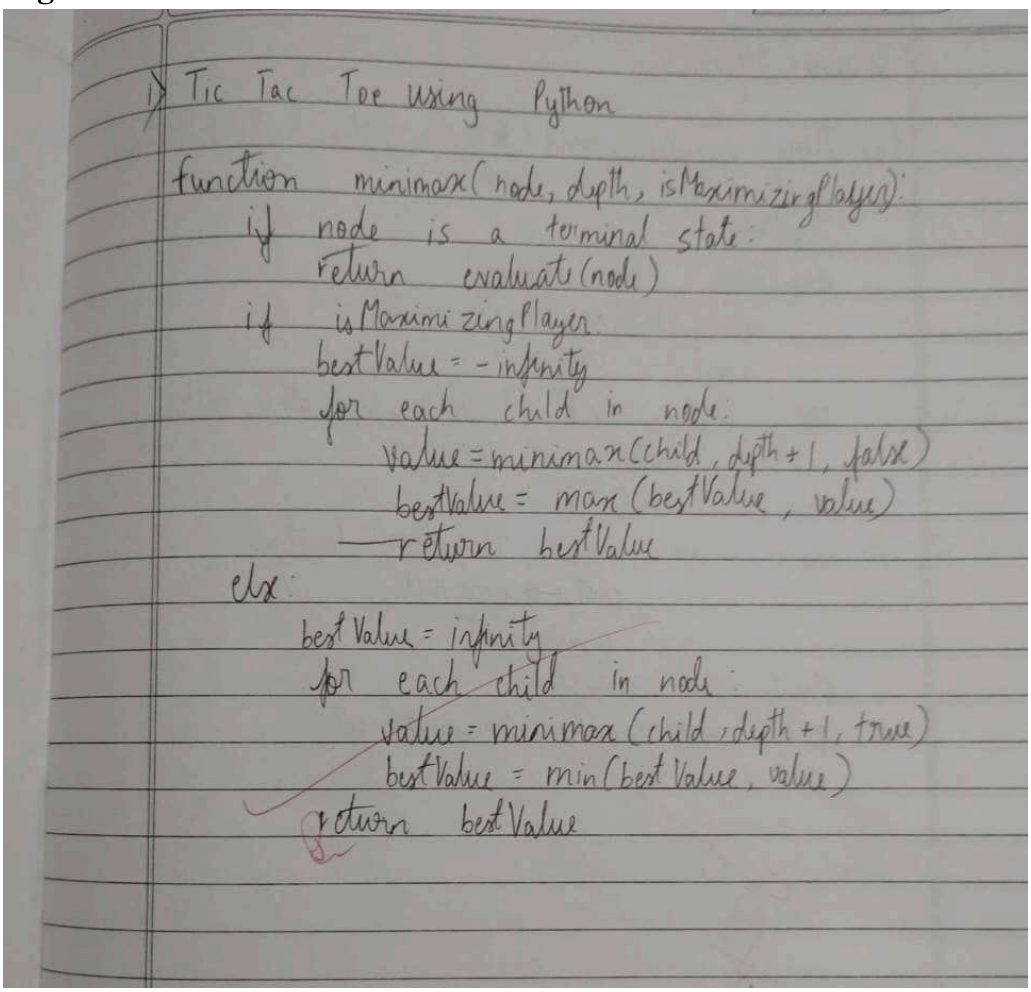
Dr Rashmi H Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor and Head Department of CSE, BMSCE

Index

Program 1

Implement Tic –Tac –Toe Game

Algorithm :



1) Tic Tac Toe using Python

```
function minimax(node, depth, isMaximizingPlayer):
    if node is a terminal state:
        Return evaluate(node)
    if isMaximizingPlayer:
        bestValue = -infinity
        for each child in node:
            value = minimax(child, depth+1, false)
            bestValue = max(bestValue, value)
        return bestValue
    else:
        bestValue = infinity
        for each child in node:
            value = minimax(child, depth+1, true)
            bestValue = min(bestValue, value)
        return bestValue
```

Code :

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



def printBoard(board):
   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):
   if(board[pos]==' '):
       return True
   else:
       return False

def checkWin():
   if(board[1]==board[2] and board[1]==board[3] and board[1]!=' '):
       return True
   elif(board[4]==board[5] and board[4]==board[6] and board[4]!=' '):
       return True
   elif(board[7]==board[8] and board[7]==board[9] and board[7]!=' '):
       return True
   elif (board[1] == board[5] and board[1] == board[9] and board[1] != ' '):
       return True
   elif (board[3] == board[5] and board[3] == board[7] and board[3] != ' '):
       return True
   elif (board[1] == board[4] and board[1] == board[7] and board[1] != ' '):
       return True
   elif (board[2] == board[5] and board[2] == board[8] and board[2] != ' '):
       return True
   elif (board[3] == board[6] and board[3] == board[9] and board[3] != ' '):
       return True
   else:
       return False
```

```
def checkMoveForWin(move):
    if (board[1]==board[2] and board[1]==board[3] and board[1] ==move):
        return True
    elif (board[4]==board[5] and board[4]==board[6] and board[4] ==move):
        return True
    elif (board[7]==board[8] and board[7]==board[9] and board[7] ==move):
        return True
    elif (board[1]==board[5] and board[1]==board[9] and board[1] ==move):
        return True
    elif (board[3]==board[5] and board[3]==board[7] and board[3] ==move):
        return True
    elif (board[1]==board[4] and board[1]==board[7] and board[1] ==move):
        return True
    elif (board[2]==board[5] and board[2]==board[8] and board[2] ==move):
        return True
    elif (board[3]==board[6] and board[3]==board[9] and board[3] ==move):
        return True
    else:
        return False

def checkDraw():
    for key in board.keys():
        if (board[key]==' '):
            return False
    return True


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

        if (checkDraw()):
            print('Draw!')
        elif (checkWin()):
            if (letter == 'X'):
                print('Bot wins!')
            else:
                print('You win!')
        return

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

```
player = 'O'
bot ='X'

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

def compMove():
   bestScore=-100
   0 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

   insertLetter(bot, bestMove)
   return


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] = ' '
               if (score > bestScore):
                   bestScore = score
       return bestScore
   else:
       bestScore = 1000

       for key in board.keys():
```

```
            if board[key] == ' ':
                board[key] = player
score = minimax(board, True) board[key] = ' '
                if (score < bestScore):
                    bestScore = score
return bestScore

while not checkWin():
   playerMove()
   compMove()
```

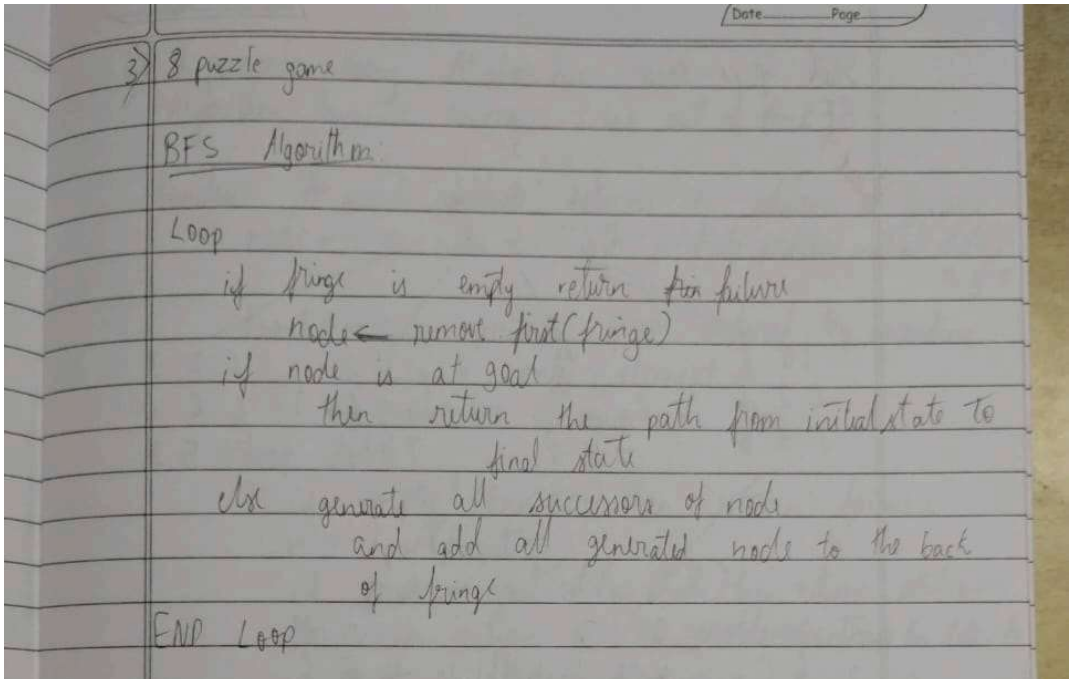
Output :

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


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


Enter position for O:2
 |O|O
-+-+-
 |X|
-+-+-
 | |


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


Enter position for O:6
X|O|O
-+-+-
 |X|O
-+-+-
 | |


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

Program 2

Solve 8 puzzle problems
1.BFS

Algorithm :



Code :

```
from collections import deque

class PuzzleState:
    def _init_(self, board, zero_position, path=[]):
        self.board = board
self.zero_position = zero_position self.path =
        path

def is_goal(self):
return self.board == [1, 2, 3, 4, 5, 6, 7, 8, 0]

    def get_possible_moves(self):
        moves = []
row, col = self.zero_position
directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]     # Right, Down, Left, Up

for dr, dc in directions:
new_row, new_col = row + dr, col + dc
```

```
            if 0 <= new_row < 3 and 0 <= new_col <
                3: new_board = self.board[:]
                # Swap zero with the adjacent tile
                new_board[row * 3 + col], new_board[new_row * 3 + new_col] =
new_board[new_row * 3 + new_col], new_board[row * 3 + col]
                moves.append(PuzzleState(new_board, (new_row, new_col), self.path +
[new_board]))

        return moves

def bfs(initial_state):
    queue = deque([initial_state])
    visited = set()

    while queue:
        current_state = queue.popleft()

        # Show the current board
        print("Current Board State:")
        print_board(current_state.board)
        print()

        if current_state.is_goal():
            return current_state.path

        visited.add(tuple(current_state.board))

        for next_state in current_state.get_possible_moves():
            if tuple(next_state.board) not in visited:
                queue.append(next_state)

    return None

def print_board(board):
    for i in range(3):
        print(board[i * 3:i * 3 + 3])

def main():
    print("Enter the initial state of the 8-puzzle (use 0 for the blank tile, e.g.,
'1 2 3 4 5 6 7 8 0'): ")
    user_input = input()
    initial_board = list(map(int, user_input.split()))

    if len(initial_board) != 9 or set(initial_board) != set(range(9)):
```

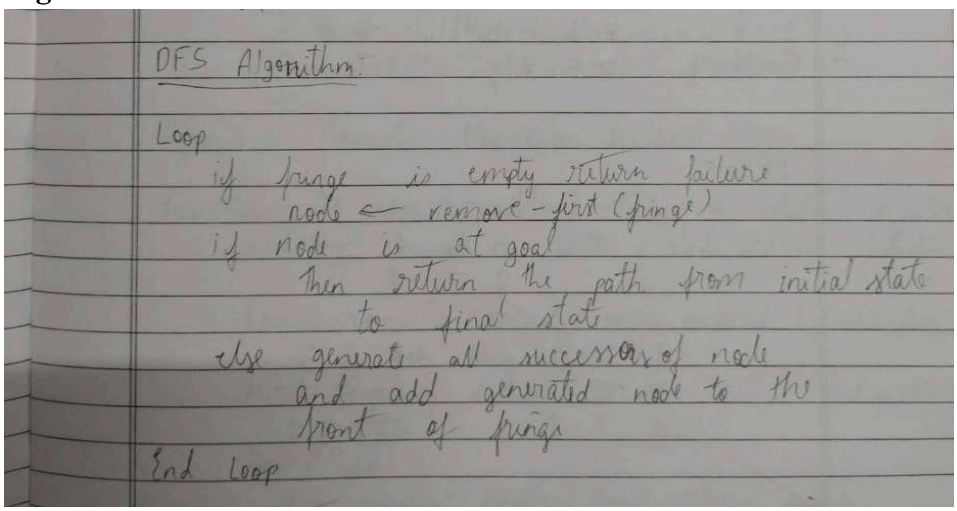
```
        print("Invalid input! Please enter 9 numbers from 0 to 8.")
      return
zero_position = initial_board.index(0)
initial_state = PuzzleState(initial_board, (zero_position // 3, zero_position %
3))
solution_path = bfs(initial_state) if
    solution_path is None:
        print("No solution found.")
    else:
print("Solution found in", len(solution_path), "steps.") for step
        in solution_path:
print_board(step) print()

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

Output :

```
Enter the initial state of the 8-puzzle (use 0 for the blank tile, e.g., '1 2 3 4 5 6 7 8 0'):
1 2 3 4 0 6 7 5 8
Current Board State:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Current Board State:
[1, 2, 3]
[4, 6, 0]
[7, 5, 8]

Current Board State:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Current Board State:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]

Current Board State:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]

Current Board State:
[1, 2, 3]
[4, 6, 8]
[7, 5, 0]

Current Board State:
[1, 2, 0]
[4, 6, 3]
[7, 5, 8]

Current Board State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Solution found in 2 steps.
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

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

2.DFS

Algorithm :



Code:

```
from collections import deque

def
    get_user_input(prompt)
    : board = []
    print(prompt)
for i in range(3):
        row = list(map(int, input(f"Enter row {i+1} (space-separated numbers, use 0
for empty space): ").split()))
        board.append(row)
    return board

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

class PuzzleState:
    def _init_(self, board, moves=0, previous=None):
        self.board = board
self.empty_tile = self.find_empty_tile()
        self.moves = moves
self.previous = previous

    def find_empty_tile(self):
        for i in range(3):
for j in range(3):
```

```
                    return (i, j)

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

    def get_possible_moves(self):
        row, col = self.empty_tile
        possible_moves = []
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]  # down, up, right, left

        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col <
                3: # Make the move
                new_board = [row[:] for row in self.board]  # Deep
                copy new_board[row][col], new_board[new_row][new_col]
                =
new_board[new_row][new_col], new_board[row][col]
                possible_moves.append(PuzzleState(new_board, self.moves + 1, self))

        return possible_moves

def dfs(initial_state,
    goal_state): stack =
    [initial_state]
    visited = set()

    while stack:
        current_state = stack.pop()

        if current_state.is_goal(goal_state):
            return current_state

        # Convert board to a tuple for the visited set
        state_tuple = tuple(tuple(row) for row in current_state.board)

        if state_tuple not in visited:
            visited.add(state_tuple)
            for next_state in current_state.get_possible_moves():
                stack.append(next_state)

    return None  # No solution found

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

```
        for row in state:
            print(row)
print()

if _name    == "_main_":
# Get user input for initial and goal states
initial_board = get_user_input("Enter the initial state of the puzzle:") goal_board
    = get_user_input("Enter the goal state of the puzzle:")

if is_solvable(initial_board):
initial_state = PuzzleState(initial_board)
        solution = dfs(initial_state, goal_board)

if solution:
print("Solution found in", solution.moves, "moves:") print_solution(solution)
else:
print("No solution found.")
else:
print("This puzzle is unsolvable.")
```

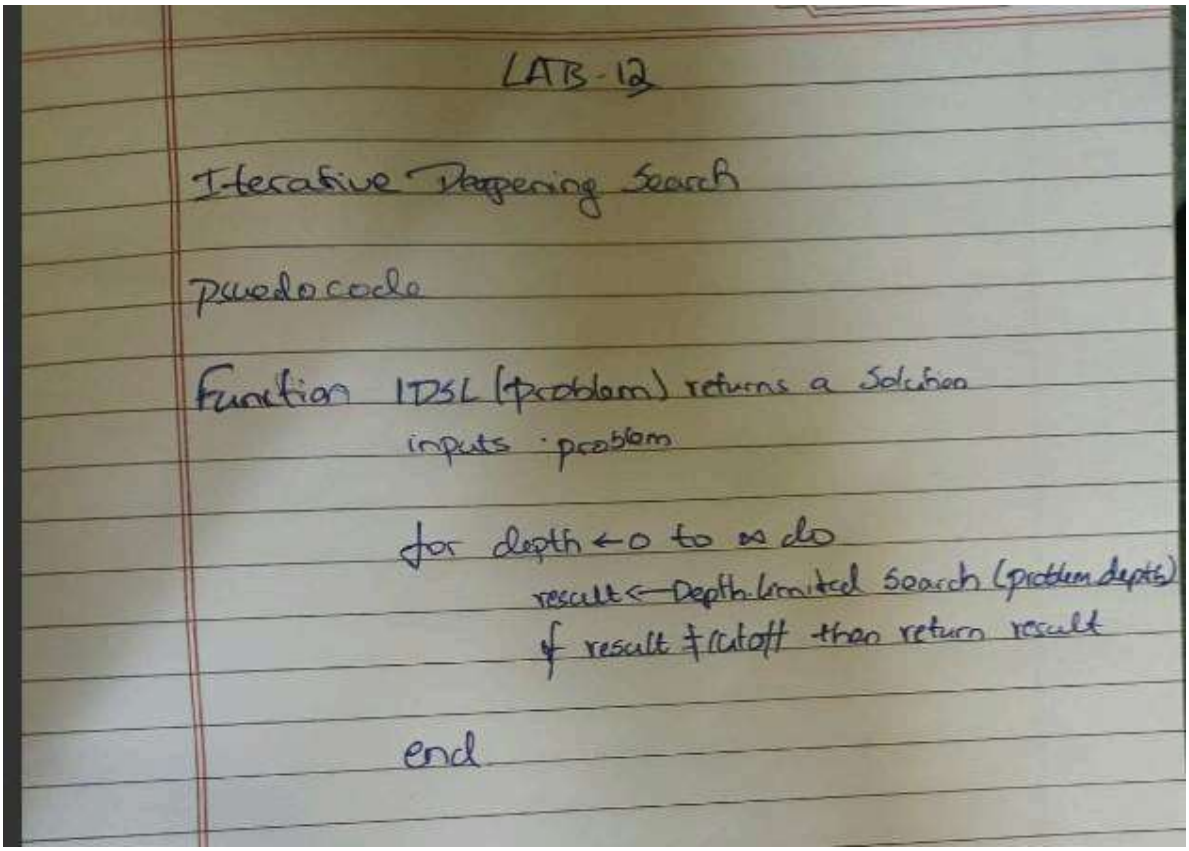
Output

```
Enter the initial state of the puzzle:
Enter row 1 (space-separated numbers, use 0 for empty space): 1 2 3
Enter row 2 (space-separated numbers, use 0 for empty space): 0 5 6
Enter row 3 (space-separated numbers, use 0 for empty space): 4 7 8
Enter the goal state of the puzzle:
Enter row 1 (space-separated numbers, use 0 for empty space): 1 2 3
Enter row 2 (space-separated numbers, use 0 for empty space): 4 5 6
Enter row 3 (space-separated numbers, use 0 for empty space): 7 8 0
Solution found in 431 moves:
```

Program 3

Implement Iterative deepening search algorithm.

Algorithm :



Code :

```
from copy import deepcopy

DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]

class PuzzleState:
    def _init_(self, board, parent=None, move=""):
        self.board = board
self.parent = parent self.move =
        move

    def get_blank_position(self):
        for i in range(3):
for j in range(3):
if self.board[i][j] == 0:
```

```
                    return i, j

    def generate_successors(self):
        successors = []
        x, y = self.get_blank_position()

        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 = deepcopy(self.board)
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
                successors.append(PuzzleState(new_board, parent=self))

        return successors

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

    def _str_(self):
        return "\n".join([" ".join(map(str, row)) for row in self.board])

def depth_limited_search(current_state, goal_state, depth):
    if depth == 0 and current_state.is_goal(goal_state):
        return current_state

    if depth > 0:
        for successor in current_state.generate_successors():
            found = depth_limited_search(successor, goal_state, depth - 1)
            if found:
                return found
    return None

def iterative_deepening_search(start_state, goal_state, max_depth):
    for depth in range(max_depth + 1):
        print(f"\nSearching at depth level: {depth}")
        result = depth_limited_search(start_state, goal_state, depth)
        if result:
            return result
    return None

def get_user_input():
    print("Enter the start state (use 0 for the blank):")
    start_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
```

```
        start_state.append(row)

    print("Enter the goal state (use 0 for the blank):")
    goal_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        goal_state.append(row)

    max_depth = int(input("Enter the maximum depth for search: "))

    return start_state, goal_state, max_depth

def main():
    start_board, goal_board, max_depth = get_user_input()
    start_state = PuzzleState(start_board)
    goal_state = goal_board
    result = iterative_deepening_search(start_state, goal_state,
    max_depth) if result:
        print("\nGoal reached!")
        path = []
        while result:
            path.append(result)
            result = result.parent
        path.reverse()
        for state in path:
            print(state, "\n")
    else:
        print("Goal state not found within the specified depth.")

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

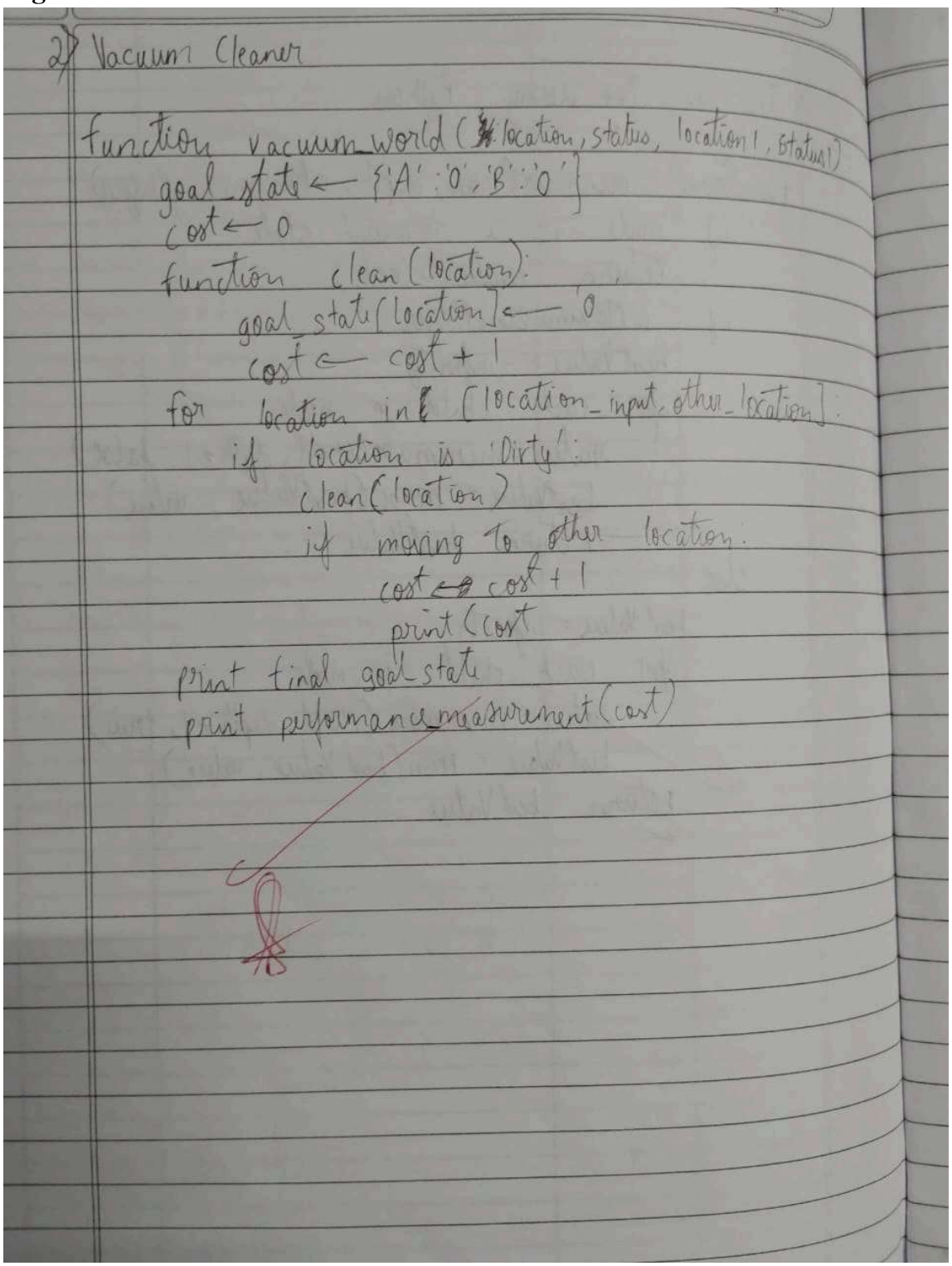
Output :

```
Enter the start state (use 0 for the blank):
2 8 3
1 6 4
7 0 5
Enter the goal state (use 0 for the blank):
1 2 3
8 0 4
7 6 5
Enter the maximum depth for search: 6

Searching at depth level: 0

Searching at depth level: 1

Searching at depth level: 2

Searching at depth level: 3

Searching at depth level: 4

Searching at depth level: 5

Goal reached!
2 8 3
1 6 4
7 0 5

2 8 3
1 0 4
7 6 5

2 0 3
1 8 4
7 6 5

0 2 3
1 8 4
7 6 5

1 2 3
0 8 4
7 6 5

1 2 3
8 0 4
7 6 5

--------------
```

Program 4

Implement a 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))
```

```
# Output
vacuum_world()
```

Output :

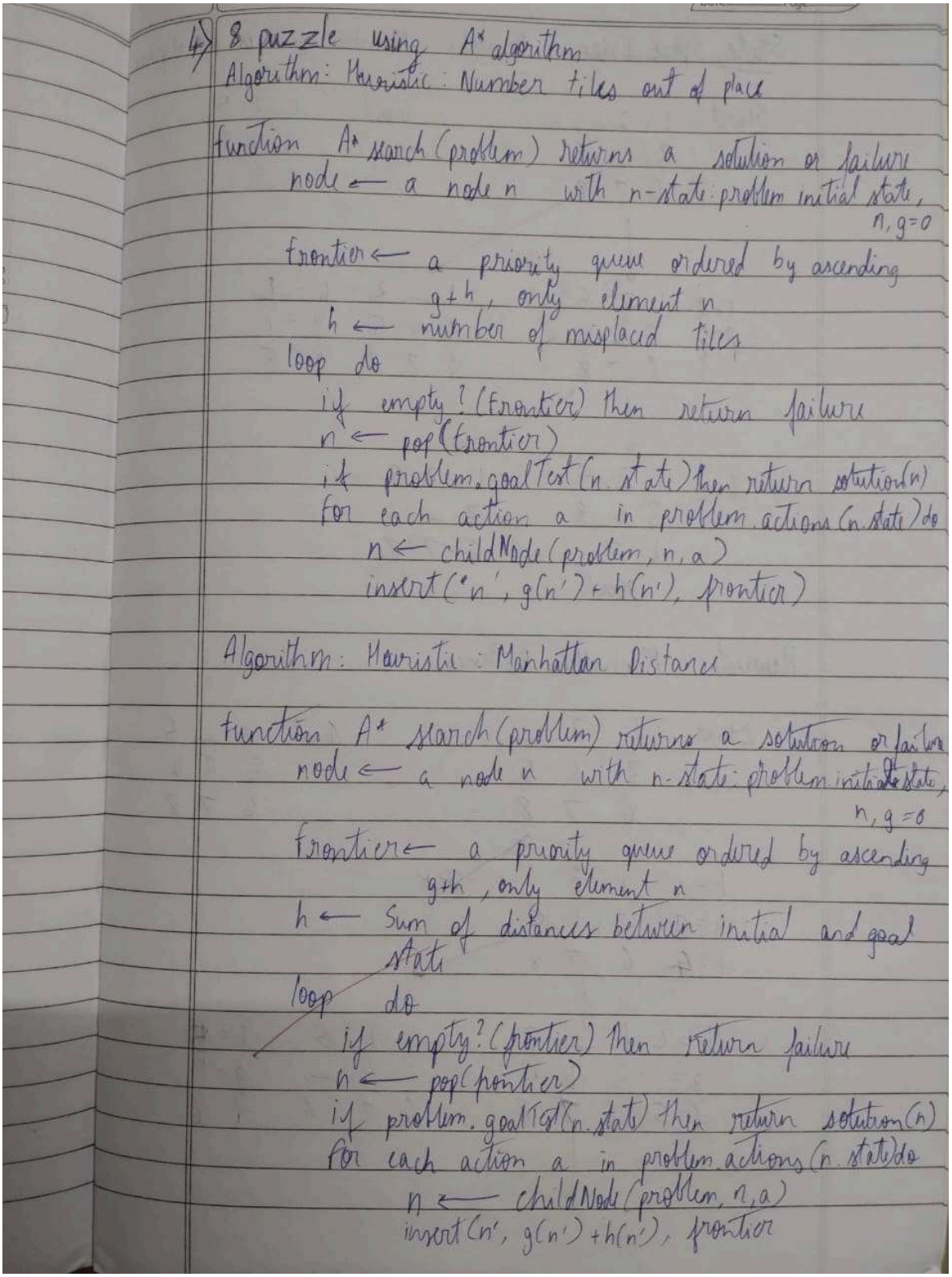
```
Enter Location of Vacuum (A or B): a
Enter status of A (0 for Clean, 1 for Dirty): 1
Enter status of B (0 for Clean, 1 for Dirty): 1
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for cleaning A: 1
Location A has been Cleaned.
Location B is Dirty.
Moving right to Location B.
Cost for moving RIGHT: 2
Cost for suck: 3
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
```

Program 5

Implement A* search algorithm.

1.Using Misplaced tiles

Algorithm :



4) 8 puzzle using A* algorithm
Algorithm: Heuristic: Number tiles out of place

function A* search (problem) returns a solution or failure
node ← a node n with n-state: problem initial state, n, g=0
frontier ← a priority queue ordered by ascending g+h, only element n
h ← number of misplaced tiles
loop do
if empty? (Frontier) then return failure
n ← pop(frontier)
if problem.goalTest(n.state) then return solution(n)
for each action a in problem.actions(n.state) do
n ← childNode(problem, n, a)
insert('n', g(n') + h(n'), frontier)

Algorithm: Heuristic: Manhattan distance

function A* search (problem) returns a solution or failure
node ← a node n with n-state: problem initial state, n, g=0
frontier ← a priority queue ordered by ascending g+h, only element n
h ← Sum of distances between initial and goal state
loop do
if empty? (frontier) then return failure
n ← pop(frontier)
if problem.goalTest(n.state) then return solution(n)
for each action a in problem.actions(n.state) do
n ← childNode(problem, n, a)
insert(n', g(n') + h(n'), frontier

Code :

```
import heapq

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

def misplaced_tiles_heuristic(state, goal_state):
    misplaced_tiles = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != goal_state[i][j] and state[i][j] != 0:
                misplaced_tiles += 1
    return misplaced_tiles

class PuzzleNode:
    def __init__(self, state, parent=None, g=0, h=0):
        self.state = state
        self.parent = parent
        self.g = g  # Cost from start to current node
        self.h = h  # Heuristic cost to goal (misplaced
        tiles) self.f = g + h      # Total cost

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

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

def generate_neighbors(node):
    neighbors = []
    x, y = get_empty_tile(node.state)

    moves = [(0, -1), (0, 1), (-1, 0), (1, 0)]
    for dx, dy in moves:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [row[:] for row in node.state]
            new_state[x][y], new_state[new_x][new_y] =
new_state[new_x][new_y], new_state[x][y]
            h = misplaced_tiles_heuristic(new_state, GOAL_STATE)
            neighbors.append(PuzzleNode(new_state, node, node.g + 1, h))
```

```
    return neighbors
def is_goal_reached(state, goal_state):
    return state == goal_state

def reconstruct_path(node):
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return path[::-1]

def a_star_search(start_state):
    open_set = [PuzzleNode(start_state, h=misplaced_tiles_heuristic(start_state,
GOAL_STATE))]
    closed_set = set()

    while open_set:
        current_node = heapq.heappop(open_set)

        if is_goal_reached(current_node.state, GOAL_STATE):
            return reconstruct_path(current_node)

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

        for neighbor in generate_neighbors(current_node):
            if tuple(map(tuple, neighbor.state)) not in closed_set:
                heapq.heappush(open_set, neighbor)

    return None

def get_user_input():
    while True:
        print("Enter your 8-puzzle configuration (0 represents the empty tile):")
        state = []
        for i in range(3):
            row = input(f"Enter row {i+1}: ").split()
            if len(row) != 3:
                print("Invalid input. Please enter 3 numbers per row.")
                continue
            try:
                row = [int(x) for x in row]
            except ValueError:
                print("Invalid input. Please enter only numbers.")
                continue
```

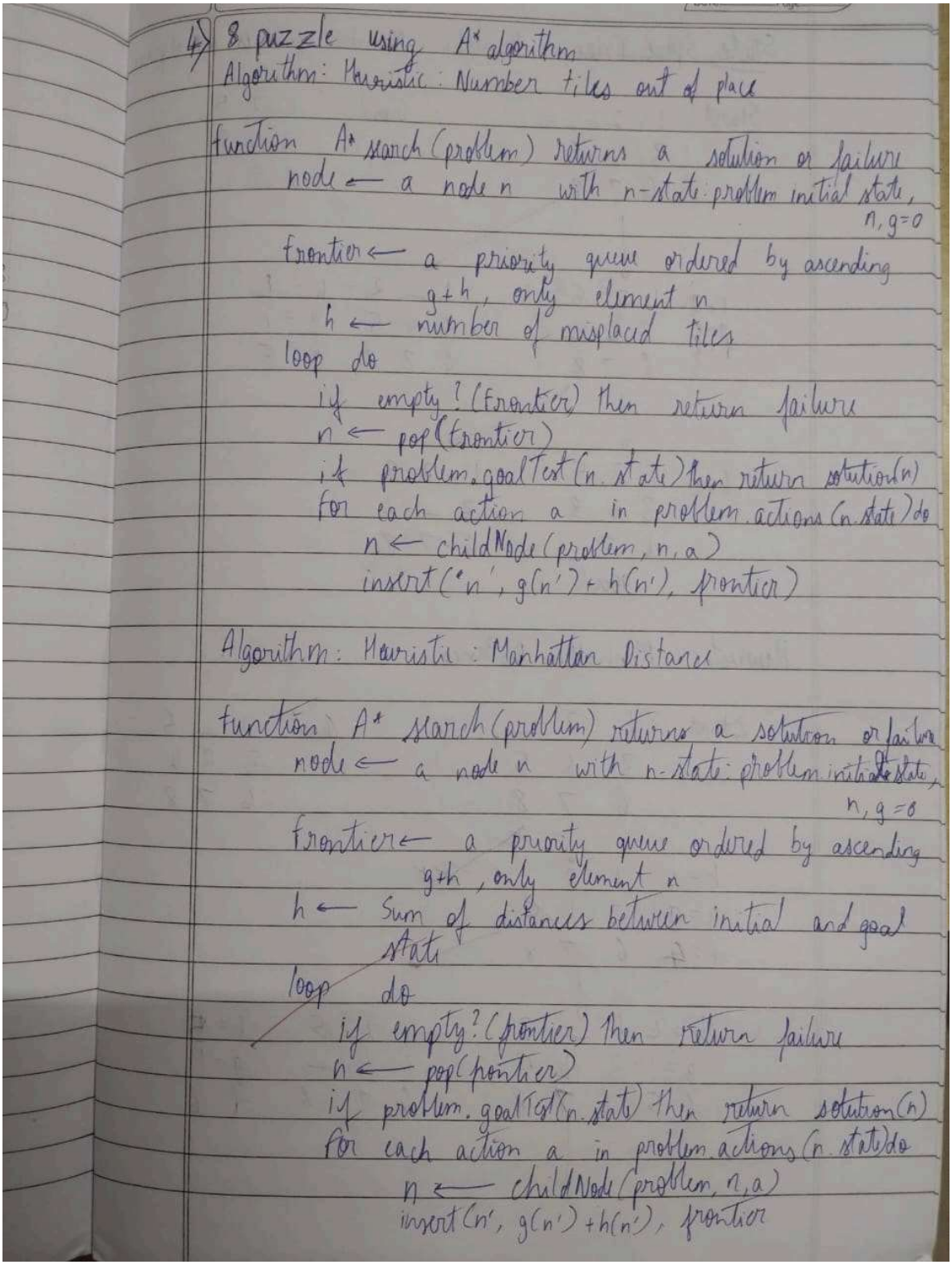
```
            if not all(0 <= x <= 8 for x in row):
                print("Numbers must be between 0 and
                8.") continue
        state.append(row)

        if len(set(sum(state, []))) != 9:
                print("Each number from 0 to 8 must appear exactly once.")
                continue

        return state

def main():
        start_state = get_user_input()
        solution =
a_star_search(start_state)
    if solution:
        print("Solution found in", len(solution) - 1, "steps:")
        for step in solution:
        for row in step:
        print(row)
        print() else:

          print("No solution found.")
        print("Vanith D Ramesh
(1BM22CS319)")
if    name  == "  main  ":
        main()
```

Output

```
Enter your 8-puzzle configuration (0 represents the empty tile):
Enter row 1: 1 2 3
Enter row 2: 0 5 6
Enter row 3: 4 7 8
Solution found in 3 steps:
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]

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

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

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

Using Manhattan distance

Algorithm:



4) 8 puzzle using A* algorithm

Algorithm: Heuristic: Number tiles out of place

function A* search (problem) returns a solution or failure
node ← a node n with n.state: problem.initial state, n.g = 0
frontier ← a priority queue ordered by ascending g+h, only element n
h ← number of misplaced tiles
loop do
if empty?(frontier) then return failure
n ← pop(frontier)
if problem.goalTest(n.state) then return solution(n)
for each action a in problem.actions(n.state) do
n ← childNode(problem, n, a)
insert('n', g(n') + h(n'), frontier)

Algorithm: Heuristic: Manhattan Distance

function A* search (problem) returns a solution or failure
node ← a node n with n.state: problem.initial state, n.g = 0
frontier ← a priority queue ordered by ascending g+h, only element n
h ← Sum of distances between initial and goal state
loop do
if empty?(frontier) then return failure
n ← pop(frontier)
if problem.goalTest(n.state) then return solution(n)
for each action a in problem.actions(n.state) do
n ← childNode(problem, n, a)
insert(n', g(n') + h(n'), frontier

Code:

```
import heapq

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

MOVES = [(0, -1), (0, 1), (-1, 0), (1, 0)]

class PuzzleNode:
    def __init__(self, board, parent=None, g=0,
        h=0): self.board = board
        self.parent = parent
        self.g = g
        self.h = h
        self.f = g + h

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

def manhattan_distance(board):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = board[i][j]
            if value != 0:
                goal_x, goal_y = (value - 1) // 3, (value - 1) % 3
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance

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

def generate_neighbors(node):
    neighbors = []
    x, y = get_empty_tile(node.board)

    for dx, dy in MOVES:
        new_x, new_y = x + dx, y + dy
```

```
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_board = [row[:] for row in
            node.board]
            new_board[x][y], new_board[new_x][new_y] =
new_board[new_x][new_y], new_board[x][y]
            h = manhattan_distance(new_board)
            neighbors.append(PuzzleNode(new_board, node, node.g + 1, h))

    return neighbors

def is_goal_reached(board):
    return board == GOAL_BOARD

def reconstruct_path(node):
    path = []
    while node:
        path.append(node.board)
        node = node.parent
    return path[::-1]

def a_star_search(start_board):
    open_set = [PuzzleNode(start_board)]
    closed_set = set()

    while open_set:
        current_node = heapq.heappop(open_set)

        if is_goal_reached(current_node.board):
            return reconstruct_path(current_node)

        closed_set.add(tuple(map(tuple, current_node.board)))

        for neighbor in generate_neighbors(current_node):
            if tuple(map(tuple, neighbor.board)) not in closed_set:
                heapq.heappush(open_set, neighbor)

    return None

def get_user_input():
    while True:
        print("Enter your 8-puzzle configuration (0 represents the empty tile):")
        state = []
        for i in range(3):
            row = input(f"Enter row {i+1}: ").split()
            if len(row) != 3:
                print("Invalid input. Please enter 3 numbers per row.")
                continue
```

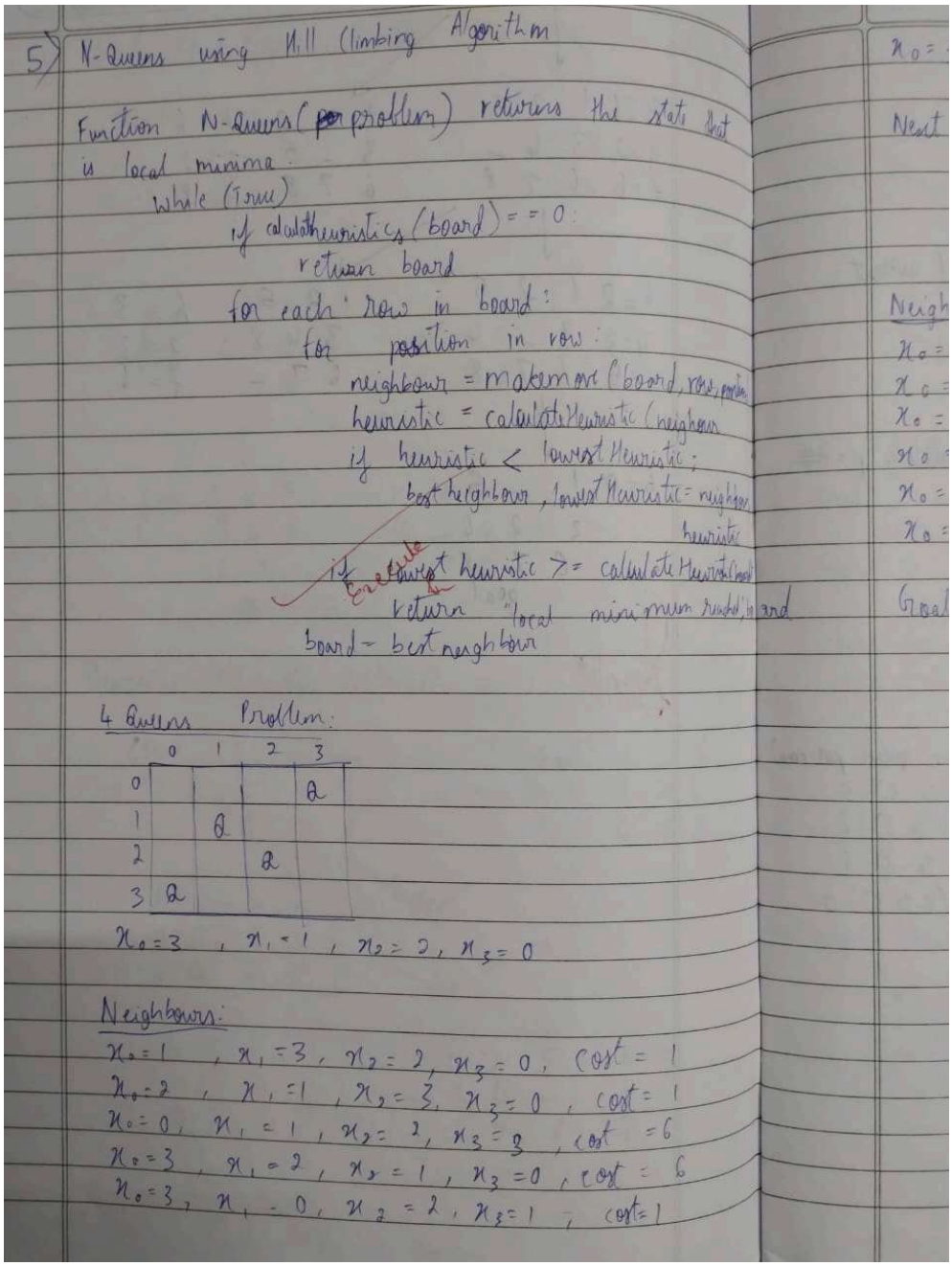
```
            try:
                row = [int(x) for x in row]
            except ValueError:
                print("Invalid input. Please enter only numbers.")
                continue
            if not all(0 <= x <= 8 for x in row):
                print("Numbers must be between 0 and 8.")
                continue
            state.append(row)

        if len(set(sum(state, []))) != 9:
            print("Each number from 0 to 8 must appear exactly
            once.") continue

        return state

def main():
    start_state = get_user_input()
    solution = a_star_search(start_state)

    if solution:
        print("Solution found in", len(solution) - 1, "steps:")
        for step in solution:
            for row in step:
                print(row)

    print() else:
        print("No solution found.")

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

Output:

```
Enter your 8-puzzle configuration (0 represents the empty tile):
Enter row 1: 1 2 3
Enter row 2: 0 5 6
Enter row 3: 4 7 8
Solution found in 3 steps:
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]

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

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

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

2.Implement Hill Climbing Algorithm.

Algorithm :



5) N-Queens using Hill Climbing Algorithm

Function N-Queens(problem) returns the state that is local minima:
  while (True)
    if calculateHeuristics(board) == 0:
      return board
    for each row in board:
      for position in row:
        neighbour = makemove(board, row, position)
        heuristic = calculateHeuristic(neighbour)
        if heuristic < lowest Heuristic:
          best neighbour, lowest Heuristic = neighbour, heuristic
    if lowest heuristic >= calculate Heuristic(board):
      return "local minimum reached" and board
    board = best neighbour

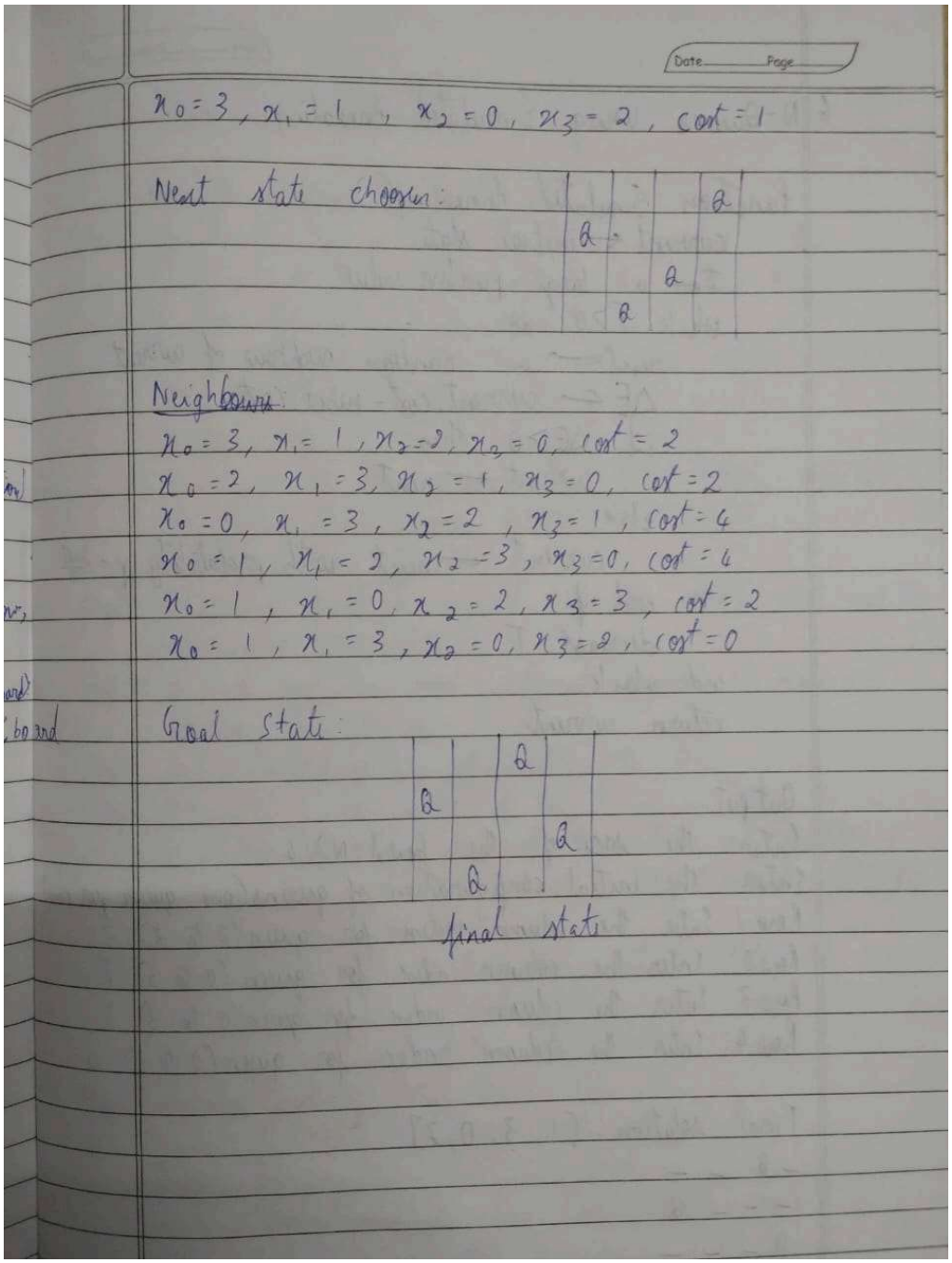
4 Queens Problem:

	0	1	2	3
0				Q
1		Q		
2			Q	
3	Q			

$x_0 = 3$, $x_1 = 1$, $x_2 = 2$, $x_3 = 0$

Neighbours:

$x_0 = 1$, $x_1 = 3$, $x_2 = 2$, $x_3 = 0$, cost = 1
$x_0 = 2$, $x_1 = 1$, $x_2 = 3$, $x_3 = 0$, cost = 1
$x_0 = 0$, $x_1 = 1$, $x_2 = 2$, $x_3 = 3$, cost = 6
$x_0 = 3$, $x_1 = 2$, $x_2 = 1$, $x_3 = 0$, cost = 6
$x_0 = 3$, $x_1 = 0$, $x_2 = 2$, $x_3 = 1$, cost = 1



Code :

```
import random

def hill_climbing_n_queens(n):
    board = generate_random_board(n)
```

```
    while True:
        current_cost = calculate_conflicts(board)
        if current_cost == 0:
            return board
        next_board, next_cost = get_best_neighbor(board)
        if next_cost >= current_cost:
            board = generate_random_board(n)
        else:

            board =

next_board def

generate_random_board(n):

    return [random.randint(0, n - 1) for _ in

range(n)] def calculate_conflicts(board):

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

def get_best_neighbor(board):
    n = len(board)
    best_board = board[:]
    best_cost = calculate_conflicts(board)
    for col in range(n):
        original_row = board[col]
        for row in range(n):
            if row != original_row:
                board[col] = row
                cost = calculate_conflicts(board)

                if cost < best_cost:
                    best_cost = cost
                    best_board =

        board[:] board[col] =

        original_row
```

```
    return best_board, best_cost

n=int(input("No of queens: "))
solution = hill_climbing_n_queens(n)
print("Solution for", n, "queens:")
print(solution)
```

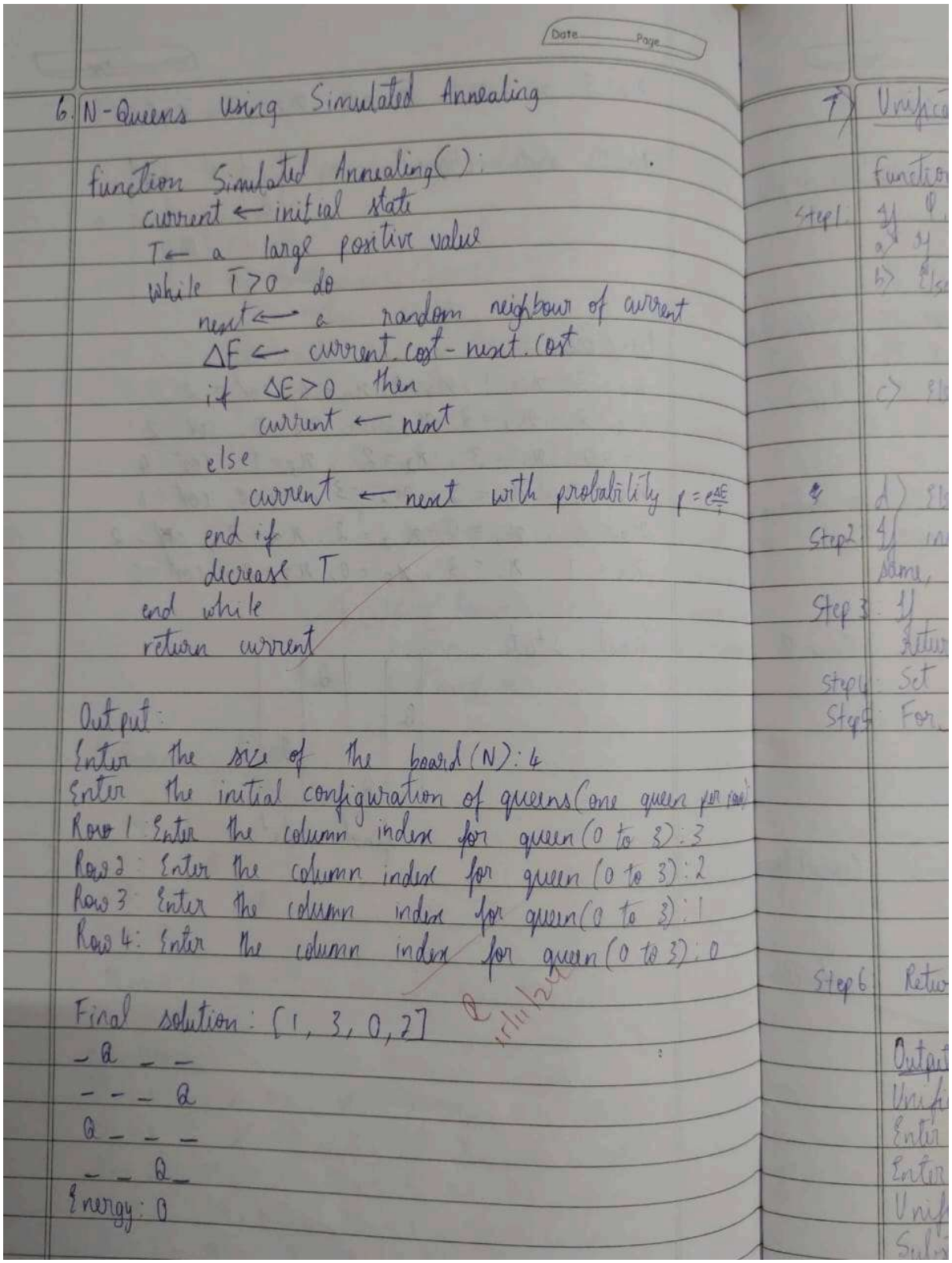
Output :

```
No of queens: 4
Solution for 4 queens:
[2, 0, 3, 1]
```

Program 6

Write a program to implement Simulated Annealing Algorithm

Algorithm :



6. N-Queens using Simulated Annealing

```
function SimulatedAnnealing():
    current ← initial state
    T ← a large positive value
    while T > 0 do
        next ← a random neighbour of current
        ΔE ← current.cost - next.cost
        if ΔE > 0 then
            current ← next
        else
            current ← next with probability p = e^(ΔE/T)
        end if
        decrease T
    end while
    return current
```

Output:

```
Enter the size of the board (N): 4
Enter the initial configuration of queens (one queen per row)
Row 1: Enter the column index for queen (0 to 3): 3
Row 2: Enter the column index for queen (0 to 3): 2
Row 3: Enter the column index for queen (0 to 3): 1
Row 4: Enter the column index for queen (0 to 3): 0

Final solution: [1, 3, 0, 2]
- Q - -
- - - Q
Q - - -
- - Q -
Energy: 0
```

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:
            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.")
```

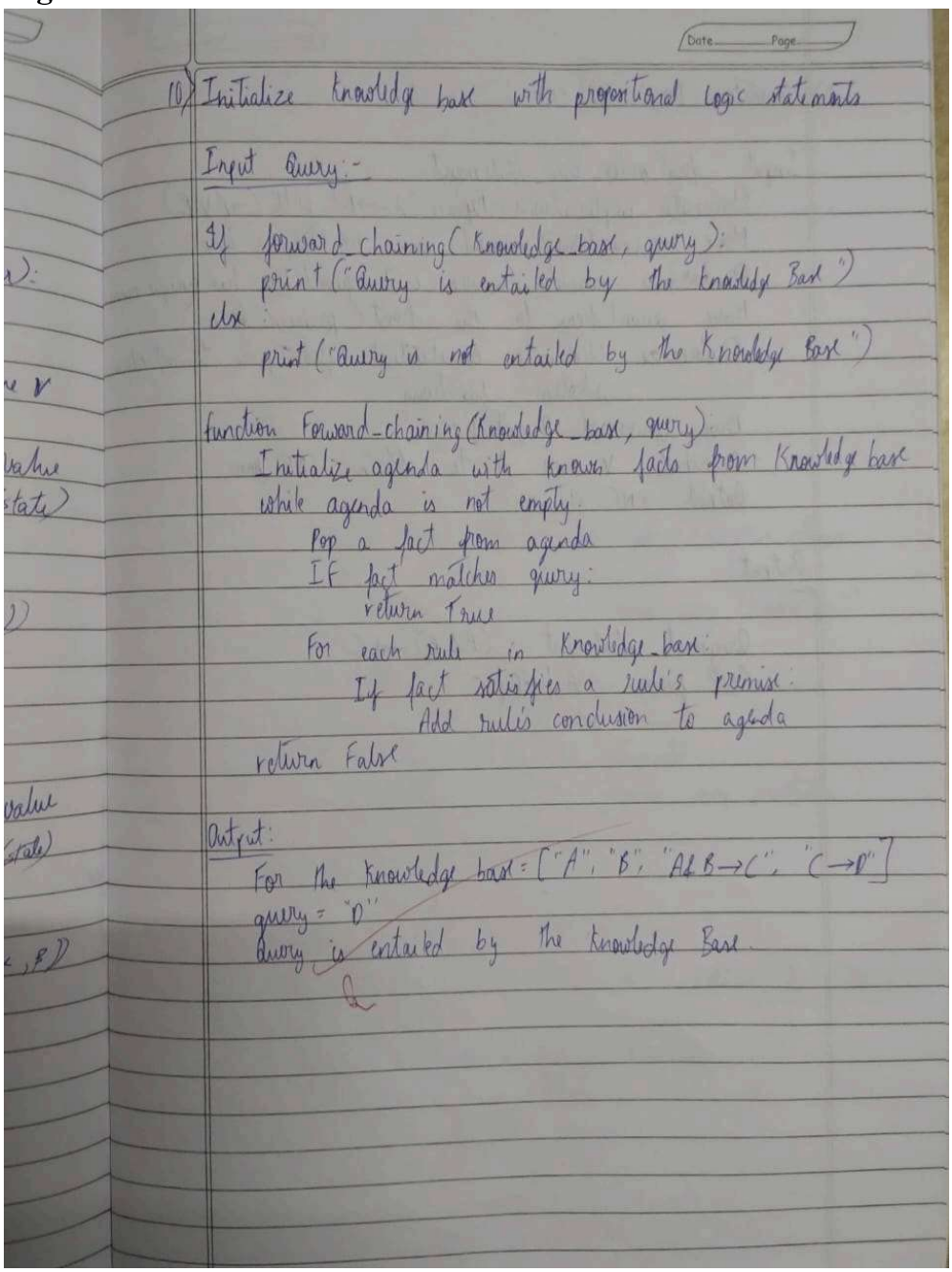
Output:

```
Enter the column positions for the queens (space-separated integers between 0 and 7): 0 1 2 3 4 5 6 7
Q . . . . . . .
. Q . . . . . .
. . Q . . . . .
. . . Q . . . .
. . . . Q . . .
. . . . . Q . .
. . . . . . Q .
. . . . . . . Q
Solution found!
. . . . . Q . .
. . Q . . . . .
Q . . . . . . .
. . . . . . . Q
. . . . Q . . .
. Q . . . . . .
. . . Q . . . .
. . . . . . Q .
```

Program 7

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

Algorithm :



Code :

```
from itertools import product
```

```
# Define a function to evaluate a propositional expression
def evaluate(expr, model):
    """
    Evaluates the given expression based on the values in the model.
    """
    for var, val in model.items():
        expr = expr.replace(var, str(val))
    return eval(expr)

# Define the truth-table enumeration algorithm
def truth_table_entails(KB, query, symbols):
    """
    Checks if KB entails query using truth-table enumeration.
    KB: list of propositional expressions (strings)
    query: propositional expression (string)
    symbols: list of symbols (propositions) in the KB and
    query """
    # Generate all possible truth assignments
    assignments = list(product([False, True], repeat=len(symbols)))

    entailing_models = []

    # Iterate over each assignment to check
    entailment for assignment in assignments:
        model = dict(zip(symbols, assignment))

        # Check if KB is true in this model
        KB_is_true = all(evaluate(expr, model) for expr in KB)

        # If KB is true, check if query is also
        true if KB_is_true:
            query_is_true = evaluate(query, model)
            if query_is_true:
                entailing_models.append(model)  # Store the
            model else:
                return False, []
               # Found a model where KB is true but query is false

    return True, entailing_models  # KB entails query if no counterexample
was found

# Get input from the user
symbols = input("Enter the propositions (symbols) separated by spaces: ").split()
KB = []
n = int(input("Enter the number of statements in the knowledge base: "))
```

```
for i in range(n):
expr = input(f"Enter statement {i + 1} in the knowledge base: ") KB.append(expr)

query = input("Enter the query: ")

# Check entailment
result, models = truth_table_entails(KB, query, symbols)
if truth_table_entails(KB, query, symbols):
print("KB entails the query.")
print("Models where KB entails query:") for
    model in models:
print(model)
else:
print("KB does not entail the query.")
```

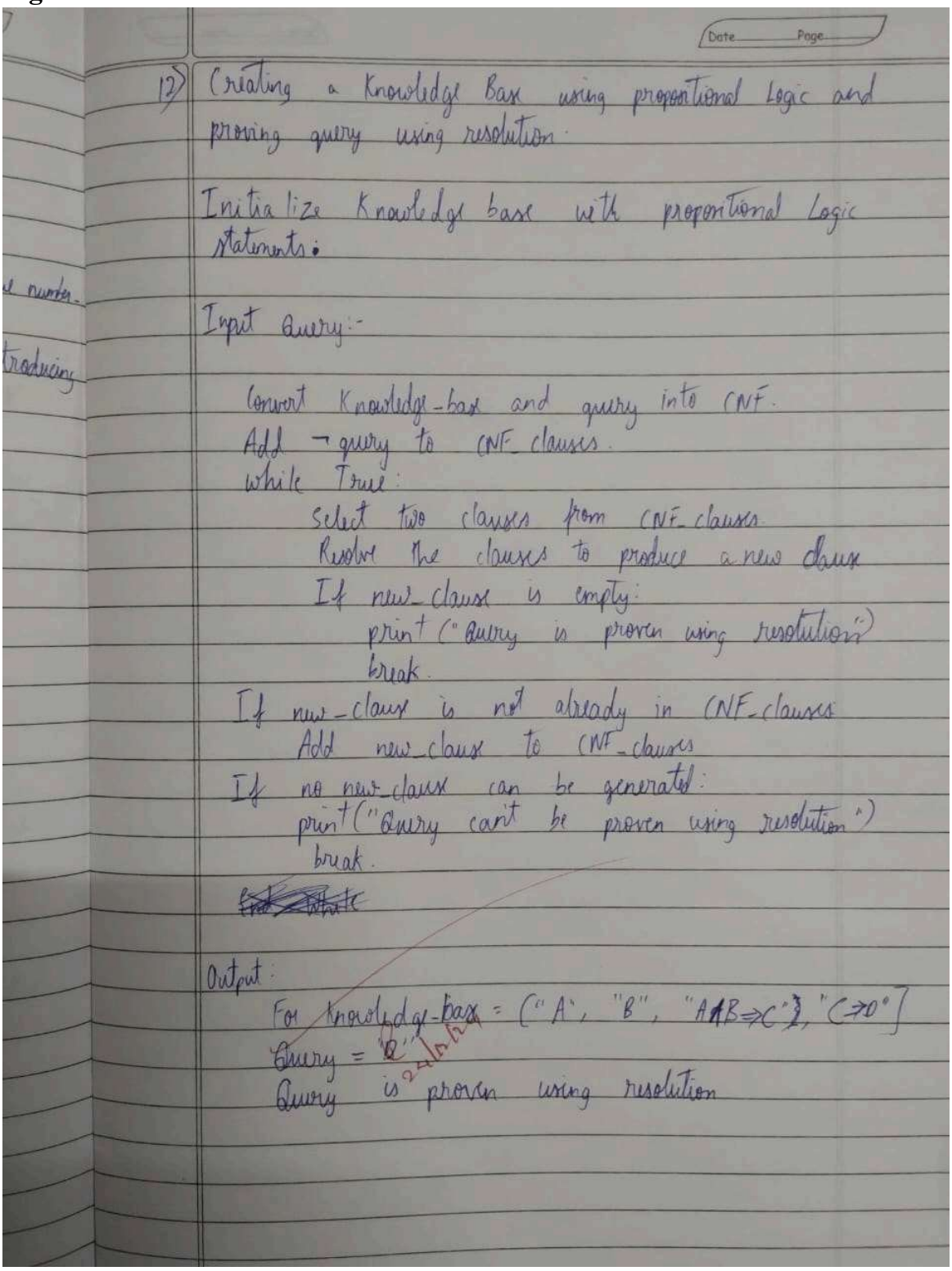
Output :

```
Enter the propositions (symbols) separated by spaces: A B C
Enter the number of statements in the knowledge base: 2
Enter statement 1 in the knowledge base: A or C
Enter statement 2 in the knowledge base: B or not C
Enter the query: A or B
KB entails the query.
Models where KB entails query:
{'A': False, 'B': True, 'C': True}
{'A': True, 'B': False, 'C': False}
{'A': True, 'B': True, 'C': False}
{'A': True, 'B': True, 'C': True}
```

Program 8

Create a knowledge base using propositional logic and prove the given query using resolution.

Algorithm :



12) Creating a Knowledge Base using propositional Logic and proving query using resolution.

Initialize Knowledge base with propositional Logic statements:

Input Query:-

Convert Knowledge-base and query into CNF.
Add ¬ query to CNF_clauses.
while True:
    Select two clauses from CNF_clauses
    Resolve the clauses to produce a new clause
    If new_clause is empty:
        print("Query is proven using resolution")
        break.
If new_clause is not already in CNF_clauses
    Add new_clause to CNF_clauses
If no new_clause can be generated:
    print("Query can't be proven using resolution")
    break.

Output:
For Knowledge-base = ["A", "B", "A∧B⇒C"], "C⇒D"]
Query = "D"
Query is proven using resolution

Code :

```
kb = [
    {"¬B", "¬C", "A"},  # ¬B ∨ ¬C ∨ A
    {"B"},              # B
    {"¬D", "¬E", "C"},  # ¬D ∨ ¬E ∨ C
    {"E", "F"},         # E ∨ F
    {"D"},              # D
    {"¬F"},             # ¬F
]

# Negate the query: If the query is "A", we negate it to "¬A"
def negate_query(query):
    if "¬" in query:
        return query.replace("¬", "")  # If it's negated, remove the negation
    else:
        return f"¬{query}"  # Otherwise, add negation in front

# Function to perform resolution on two clauses
def resolve(clause1, clause2):
    resolved_clauses = []
    for literal1 in clause1:
        # Try to find complementary literals for literal1 in clause1
        for literal2 in clause2:
            # If literals are complementary (e.g., "A" and "¬A"), resolve them
            if literal1 == f"¬{literal2}" or f"¬{literal1}" == literal2:
                new_clause = (clause1 | clause2) - {literal1, literal2}
                resolved_clauses.append(new_clause)
    return resolved_clauses

# Perform resolution-based proof
def resolution(kb, query):
    # Step 1: Negate the query and add it to the knowledge base
    negated_query = negate_query(query)
    kb.append({negated_query})

    # Step 2: Initialize the set of clauses
    new_clauses = set(frozenset(clause) for clause in kb)

    while True:
        resolved_this_round = set()
        clauses_list = list(new_clauses)

        # Try to resolve every pair of clauses
        for i in range(len(clauses_list)):
            for j in range(i + 1, len(clauses_list)):
                clause1 = clauses_list[i]
                clause2 = clauses_list[j]

                # Apply resolution to the two clauses
                resolved = resolve(clause1, clause2)
                if frozenset() in resolved:
                    return True  # Found an empty clause (contradiction), query
is provable
                resolved_this_round.update(resolved)

        # If no new clauses were added, stop
        if resolved_this_round.issubset(new_clauses):
            return False  # No new clauses, query is not provable

        # Add new resolved clauses to the set
```

```
        new_clauses.update(resolved_this_round)

# Query to prove: "A"
print("Using Resolution to prove a query")
query = input("Enter the query: ")
result = resolution(kb, query)
print(f"Is the query '{query}' provable? {'Yes' if result else 'No'}")
```

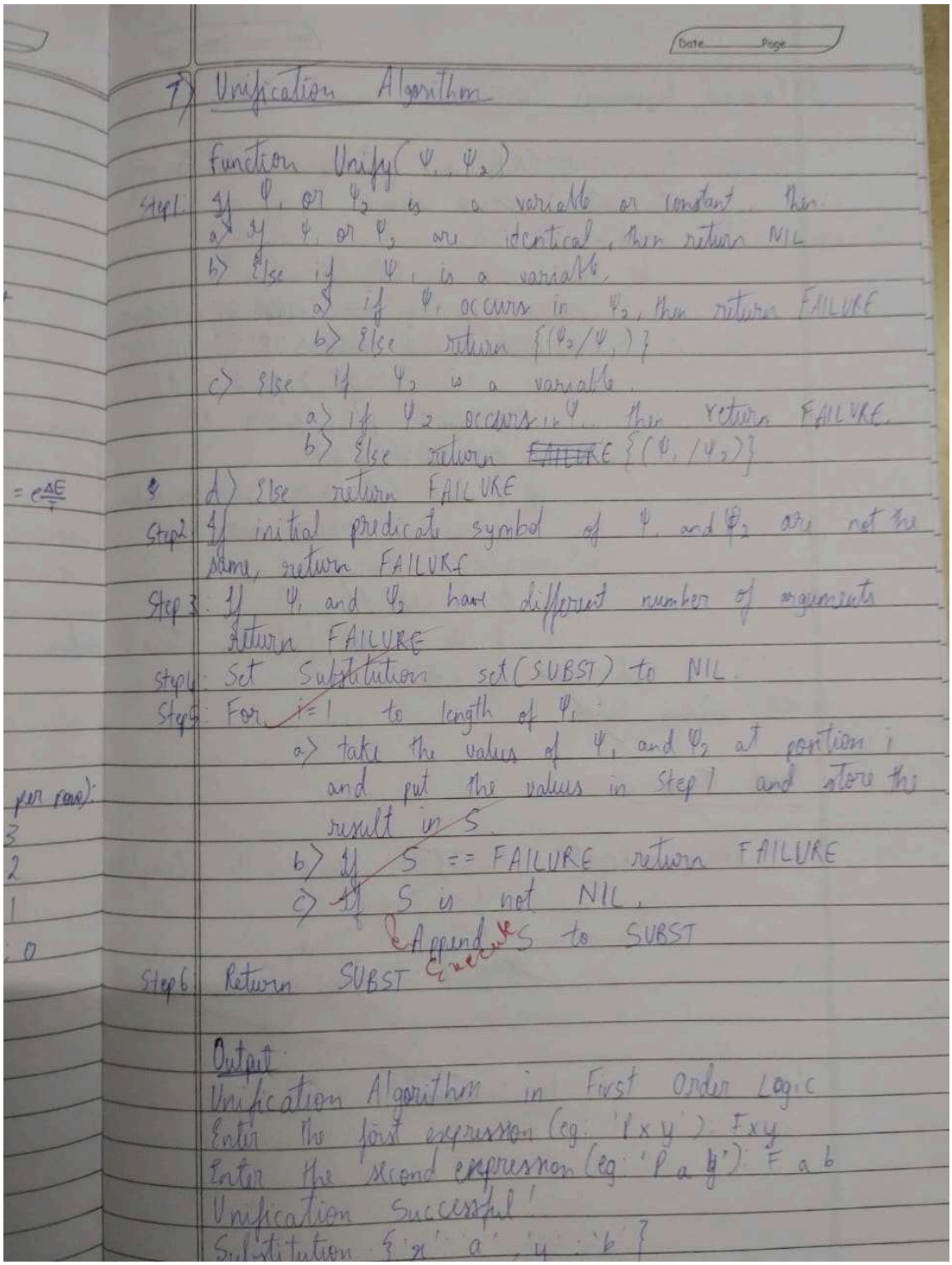
Output :

```
Enter the query: A
Using Resolution to prove a query
Is the query 'A' provable? Yes
```

Program 9

Implement unification in first order logic.

Algorithm :



Code :

```
def unify(s1, s2, theta={}):
    if theta is None:
        return None

    if s1 == s2:
        return theta

    if isinstance(s1, str) and
        s1.islower(): return unify_var(s1,
        s2, theta)

    if isinstance(s2, str) and s2.islower():
        return unify_var(s2, s1, theta)

    if isinstance(s1, tuple) and isinstance(s2, tuple) and len(s1) ==
        len(s2): return unify(s1[1:], s2[1:], unify(s1[0], s2[0], theta))

    return None

def unify_var(var, x, theta):
    if var in theta:
        return unify(theta[var], x,
    theta) elif x in theta:
        return unify(var, theta[x], theta)
    elif occurs_check(var, x, theta):
        return None
    else:
        theta[var] = x
        return theta

def occurs_check(var, x, theta):
```

```
    if var == x:
        return True
    elif isinstance(x, str) and x.islower() and x in theta:
        return occurs_check(var, theta[x], theta)
    elif isinstance(x, tuple):
        for arg in x:
            if occurs_check(var, arg, theta):
                return True
        return False

s1 = ('p', 'x', ('f', 'x'), ('y'))
s2 = ('p', 'a', 'y', ('f', 'x'))

substitution = unify(s1, s2)

if substitution:
    print("Unification successful:")
    print(f"Substitution:
else{substitution}")
    print("Unification
failed.")
```

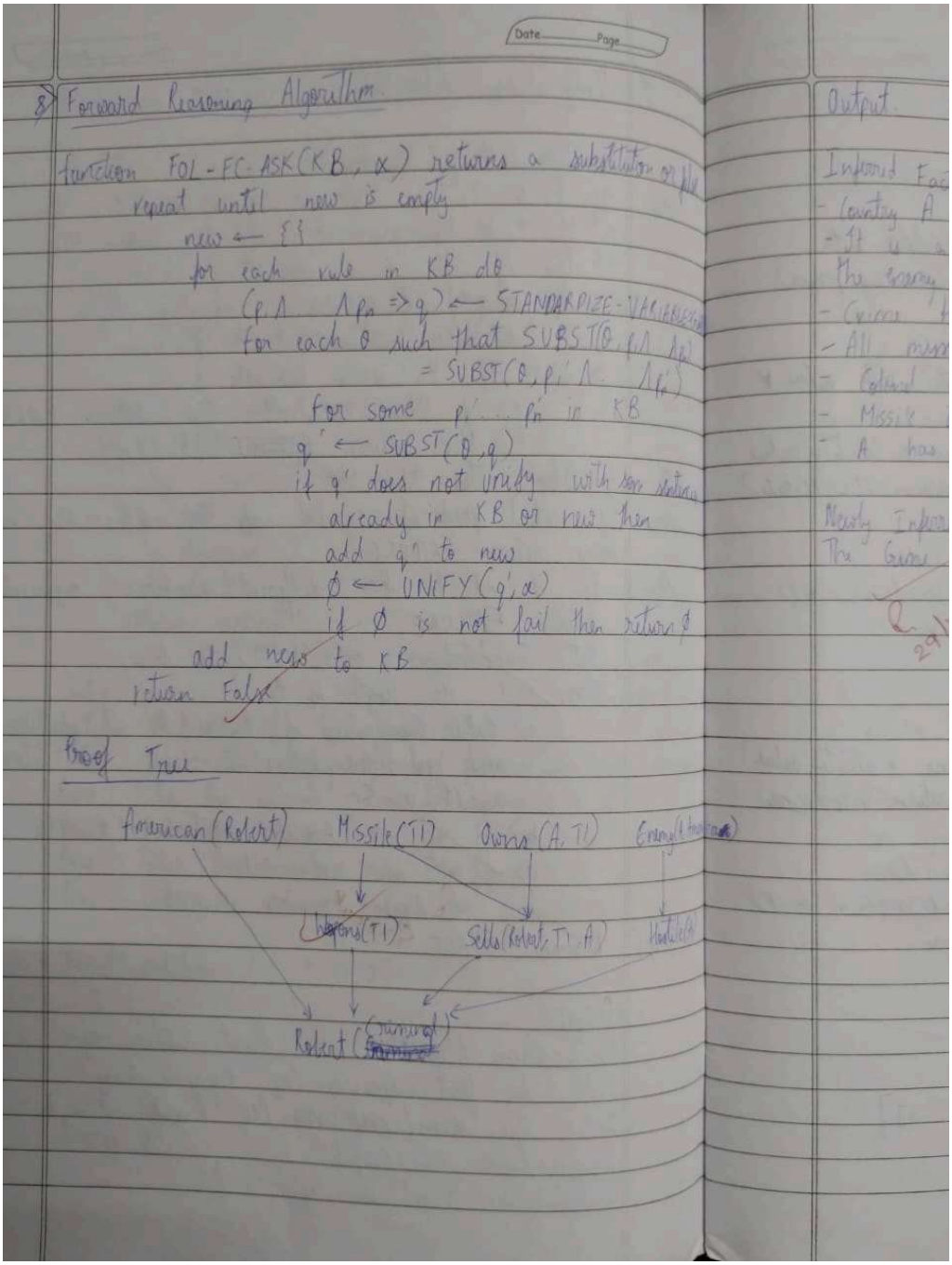
Output:

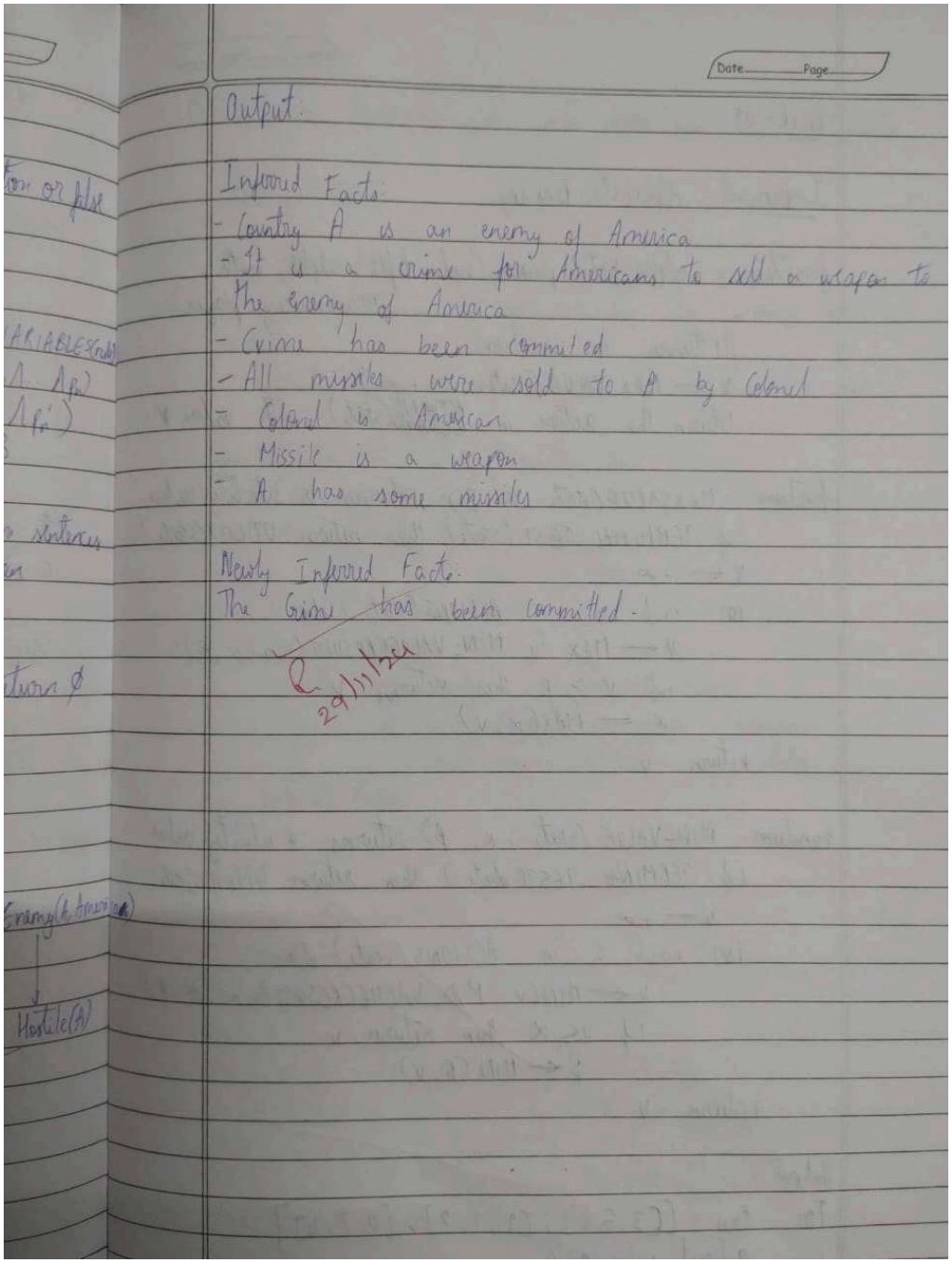
```
Unification successful:
Substitution: {'x': 'a', 'y': ('f', 'x')}
```

Program 10

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

Algorithm :





Code :

```
# Define the knowledge base as a list of rules and facts
class KnowledgeBase:
    def _init_(self):
self.facts = set()      # Set of known facts
        self.rules = []      # List of rules

def add_fact(self, fact):
```

```
        self.facts.add(fact)

    def add_rule(self, rule):
        self.rules.append(rule)

    def infer(self):
        inferred = True
        while inferred:
            inferred = False
            for rule in self.rules:
                if rule.apply(self.facts):
                    inferred = True

# Define the Rule class
class Rule:
    def _init_(self, premises, conclusion):
        self.premises = premises  # List of conditions
        self.conclusion = conclusion  # Conclusion to add if premises are met

    def apply(self, facts):
        if all(premise in facts for premise in
            self.premises): if self.conclusion not in facts:
                facts.add(self.conclusion)
                print(f"Inferred:
                {self.conclusion}") return True
        return False

# Initialize the knowledge base
kb = KnowledgeBase()

# Facts in the problem
kb.add_fact("American(Robert)")
kb.add_fact("Missile(T1)")
kb.add_fact("Owns(A, T1)")
kb.add_fact("Enemy(A, America)")

# Rules based on the problem
# 1. Missile(x) implies Weapon(x)
kb.add_rule(Rule(["Missile(T1)"], "Weapon(T1)"))

# 2. Enemy(x, America) implies Hostile(x)
kb.add_rule(Rule(["Enemy(A, America)"], "Hostile(A)"))

# 3. Missile(x) and Owns(A, x) imply Sells(Robert, x, A)
kb.add_rule(Rule(["Missile(T1)", "Owns(A, T1)"], "Sells(Robert, T1, A)"))

# 4. American(p) and Weapon(q) and Sells(p, q, r) and Hostile(r) imply Criminal(p)
```

```
kb.add_rule(Rule(["American(Robert)", "Weapon(T1)", "Sells(Robert, T1, A)",
"Hostile(A)"], "Criminal(Robert)"))

# Infer new facts based on the rules
kb.infer()

# Check if Robert is a criminal
if "Criminal(Robert)" in kb.facts:
    print("Conclusion: Robert is a criminal.")
else:
    print("Conclusion: Unable to prove Robert is a criminal.")
```

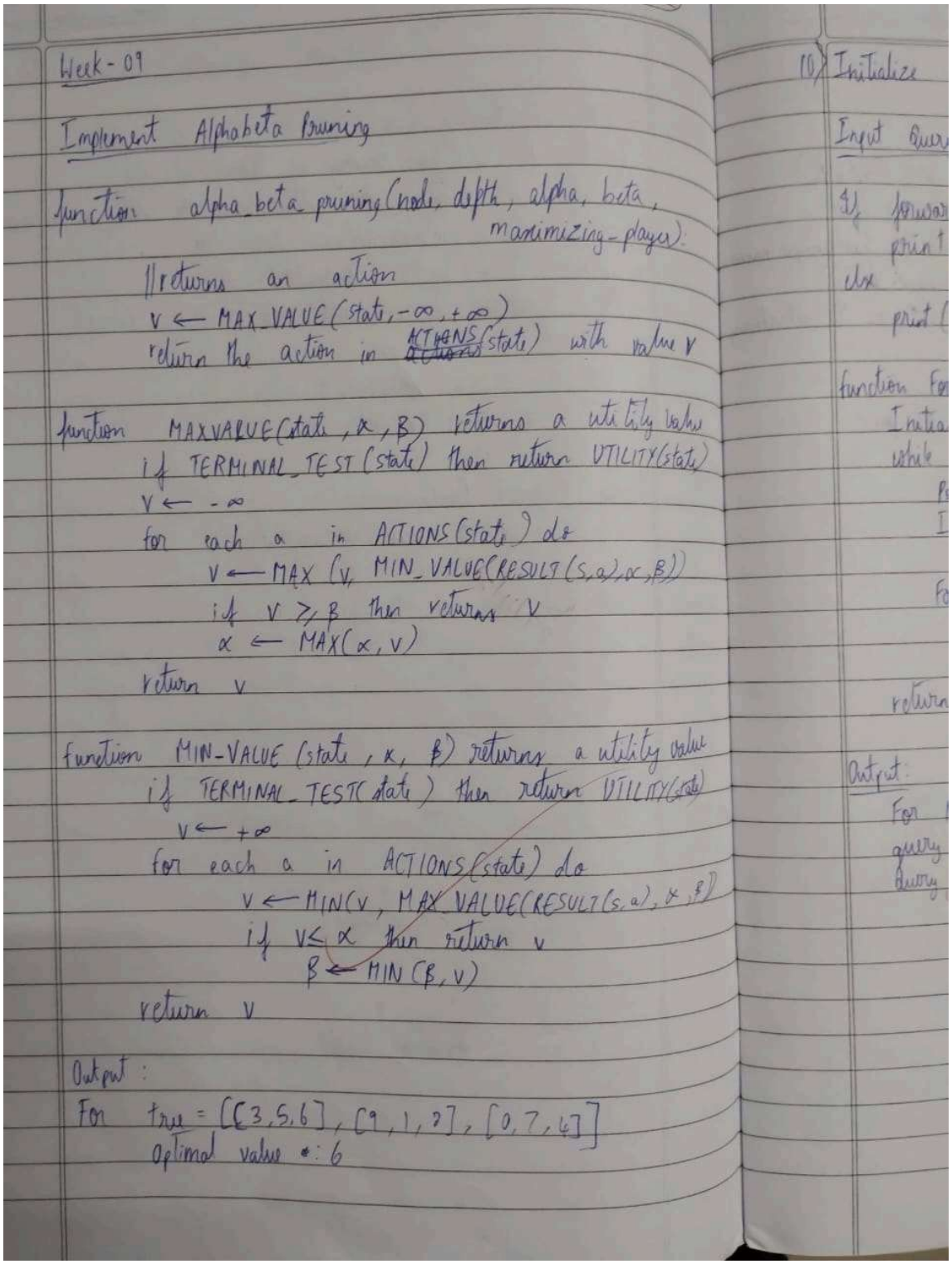
Output :

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

Program 11

Implement Alpha-Beta Pruning.

Algorithm :



Week - 09

Implement Alphabeta Pruning

function alpha_beta_pruning(node, depth, alpha, beta, maximizing_player):

//returns an action

v ← MAX_VALUE(state, -∞, +∞)

return the action in ACTIONS(state) with value v

function MAXVALUE(state, α, β) returns a utility value

if TERMINAL_TEST(state) then return UTILITY(state)

v ← -∞

for each a in ACTIONS(state) do

v ← MAX(v, MIN_VALUE(RESULT(s,a), α, β))

if v ≥ β then return v

α ← MAX(α, v)

return v

function MIN-VALUE(state, α, β) returns a utility value

if TERMINAL_TEST(state) then return UTILITY(state)

v ← +∞

for each a in ACTIONS(state) do

v ← MIN(v, MAX_VALUE(RESULT(s, a), α, β))

if v ≤ α then return v

β ← MIN(β, v)

return v

Output :

For true = [[3,5,6], [9,1,2], [0,7,4]]

Optimal value : 6

Code :

```
MAX, MIN = 1000, -1000

def minimax(depth, nodeIndex, maximizingPlayer,
      values, alpha, beta):

  if depth == 3:
    return values[nodeIndex]

  if maximizingPlayer:

    best = MIN

    # Recur for left and right children
    for i in range(0, 2):

      val = minimax(depth + 1, nodeIndex * 2 + i,
            False, values, alpha, beta)
      best = max(best, val)
      alpha = max(alpha,
      best)

      # Alpha Beta Pruning
      if beta <= alpha:
        break
    return best
  else:
    best = MAX

    # Recur for left and
    # right children
    for i in range(0, 2):

      val = minimax(depth + 1, nodeIndex * 2 + i,
              True, values, alpha, beta)
      best = min(best, val)
      beta = min(beta, best)

      # Alpha Beta Pruning
      if beta <= alpha:
        break

    return best
```

```
# Driver Code
if _name    == "_main_":

  values = [3, 5, 6, 9, 1, 2, 0, -1]
  print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

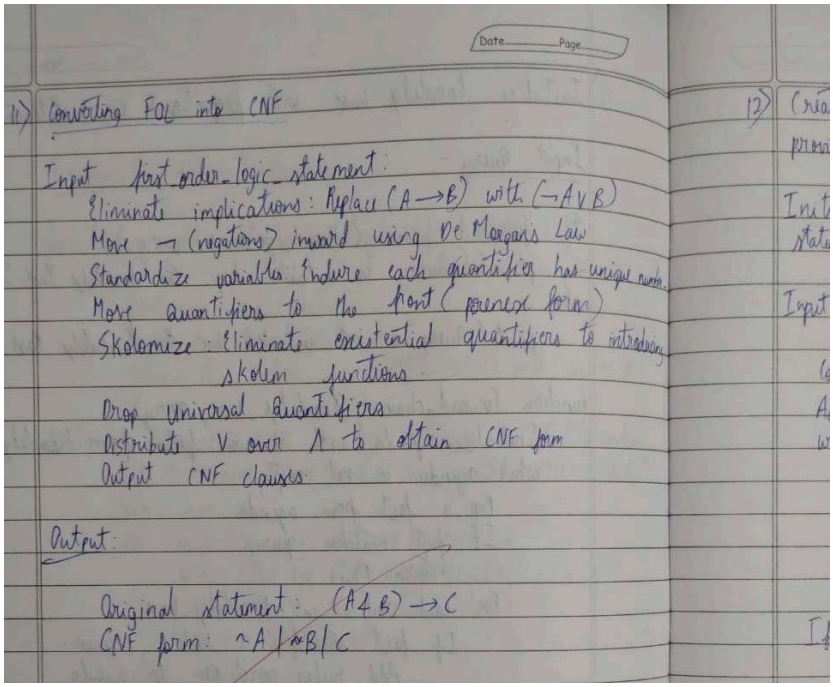
Output :

```
The optimal value is : 5
---------
```

Program 12

CONVERTING FOL TO CNF:

ALGORITHM:



CODE:

```
from sympy import symbols, Not, Or, And, Implies, Equivalent
from sympy.logic.boolalg import to_cnf

def fol_to_cnf(fol_expr):
      fol_expr = fol_expr.replace(Equivalent, lambda a, b: And(Implies(a,
b),Implies(b, a)))
    fol_expr = fol_expr.replace(Implies, lambda a, b: Or(Not(a), b))
      cnf_form = to_cnf(fol_expr, simplify=True)
      return cnf_form
```

```
def main():
      P = symbols("P")
    Q = symbols("Q")
    R = symbols("R")

      fol_expr1 = Implies(P, Q)
      print("Example 1: P → Q")
      print("Original FOL
Expression:")
      print(fol_expr1)

    cnf1 = fol_to_cnf(fol_expr1)
    print("\nCNF Form:")
      print(cnf1)

      fol_expr2 = Implies(Or(P, Not(Q)), Or(Q,
R))
      print("\nExample 2: (P ∨ ¬Q) → (Q ∨ R)")
    print("Original FOL Expression:")
    print(fol_expr2)

      cnf2 = fol_to_cnf(fol_expr2)
      print("\nCNF Form:")
      print(cnf2)

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

OUTPUT:

```
Example 1: P → Q
Original FOL Expression:
Implies(P, Q)

CNF Form:
Q | ~P

Example 2: (P ∨ ¬Q) → (Q ∨ R)
Original FOL Expression:
Implies(P | ~Q, Q | R)

CNF Form:
Q | R
```