VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

Artificial Intelligence (23CS5PCAIN)

Submitted by

Nischal Kiran(1BM22CS182)

in partial fulfilment 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 **Nischal Kiran(1BM22CS182),** who is Bonafide student of **B.M.S. College of Engineering.** It is in partial fulfilment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

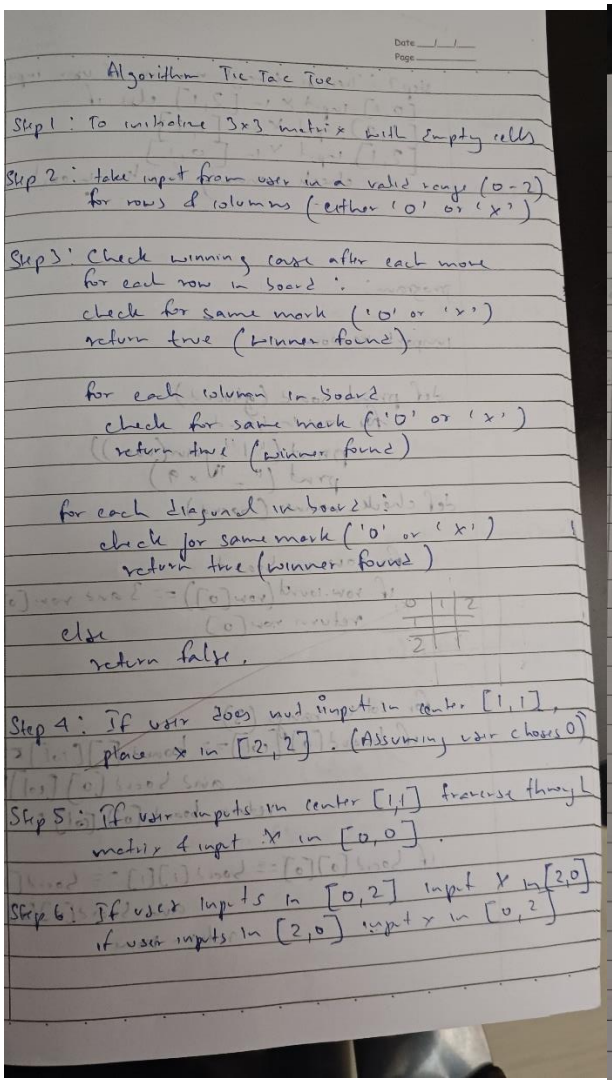
Github Link:

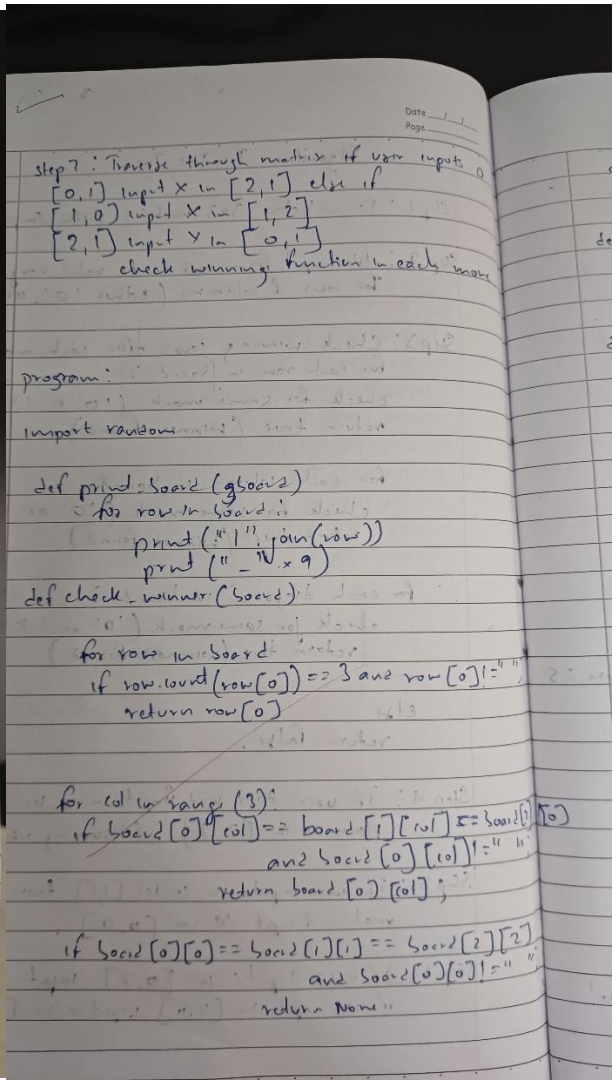
https://github.com/nischal-kiran/AI

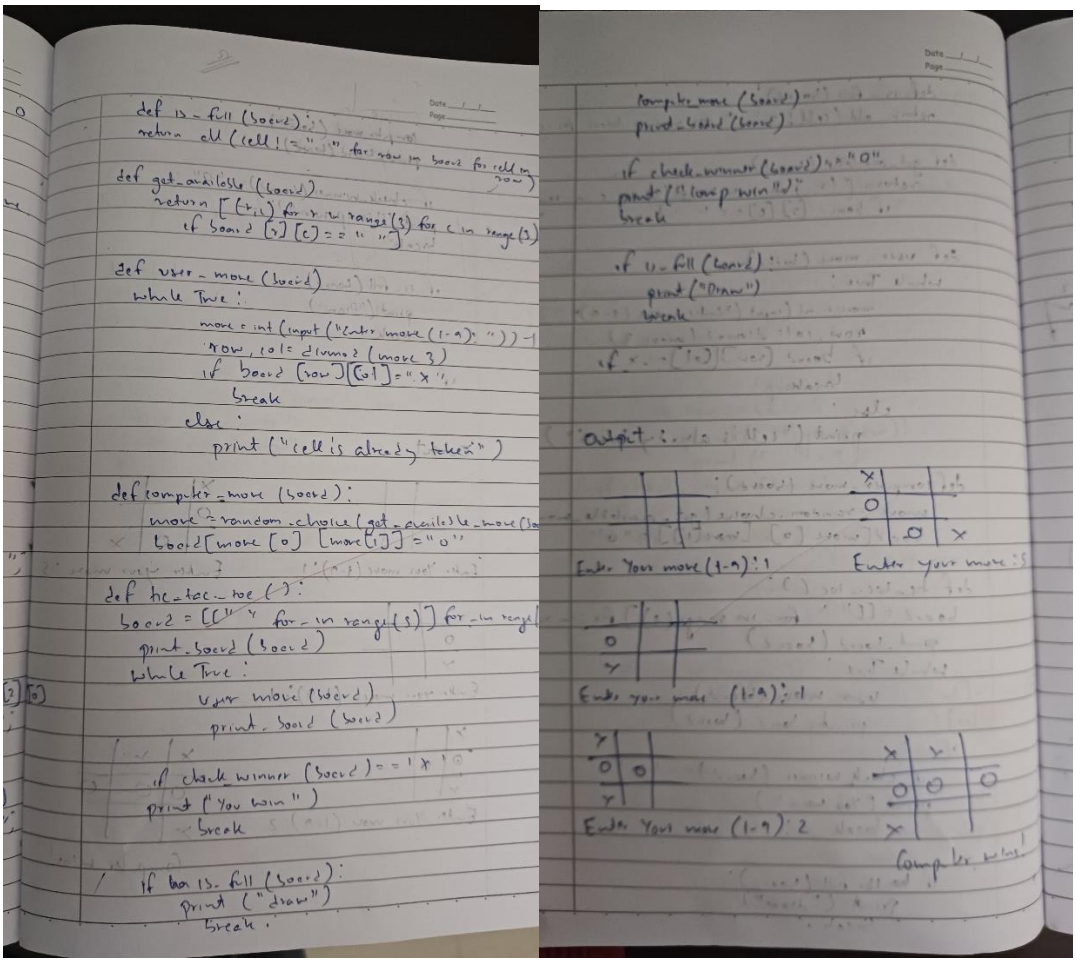
Program 1

Implement Tic –Tac –Toe Game

Algorithm:







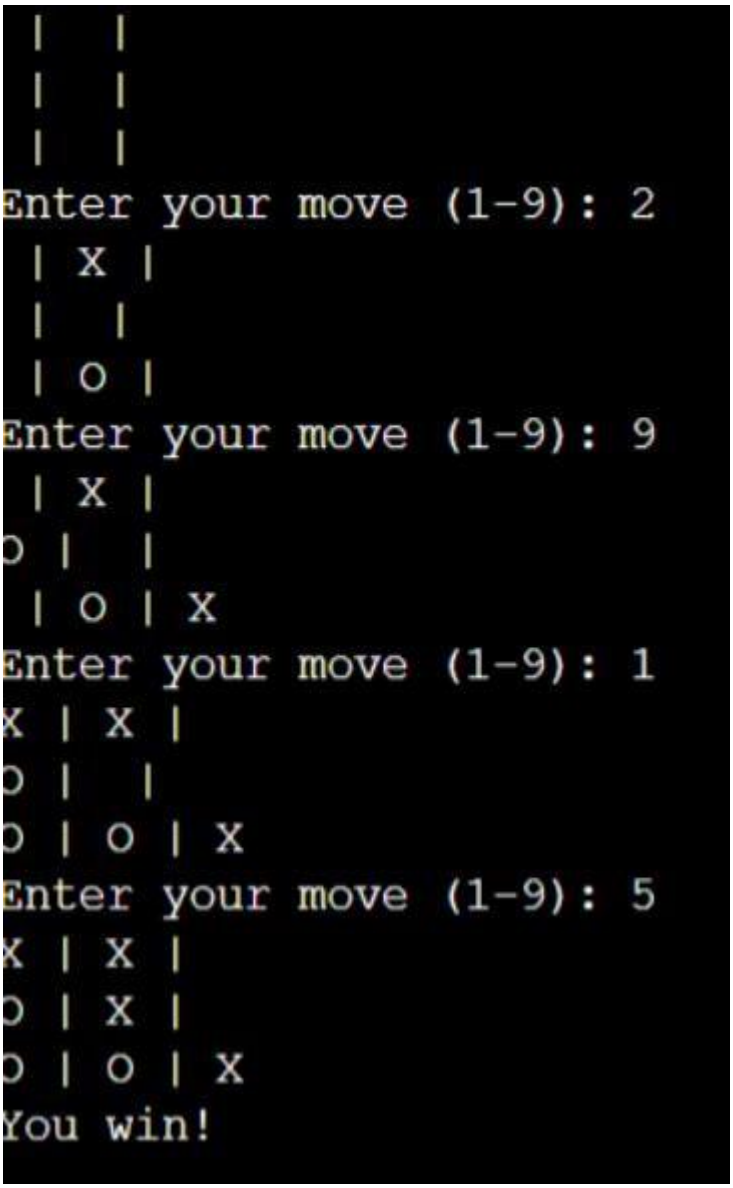
Code:

```
import random

def win(board):     for row in board:
if row[0] == row[1] == row[2] != "":
            return True     for col in range(3):          if
board[0][col] == board[1][col] == board[2][col] != "":
            return True     if board[0][0] == board[1][1]
== board[2][2] != "":          return True     if
board[0][2] == board[1][1] == board[2][0] != "":
        return True
return False
```

```
def printBoard(board):
    print("\n".join([" | ".join(row) for row in board]))

def draw(board):
    return all(cell != "" for row in board for cell in row)

def user_move(board):
    while True:
try:
            move = int(input("Enter your move (1-9): ")) - 1
row, col = divmod(move, 3)              if board[row][col]
== "":                  board[row][col] = "X"                  break
else:
                print("That space is already taken. Try again.")
except (ValueError, IndexError):
            print("Invalid input. Please enter a number from 1 to 9.")

def computer_move(board):
while True:
        move = random.randint(0, 8)
row, col = divmod(move, 3)          if
board[row][col] == "":
board[row][col] = "O"
break

def _main():
    board = [["" for _ in range(3)] for _ in range(3)]

    while True:
```

```
        printBoard(board)
user_move(board)        if
win(board):
            printBoard(board)
print("You win!")
break        if
draw(board):
            printBoard(board)
print("It's a draw!")            break
        computer_move(board)
if win(board):
            printBoard(board)
print("Computer wins!")
break        if draw(board):
printBoard(board)
print("It's a draw!")
break

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

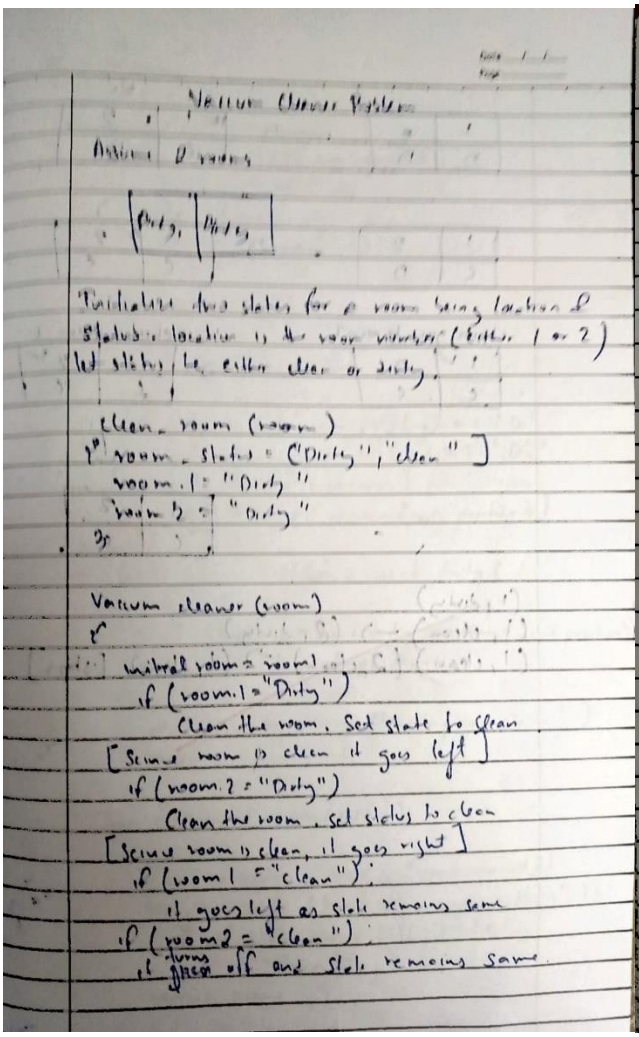
Output:


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

Program 2

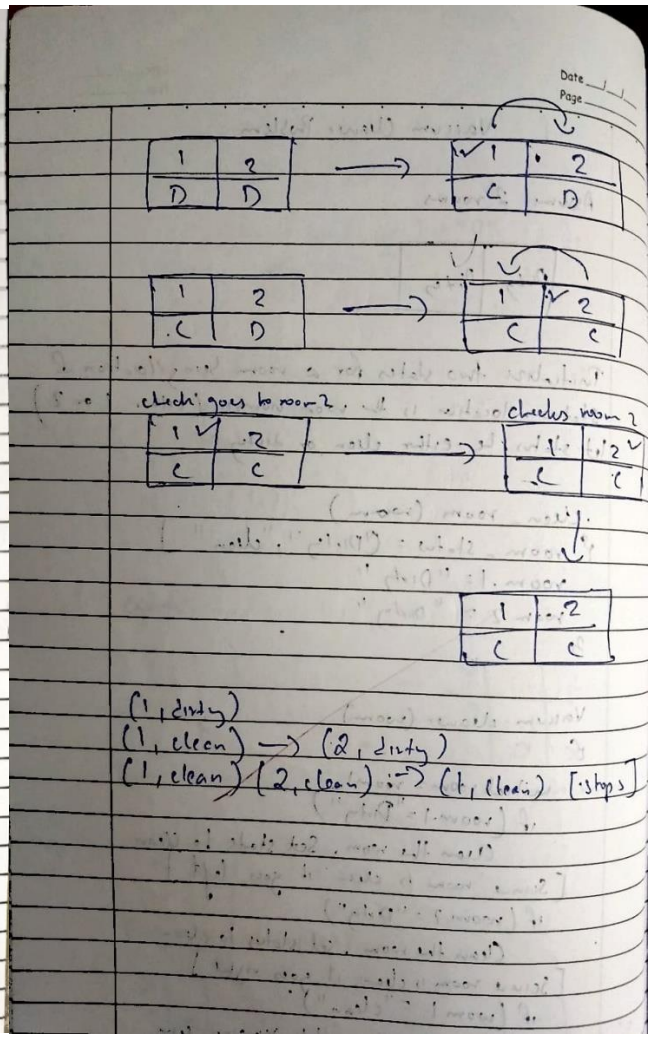
Implement vacuum cleaner agent

Algorithm:



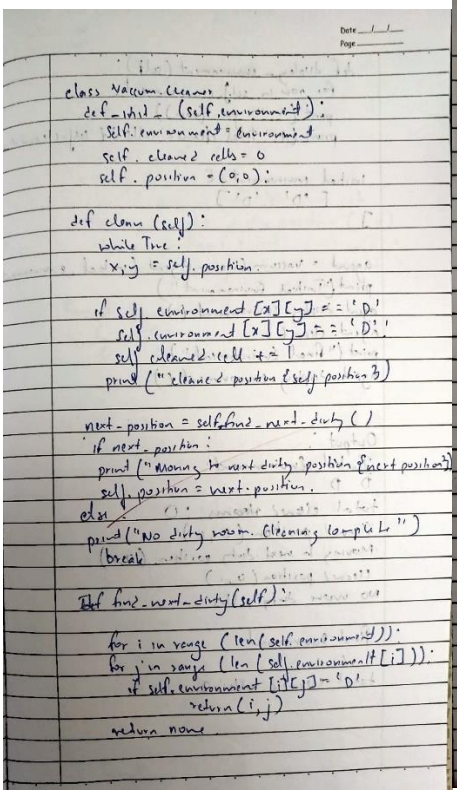
clean_room (room)
room_states = ["Dirty", "clean"]
room.1 = "Dirty"
room.2 = "Dirty"

Vacuum cleaner (room)
initial room = room1
if (room.1 = "Dirty")
Clean the room. Set state to clean
[Since room is clean it goes left]
if (room.2 = "Dirty")
Clean the room. Set states to clean
[Since room is clean, it goes right]
if (room1 = "clean"):
it goes left as state remains same
if (room2 = "clean")
it turns off and state remains same

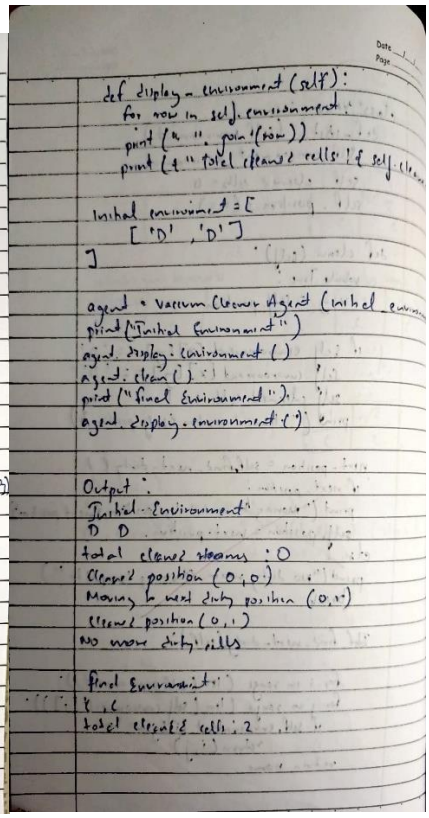


clock goes to room 2.
checks room 2

(1, dirty)
(1, clean) → (2, dirty)
(1, clean) (2, clean) → (1, clean) [stops]



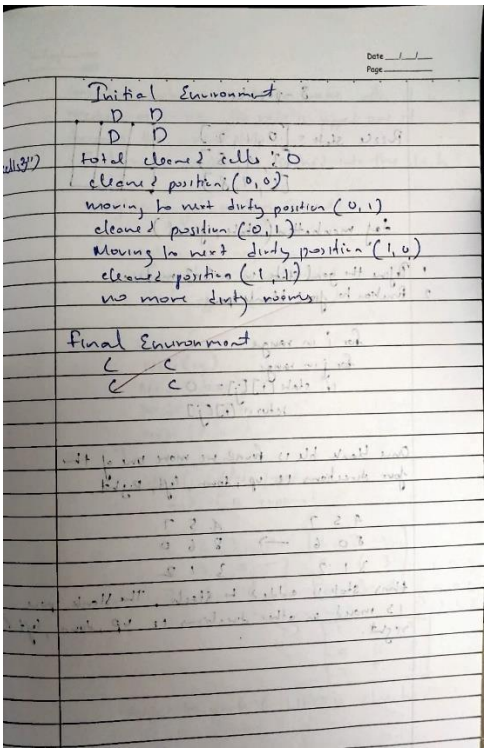
```
class Vacuum_Cleaner:
    def __init__(self, environment):
        self.environment = environment
        self.cleaned_cells = 0
        self.position = (0,0)

    def clean(self):
        while True:
            x,y = self.position

            if self.environment[x][y] == 'D':
                self.environment[x][y] = 'C'
                self.cleaned_cell += 1
                print("Cleaned position", self.position)

            next_position = self.find_next_dirty()
            if next_position:
                print("Moving to next dirty position", next_position)
                self.position = next_position
            else:
                print("No dirty rooms. Cleaning complete")
                break

    def find_next_dirty(self):
        for i in range(len(self.environment)):
            for j in range(len(self.environment[i])):
                if self.environment[i][j] == 'D':
                    return (i,j)
        return None
```



```
def display_environment(self):
    for row in self.environment:
        print(" ".join(row))
    print("Total cleaned cells:", self.cleaned_cells)

Initial environment = [
    ['D', 'D']
]

agent = Vacuum Cleaner Agent(initial_environment)
print("Initial Environment")
agent.display_environment()
agent.clean()
print("Final Environment")
agent.display_environment()
```

Output:

```
Initial Environment
D D
total cleaned rooms : 0
Cleaned position (0,0)
Moving to next dirty position (0,1)
Cleaned position (0,1)
No more dirty cells

Final Environment
C C
total cleaned cells : 2
```



```
Initial Environment
D D
D D
total cleaned cells : 0
cleaned position (0,0)
moving to next dirty position (0,1)
cleaned position (0,1)
Moving to next dirty position (1,0)
cleaned position (1,1)
no more dirty rooms

Final Environment
C C
C C
```

Code:

```
def printArr(arr):
for row in arr:
print(row)
print() def
clean(arr, x, y):
if arr[x][y] == 1:
arr[x][y] = 0  def
check(arr):     for
row in arr:         if
1 in row:
return True
return False

# Directions: right (0,1), down (1,0), left (0,-1), up (-1,0)
directions = [(0, 1), (1, 0), (0, -1), (-1, 0)] direction_index
= 0  # Start moving right

# Get room status print("Enter the status of the rooms (0 for
clean; 1 for dirty):") arr1 = [] for i in range(2):
   row = []     for j in range(2):         a =
int(input(f"Status of room ({i}, {j}): "))
row.append(a)     arr1.append(row)
x, y = 0, 0  #Start cleaning from the first room
while True:     printArr(arr1)     if not
check(arr1):
       break
clean(arr1, x, y)
   #Move to the next room in the current direction
dx, dy = directions[direction_index]     new_x,
new_y = x + dx, y + dy
```

```
   #Check bounds     if 0 <= new_x < 2
and 0 <= new_y < 2:
       x, y = new_x, new_y
else:
       #Change direction(turn right)
direction_index = (direction_index + 1) % 4         dx,
dy = directions[direction_index]         x, y = x + dx, y
+ dy  #Move in the new direction print("All rooms are
cleaned!")
```

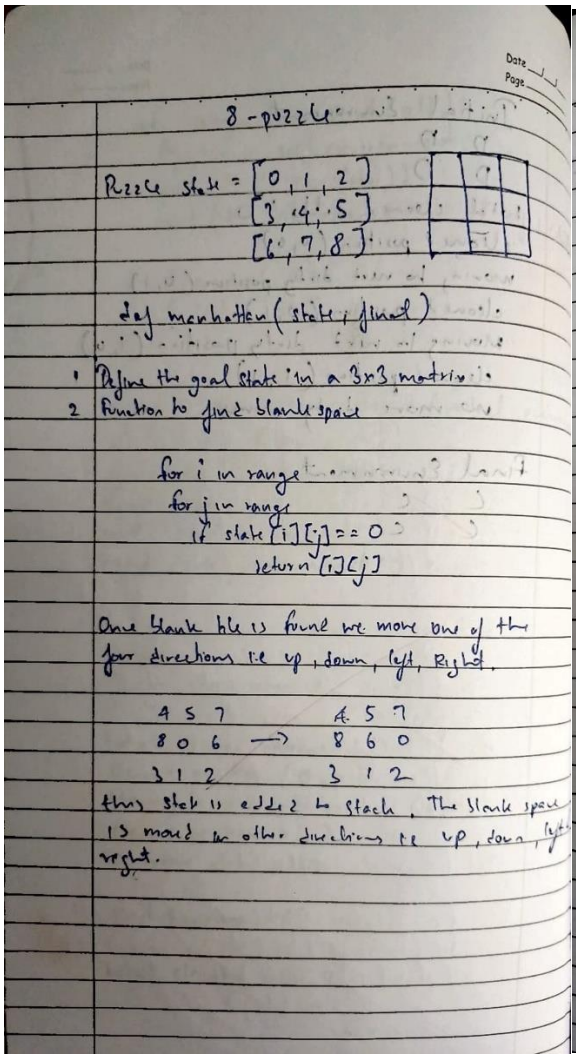
Output:

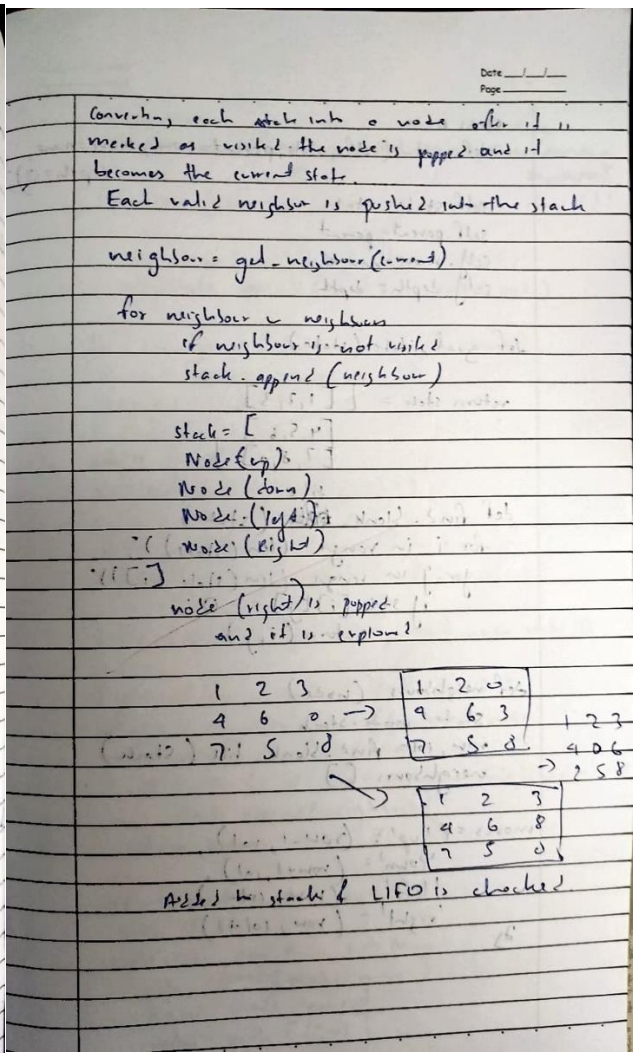
```
Enter the status of the rooms (0 for clean; 1 for dirty):
Status of room (0, 0): 1
Status of room (0, 1): 0
Status of room (1, 0): 1
Status of room (1, 1): 0
[1, 0]
[1, 0]

[0, 0]
[1, 0]

[0, 0]
[1, 0]

[0, 0]
[1, 0]

[0, 0]
[0, 0]

All rooms are cleaned!
```

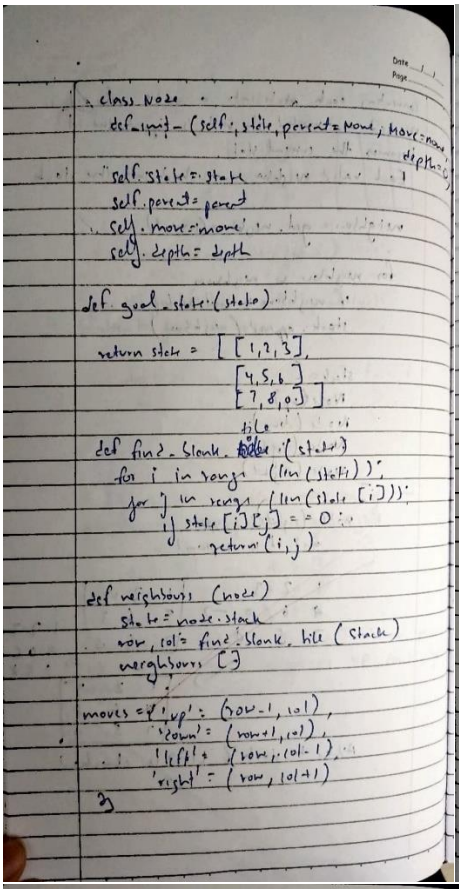
Program 3

Implement 8 puzzle problems using Depth First Search (DFS)

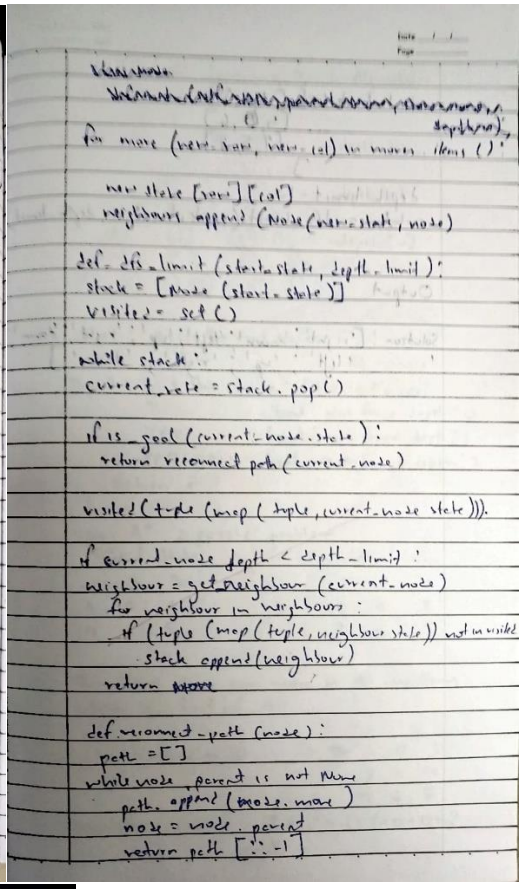
Algorithm:



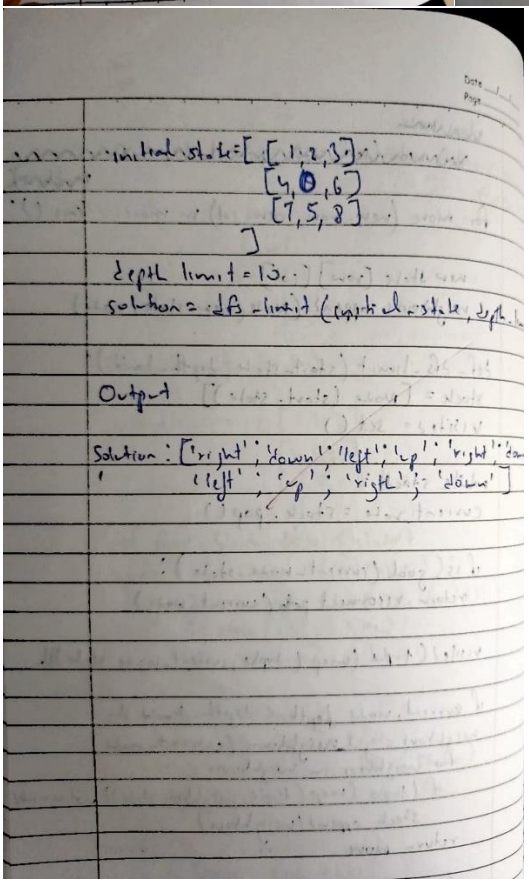




```
class Node
def __init__(self, state, parent=None, move=None, depth=0):
    self.state = state
    self.parent = parent
    self.move = move
    self.depth = depth

def goal_state(state):
    return state == [[1,2,3],
                     [4,5,6],
                     [7,8,0]]

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

def neighbours(node):
    state = node.state
    row, col = find_blank_tile(state)
    neighbours = []

moves = {'up': (row-1, col),
         'down': (row+1, col),
         'left': (row, col-1),
         'right': (row, col+1)
}
```



```
for move (new_row, new_col) in moves.items():

    new_state[row][col]
    neighbours.append(Node(new_state, node,
                           depth+1))

def dfs_limit(start_state, depth_limit):
    stack = [Node(start_state)]
    visited = set()

    while stack:
    current_node = stack.pop()

    if is_goal(current_node.state):
        return reconnect_path(current_node)

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

    if current_node.depth < depth_limit:
    neighbour = get_neighbour(current_node)
        for neighbour in neighbours:
            if (tuple(map(tuple, neighbour.state)) not in visited
                stack.append(neighbour)
        return None

def reconnect_path(node):
    path = []
    while node.parent is not None:
        path.append(node.move)
        node = node.parent
        return path[::-1]
```



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

depth_limit = 10
solution = dfs_limit(initial_state, depth_limit)
```

Output

Solution : ['right', 'down', 'left', 'up', 'right', 'down', 'left', 'up', 'right', 'down']

Code:

```
class PuzzleState:     def __init__(self, board,
moves=0, previous=None):
        self.board = board         self.moves
= moves         self.previous = previous
self.empty_pos = self.find_empty()

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

    def manhattan_distance(self):
dist = 0         for i in range(3):
for j in range(3):
                tile = self.board[i][j]
if tile != 0:
                    target_x = (tile - 1) // 3
target_y = (tile - 1) % 3                    dist += abs(i -
target_x) + abs(j - target_y)         return dist

    def generate_moves(self):
moves = []         x, y =
self.empty_pos
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

        for dx, dy in directions:
```

```
            new_x, new_y = x + dx, y + dy
if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = [row[:] for row in self.board]                 new_board[x][y],
            new_board[new_x][new_y]    =          new_board[new_x][new_y], new_board[x][y]
                moves.append(PuzzleState(new_board, self.moves + 1, self))

        return moves

  def dfs(start_board, max_depth):     stack
= [PuzzleState(start_board)]     visited =
set()     goal_state = [[1, 2, 3], [4, 5, 6], [7,
8, 0]]

    while stack:
        current_state = stack.pop()         if
current_state.board == goal_state:
            return current_state
visited.add(tuple(map(tuple, current_state.board)))
if current_state.moves < max_depth:
            for next_state in current_state.generate_moves():
                if tuple(map(tuple, next_state.board)) not in visited:
if next_state.manhattan_distance() < 10:
                        stack.append(next_state)
return None

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

```
moves taken to reach the final state:
{len(path) - 1}") initial_board = [[1, 2,
3], [4, 0, 5], [7, 8, 6]] max_depth = 10
solution = dfs(initial_board, max_depth)
if solution:
    print("Solution found:")
print_solution(solution) else:
    print("No solution found.")
```

Output:

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

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

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

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

Program 4

Implement A* search algorithm

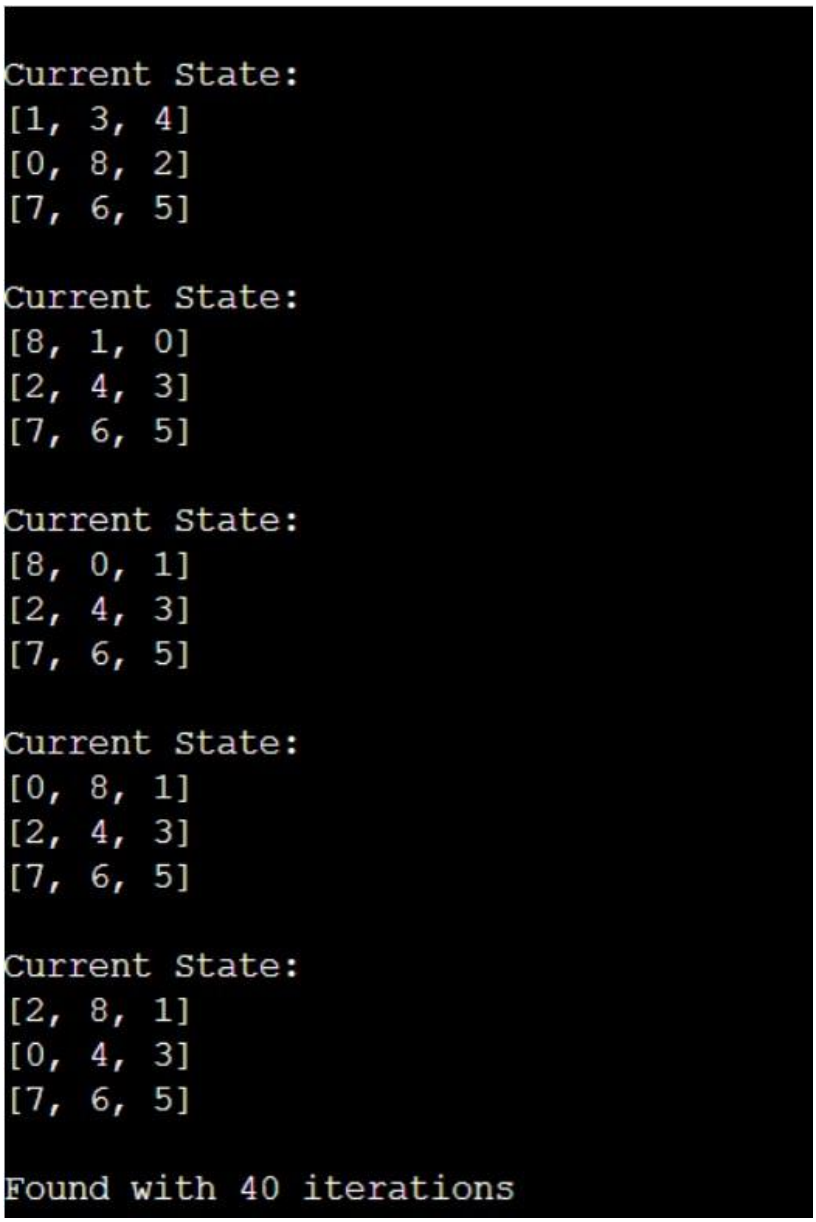
Algorithm

Code:

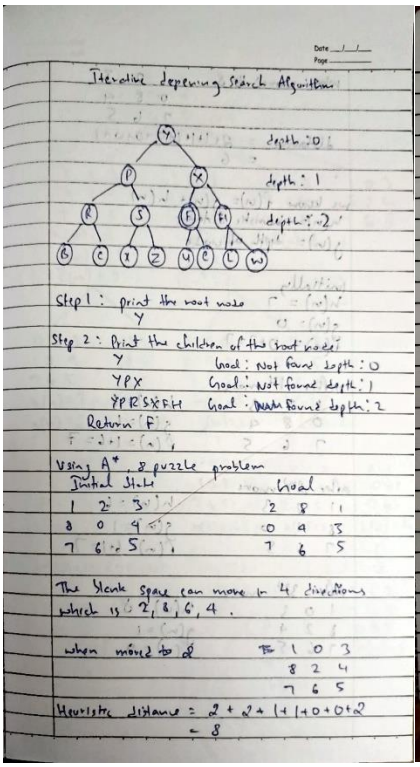
```
def H_n(state, target):
   return sum(x != y for x, y in zip(state, target))
def F_n(state_with_lvl, target):     state, lvl =
state_with_lvl     return H_n(state, target) + lvl def
possible_moves(state_with_lvl, visited_states):
   state, lvl = state_with_lvl     b =
state.index(0)       directions = []
pos_moves = []     if b <= 5:
directions.append('d')     if b >= 3:
directions.append('u')     if b % 3 >
0: directions.append('l')     if b % 3 <
2: directions.append('r')     for move
in directions:
       temp = gen(state, move, b)
if temp not in visited_states:
           pos_moves.append([temp, lvl + 1])      return
pos_moves def gen(state, move, b):     temp = state.copy()
if move == 'l': temp[b], temp[b - 1] = temp[b - 1], temp[b]
if move == 'r': temp[b], temp[b + 1] = temp[b + 1], temp[b]
```

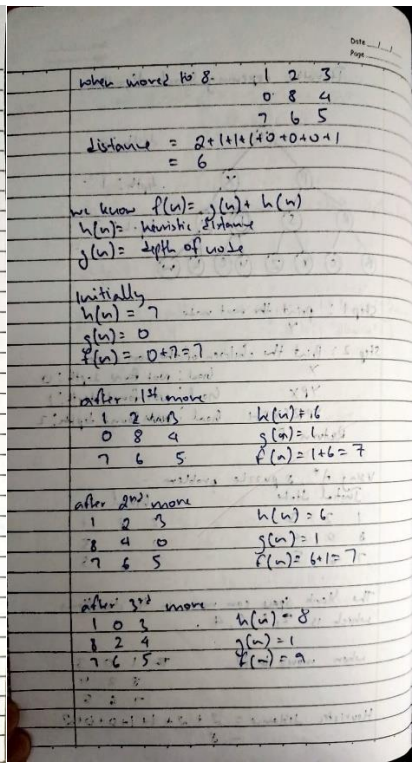
```
if move == 'u': temp[b], temp[b - 3] = temp[b - 3], temp[b]
if move == 'd': temp[b], temp[b + 3] = temp[b + 3], temp[b]
return temp def display_state(state):     print("Current State:")
for i in range(0, 9, 3):         print(state[i:i+3])     print()   def
astar(src, target):     arr = [[src, 0]]       visited_states = []
iterations = 0     while arr:
       iterations += 1         current = min(arr,
key=lambda x: F_n(x, target))
arr.remove(current)         display_state(current[0])
if current[0] == target:
           return f'Found with {iterations} iterations'
visited_states.append(current[0])
arr.extend(possible_moves(current, visited_states))
return 'Not found' src = [1, 2, 3, 8, 0, 4, 7, 6, 5]   target
= [2, 8, 1, 0, 4, 3, 7, 6, 5]   print(astar(src, target))
```

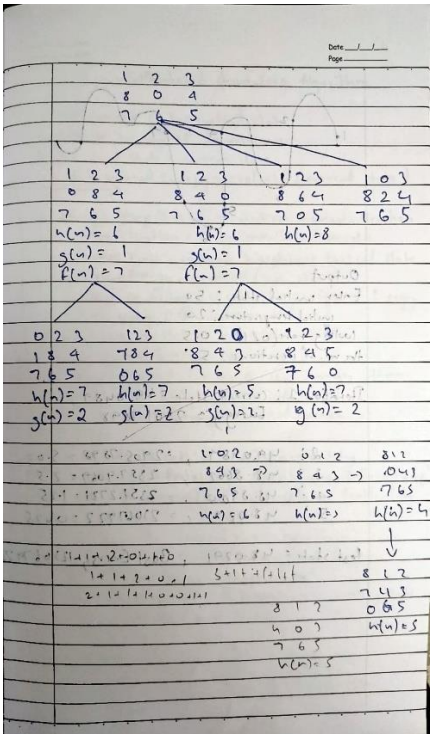
Output:



Implement Iterative deepening search algorithm **Algorithm:**


Iterative deepening Search Algorithm
depth : 0
depth : 1
depth : 2
Step 1 : print the root node
Y
Step 2 : Print the children of the root node
Y
goal : Not found depth : 0
YPX
Goal : Not found depth : 1
YPRSXFH
Goal : Found depth : 2
Return 'F'
Using A* , 8 puzzle problem
Initial State
Goal
Heuristic distance = 2 + 2 + 1 + 1 + 0 + 0 + 2
= 8


when moved to 8
distance = 2+1+1+1+0+0+0+1
= 6
we know f(n) = g(n) + h(n)
h(n) = Heuristic distance
g(n) = depth of node
Initially
h(n) = 7
g(n) = 0
f(n) = 0+7 = 7
after 1st move
after 2nd move
after 3rd move

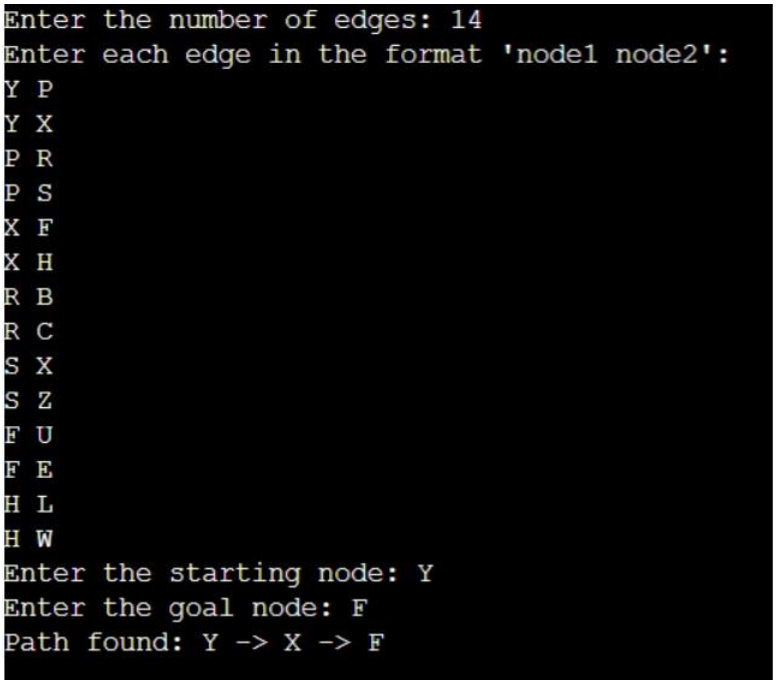

h(n) = 6
h(n) = 6
h(n) = 8
g(n) = 1
g(n) = 1
f(n) = 7
f(n) = 7
h(n) = 7
h(n) = 7
h(n) = 5
h(n) = 7
g(n) = 2
g(n) = 2
g(n) = 2
g(n) = 2
h(n) = 6
h(n) = 5
h(n) = 4
h(n) = 5

Code:

```
def iterative_deepening_search(graph, start, goal):
def depth_limited_search(node, goal, depth):
if depth == 0:             if node == goal:
return [node]             else:
                return None
elif depth > 0:
            for child in graph.get(node, []):
                result = depth_limited_search(child, goal, depth - 1)
if result is not None:                        return [node] + result
return None     depth = 0     while True:
        result = depth_limited_search(start, goal, depth)
if result is not None:
            return result
depth += 1 def
get_user_input_graph():
    graph = {}     num_edges = int(input("Enter the
number of edges: "))     print("Enter each edge in the
format 'node1 node2':")     for _ in range(num_edges):
node1, node2 = input().split()         if node1 in graph:
            graph[node1].append(node2)
else:
            graph[node1] = [node2]
if node2 in graph:
graph[node2].append(node1)
else:
            graph[node2] = [node1]
return graph def main():
    graph = get_user_input_graph()     start_node = input("Enter the
starting node: ")     goal_node = input("Enter the goal node: ")
path = iterative_deepening_search(graph, start_node, goal_node)
if path:
```

```
    print(f"Path found: {' -> '.join(path)}")
else:
    print("No path found") if
__name__ == "__main__":
main()
```

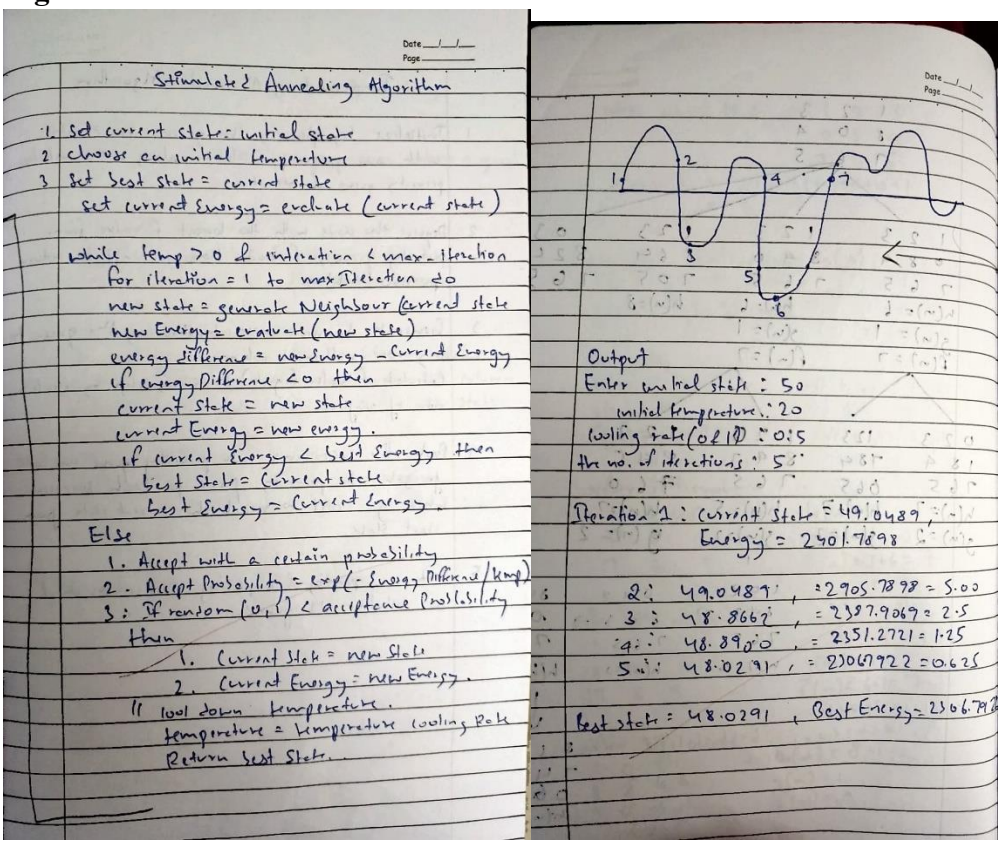
Output:



Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



Simulated Annealing Algorithm

1. Set current state: initial state
2. choose an initial temperature
3. Set best state = current state
   set current Energy = evaluate (current state)

while temp > 0 & iteration < max_iteration
for iteration = 1 to maxIteration do
new state = generate Neighbour (current state
new Energy = evaluate (new state)
energy Difference = new Energy - current Energy
if energy Difference < 0 then
current state = new state
current Energy = new energy.
if current Energy < best Energy then
best state = Current state
best Energy = Current Energy.
Else
1. Accept with a certain probability
2. Accept Probability = exp(- Energy Difference / temp)
3: If random (0,1) < acceptance Probability
then
1. Current State = new State
2. Current Energy = new Energy.
// cool down temperature.
temperature = temperature cooling Rate
Return best State.

Output
Enter initial state : 50
initial temperature : 20
cooling rate (0 & 1) : 0.5
the no. of iterations : 5

Iteration 1 : current state = 49.0489,
Energy = 2401.7898

2: 49.0487 , = 2905.7898 = 5.00
3: 48.8667 , = 2387.9069 = 2.5
4: 48.8900 , = 2351.2721 = 1.25
5: 48.0291 , = 2306.7922 = 0.625

Best state = 48.0291 , Best Energy = 2306.79

Code:

```
import random import
math


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


def adaptive_simulated_annealing(start, temp, cooling_rate, lower_limit, upper_limit):
```

```
    current = start     current_energy
= energy(current)

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

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

        new_energy = energy(new)

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

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

    return current
```

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

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

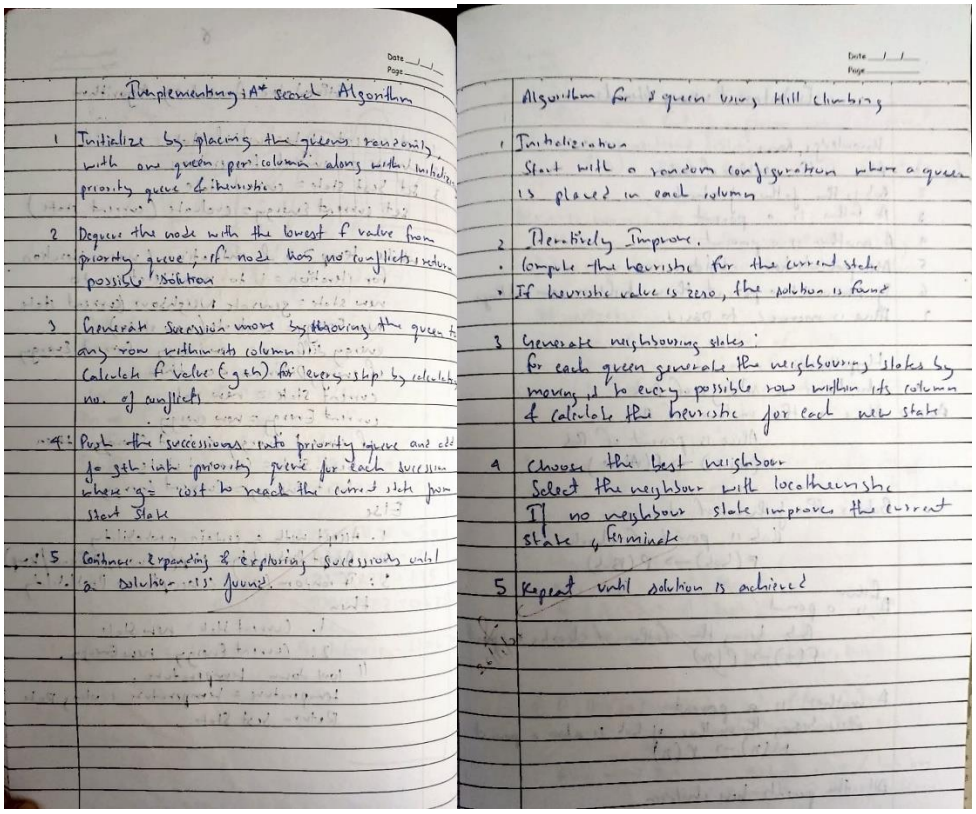
Output:

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

Program 6

Implement A* search algorithm for N queens

Algorithm:



Implementing A* search Algorithm

1. Initialize by placing the queens randomly with one queen per column along with initializing priority queue of heuristic
2. Dequeue the node with the lowest f value from priority queue ; if node has no conflicts return possible solution
3. Generate successive moves by shifting the queen to any row within its column
   Calculate f value (g+h) for every step by calculating no. of conflicts
4. Push the successions into priority queue and add to state into priority queue for each succession where g = cost to reach the current state from start state
5. Continue expanding & exploring successors until a solution is found.

Algorithm for 8 queen using Hill climbing

1. Initialization
   Start with a random configuration where a queen is placed in each column
2. Iteratively Improve.
   - Compute the heuristic for the current state.
   - If heuristic value is zero, the solution is found
3. Generate neighbouring states :
   For each queen generate the neighbouring states by moving it to every possible row within its column & calculate the heuristic for each new state
4. Choose the best neighbour
   Select the neighbour with lowest heuristic
   If no neighbour state improves the current state, terminate
5. Repeat until solution is achieved

Code:

```
import heapq


# Helper function to calculate the heuristic (number of conflicts)
def heuristic(board):
```

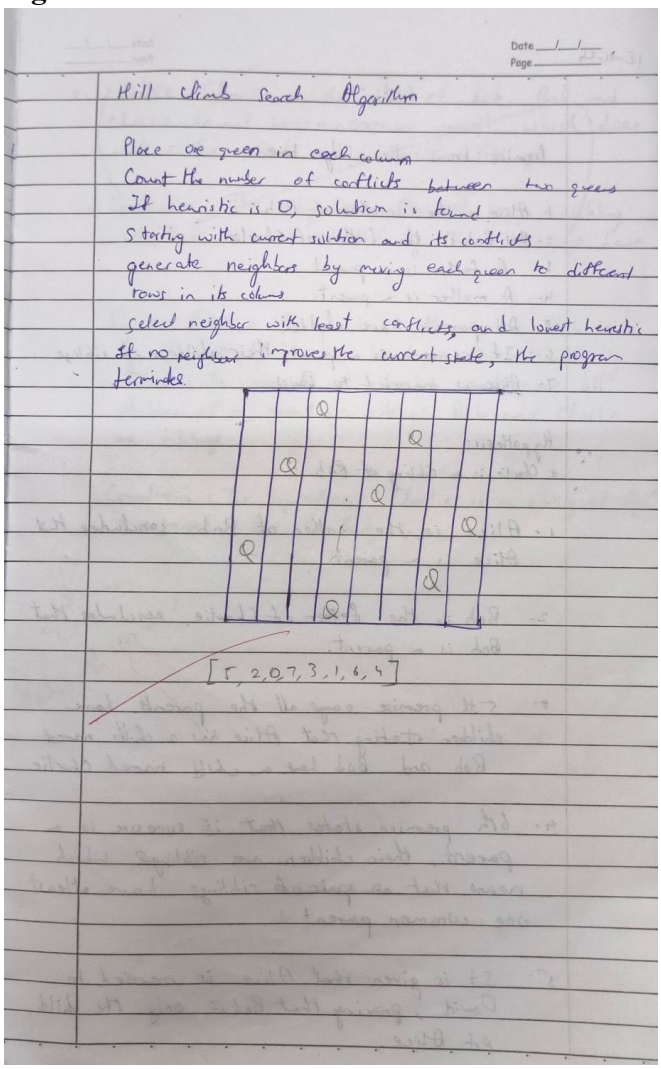
```
    conflicts = 0     for i in range(len(board)):         for j in range(i
+ 1, len(board)):             if board[i] == board[j] or abs(board[i] -
board[j]) == j - i:
                conflicts += 1
return conflicts

# A* Search for 8-queens
def a_star_8_queens():
n = 8     open_set = []
    # Initial state: empty board
heapq.heappush(open_set, (0, []))  # (f, board)

    while open_set:         f, board =
heapq.heappop(open_set)
        # Goal check         if len(board) == n and
heuristic(board) == 0:
            return board         #
Generate successors
row = len(board)         for
col in range(n):
             new_board = board + [col]             if
heuristic(new_board) == 0:  # No conflicts so far
g = row + 1                 h = heuristic(new_board)
heapq.heappush(open_set, (g + h, new_board))     return
None  # No solution found

# Run A* search solution = a_star_8_queens() print("Solution
board (column positions for each row):", solution)
```

Output:

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

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



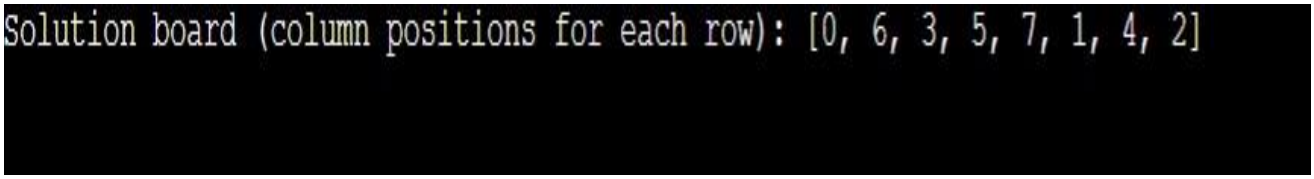
Code:

import random

```
# Helper function to calculate the heuristic (number of conflicts)
def heuristic(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]) == j - i:                 conflicts += 1     return conflicts

# Hill climbing for 8-queens def
hill_climbing_8_queens():     n
= 8
    # Generate a random initial state     board =
[random.randint(0, n - 1) for _ in range(n)]

    while True:
        current_h = heuristic(board)
if current_h == 0:
            return board  # Solution found

        # Find the best neighbor by moving each queen to every other column in its row
best_board = board[:]         best_h = current_h         for row in range(n):             for
col in range(n):                 if col == board[row]:                     continue
new_board = board[:]                 new_board[row] = col                 new_h =
heuristic(new_board)

                # If the new board has fewer conflicts, update the best board
if new_h < best_h:                     best_h = new_h
best_board = new_board

        # If no improvement, we're stuck in a local minimum; restart
if best_h >= current_h:
            board = [random.randint(0, n - 1) for _ in range(n)]
else:
            board = best_board
```

```
# Run hill climbing search solution = hill_climbing_8_queens()
print("Solution board (column positions for each row):", solution)
```

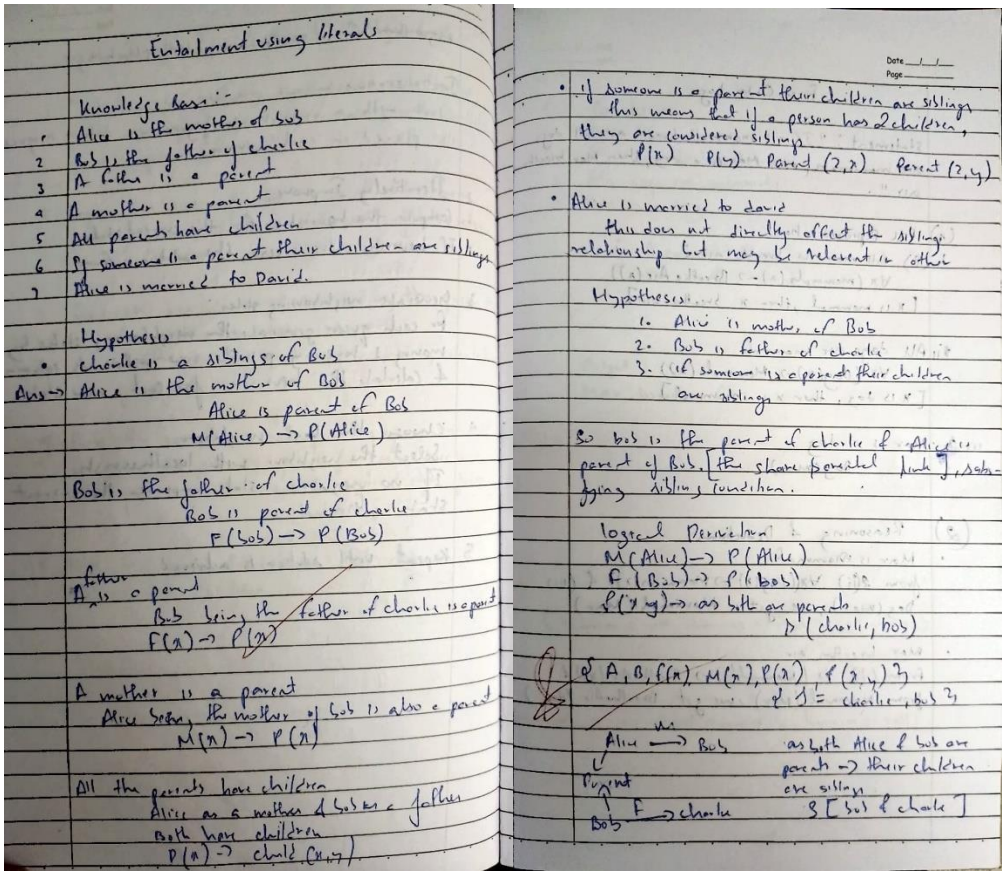
Output:



Program 7

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

Algorithm:



Code: import
random

```
# Helper function to calculate the heuristic (number of conflicts)
def heuristic(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]) == j - i:
            conflicts += 1
return conflicts

# Hill climbing for 8-queens def
hill_climbing_8_queens():     n
= 8
   # Generate a random initial state     board =
[random.randint(0, n - 1) for _ in range(n)]

   while True:
      current_h = heuristic(board)
if current_h == 0:
         return board  # Solution found

      # Find the best neighbor by moving each queen to every other column in its row
best_board = board[:]         best_h = current_h         for row in range(n):             for
col in range(n):                if col == board[row]:                    continue
new_board = board[:]               new_board[row] = col
            new_h = heuristic(new_board)
```

```
            # If the new board has fewer conflicts, update the best board
if new_h < best_h:                  best_h = new_h
best_board = new_board

      # If no improvement, we're stuck in a local minimum; restart
if best_h >= current_h:
          board = [random.randint(0, n - 1) for _ in range(n)]
else:
          board = best_board

# Run hill climbing search solution = hill_climbing_8_queens()
print("Solution board (column positions for each row):", solution)
```

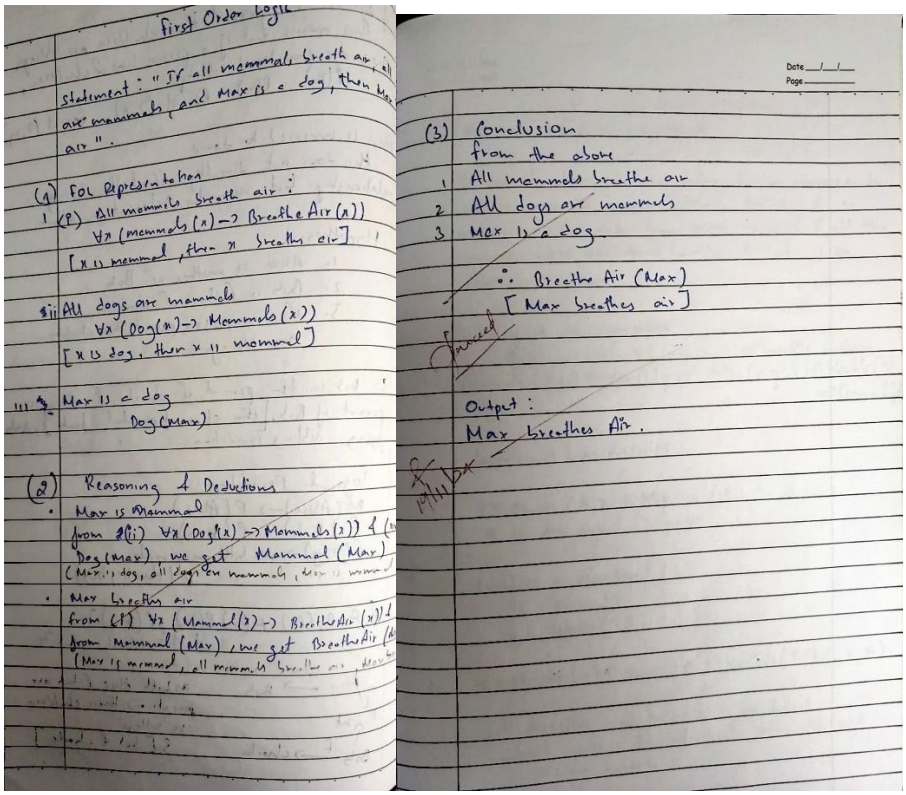
Output:

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

Program 8

Implement unification in first order logic

Algorithm:



Code:

```
def unify(x, y, subst=None):
    """
    Unification Algorithm: Unifies two terms, X and Y.
    """     if subst is
None:       subst
= {}
```

```
    if x == y:  # Step 1(a): If X and Y are identical         return subst
elif isinstance(x, str) and x.islower():  # Step 1(b): If X is a variable
return unify_variable(x, y, subst)
    elif isinstance(y, str) and y.islower():  # Step 1(c): If Y is a variable         return
unify_variable(y, x, subst)     elif isinstance(x, tuple) and isinstance(y, tuple):  # Step 2:
Check predicates and arguments         if x[0] != y[0] or len(x) != len(y):  # Predicate symbol
or argument count mismatch             return None         for x_i, y_i in zip(x[1:], y[1:]):  # Step
5: Recurse through arguments             subst = unify(x_i, y_i, subst)             if subst is None:
return None         return subst     else:
        return None  # Step 1(d): Failure case


def unify_variable(var, x, subst):
    """
    Unify variable with another term.
    """     if var in
subst:
        return unify(subst[var], x, subst)     elif
occurs_check(var, x, subst):  # Check if var occurs in x
return None     else:
        subst[var] = x
return subst


def occurs_check(var, x, subst):
    """
    Check if a variable occurs in a term.
    """     if
var == x:
return True
elif
```

```
isinstance(x,
tuple):
    return any(occurs_check(var, xi, subst) for xi in x)
elif isinstance(x, str) and x in subst:
    return occurs_check(var, subst[x], subst)
return False


# Test cases for unification
x1 = ("P", "a", "x") y1 =
("P", "a", "b")

x2 = ("Q", "x", ("R", "x")) y2
= ("Q", "a", ("R", "a"))

print("Unifying", x1, "and", y1, "=>", unify(x1, y1))
print("Unifying", x2, "and", y2, "=>", unify(x2, y2))
```

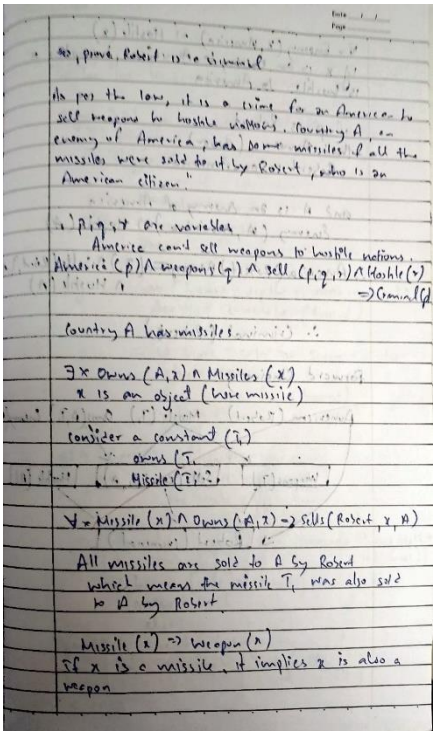
Output:

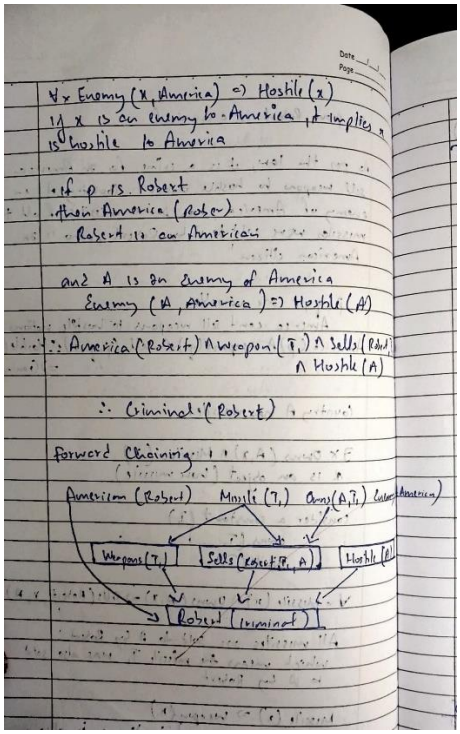
```
Unifying ('P', 'a', 'x') and ('P', 'a', 'b') => {'x': 'b'}
Unifying ('Q', 'x', ('R', 'x')) and ('Q', 'a', ('R', 'a')) => {'x': 'a'}
```

Program 9

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

Algorithm:





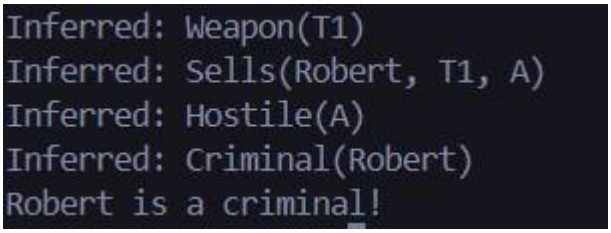
Code:

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


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

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

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

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

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

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

    # Check if we've reached our goal
    if 'Criminal(Robert)' in KB:
print("Robert is a criminal!")     else:
```

```
        print("No more inferences can be made.")

# Run forward chaining to attempt to derive the conclusion
forward_chaining()
```

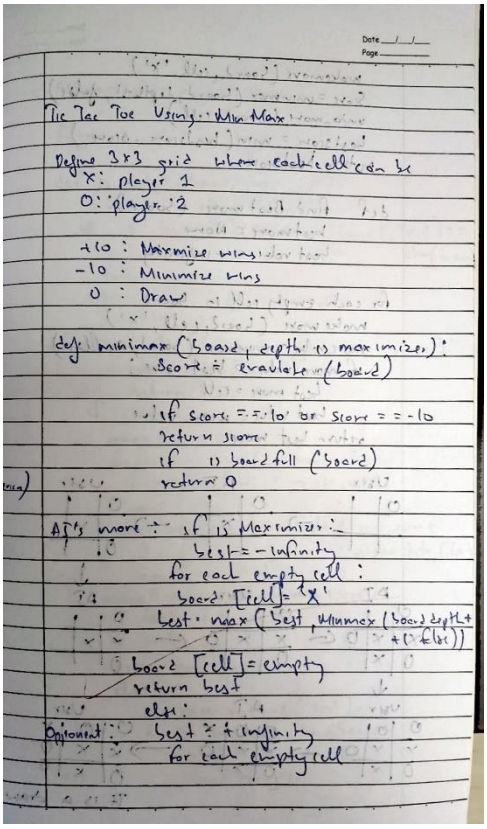
Output:

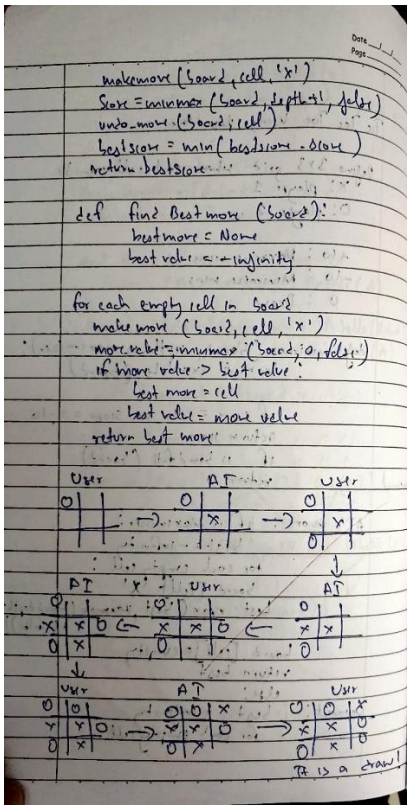


Program 10

Implement Min-Max Algorithm for Tic Tac Toe

Algorithm:





Code:

```
import math


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

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

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

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

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

    if is_maximizing:
```

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

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

# Example usage if
__name__ == "__main__":
# Initialize a sample board
board = [
```

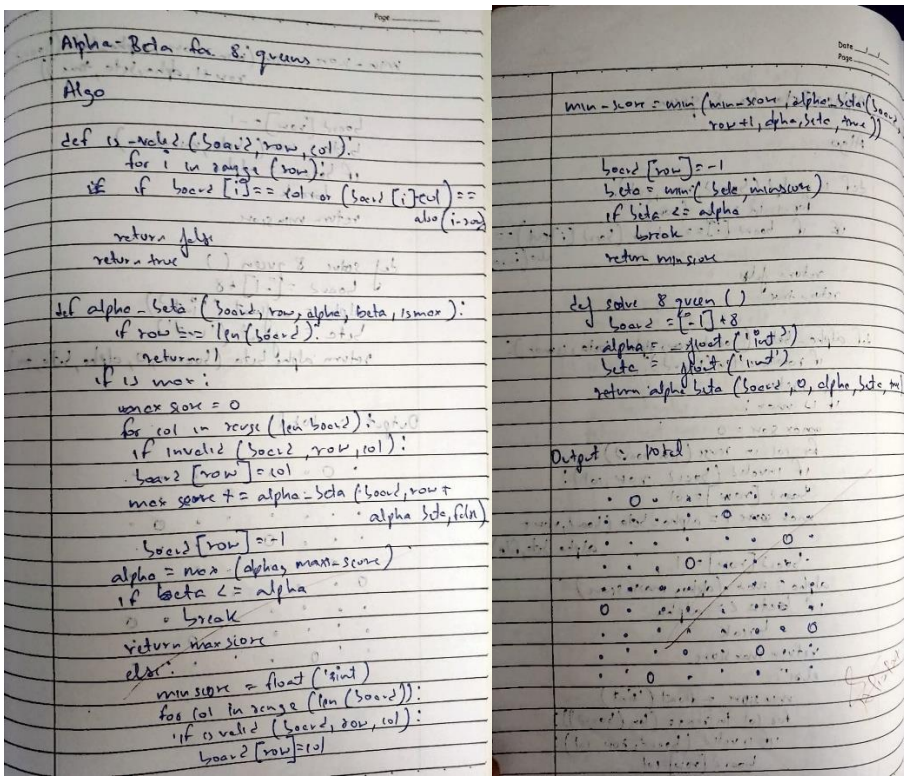
```
        ['X', 'O', 'X'],
        ['O', 'X', 'O'],
        ['_', '_', '_']
    ]
    print("Current Board:")
print_board(board)

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

Output:

```
Welcome to Tic Tac Toe!
 | |
-----
 | |
-----
 | |
-----
Player X's turn.
Enter the row (0, 1, 2): 0
Enter the column (0, 1, 2): 0
X| |
-----
 | |
-----
 | |
-----
Player O's turn.
Enter the row (0, 1, 2): 1
Enter the column (0, 1, 2): 1
X| |
-----
 |O|
-----
 | |
-----
Player X's turn.
Enter the row (0, 1, 2): 0
Enter the column (0, 1, 2): 2
X| |X
-----
 |O|
-----
 | |
-----
Player O's turn.
Enter the row (0, 1, 2): 0
Enter the column (0, 1, 2): 1
X|O|X
-----
 |O|
-----
 | |
-----
Player X's turn.
Enter the row (0, 1, 2): 2
Enter the column (0, 1, 2): 2
X|O|X
-----
 |O|
-----
 | |X
-----
Player O's turn.
Enter the row (0, 1, 2): 2
Enter the column (0, 1, 2): 1
X|O|X
-----
 |O|
-----
 |O|X
-----
Player O wins!
```

Implement Alpha-Beta Pruning for 8 queens

Algorithm:



Code:

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

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

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

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

```
            return False

        return True

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

        if maximizing_player:
max_eval = float('-inf')
best_board = None             for row in
range(self.size):                 if
self.is_safe(board, row, col):
                board[row][col] = 1
                eval_score, potential_board = self.alpha_beta_search(board, col + 1, alpha, beta, False)
board[row][col] = 0                     if eval_score > max_eval:                         max_eval = eval_score
best_board = potential_board                     alpha = max(alpha, eval_score)                     if beta <=
alpha:  # Beta cutoff                         break
        return max_eval, best_board
else:
        min_eval = float('inf')
        best_board = None             for
row in range(self.size):                 if
self.is_safe(board, row, col):
                board[row][col] = 1                     eval_score, potential_board =
self.alpha_beta_search(board, col + 1, alpha, beta, True)                     board[row][col] = 0
if eval_score < min_eval:                         min_eval = eval_score                         best_board =
potential_board                     beta = min(beta, eval_score)                     if beta <= alpha:  # Alpha
cutoff                         break
        return min_eval, best_board

    def solve(self):
```

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

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



if __name__ == "__main__":
game = EightQueens()
solution = game.solve()    if
solution:
    print("Solution found:")
    game.print_board(solution)
else:
    print("No solution exists.")
```

Output: