VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

ANKIT SINGH BHATTI (1BM22CS353)

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 **ANKIT SINGH BHATTI (1BM22CS353),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Prof. Sunayana S Assistant Professor Department of CSE, BMSCE	Dr. Jyothi S Nayak Professor & HOD Department of CSE, BMSCE

Index

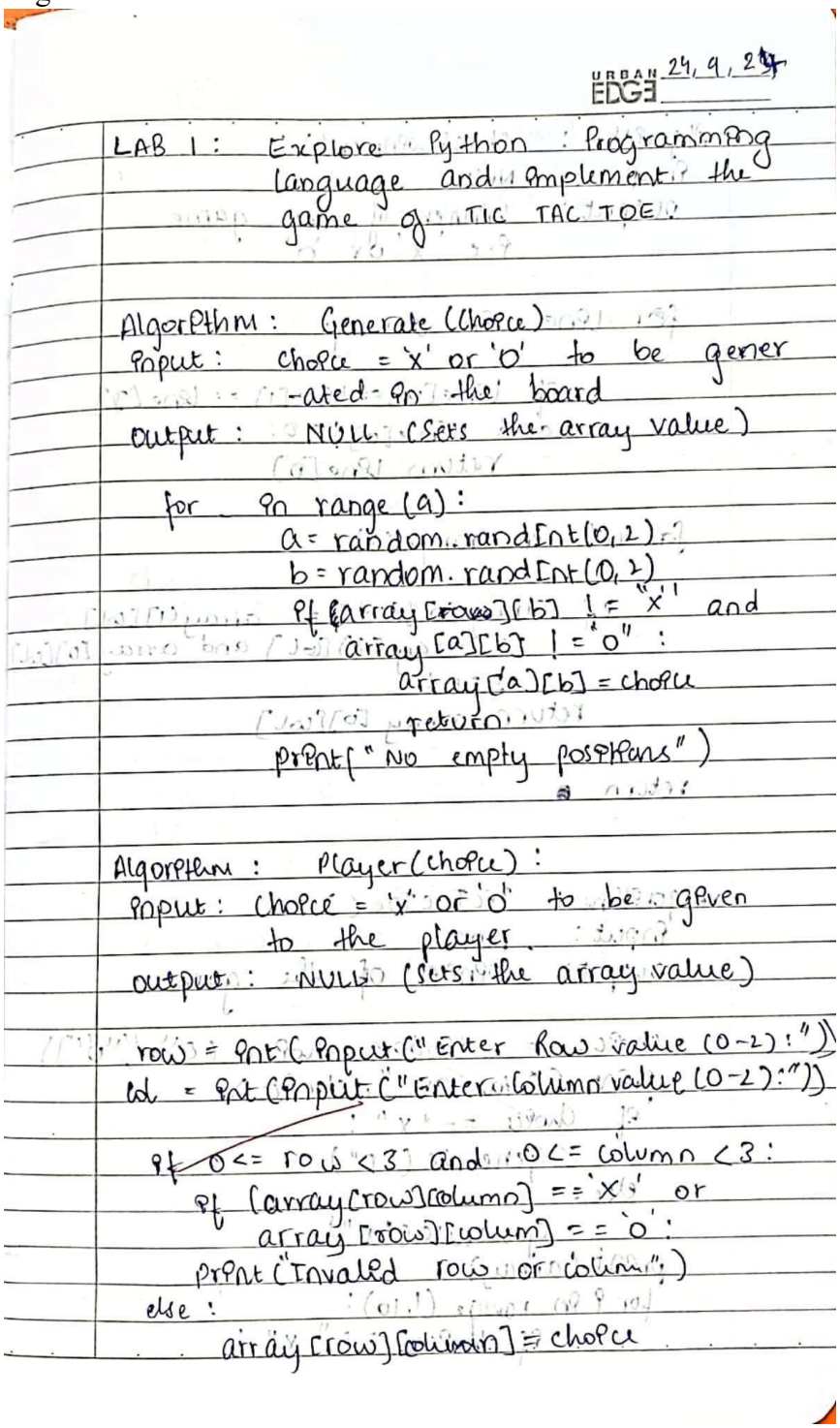
Github Link: GITHUB LINK

Program 1

Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Algorithm:



URBAN EDGE 24, 9, 2[illegible]

LAB 1: Explore Python : Programming language and implement the game of TIC TAC TOE.

Algorithm: Generate (choice)
Input: choice = 'x' or 'o' to be gener-ated on the board
output: NULL (sets the array value)

for in range (9):
a = random.randint(0,2)
b = random.randint(0,2)
if [array[row][b] != 'x' and
array[a][b] != 'o':
array[a][b] = choice
return
print("No empty positions")

Algorithm: Player(choice):
Input: choice = 'x' or 'o' to be given to the player.
output: NULL (sets the array value)

row = int(input("Enter Row value (0-2):"))
cl = int(input("Enter Column value (0-2):"))

if 0 <= row < 3 and 0 <= column < 3:
if (array[row][column] == 'x' or
array[row][column] == 'o':
print("Invalid row or column")
else:
array[row][column] = choice

URBAN EDGE

Algorithm : checkWinner():
Input : Null
output : winner of the game
p.e 'x' or 'o'

for lene in array:
    if lene[0] == lene[1] == lene[2]
    and lene[0] != 0:
        return lene[0]

for col in range(3):
    if array[0][col] == array[1][col]
    == array[2][col] and array[0][col]
    != 0:
        return array[0][col]

return

Algorithm : TicTacToe()
Input : None
output : winner of the game

choice = random.choice(["x","o"])
print("You are", choice)
if choice == "x":
    generate("o")
else:
    generate("x")
print(array)
for i in range(1,10):
    if choice == "x":

```
        generate ("O")
    else
        generate ("X")
    player (choice)
    print (array)

    winner = checkWinner()
    if winner:
        print ("The winner is: {winner}")
        break
else:
    print ("It a draw!")
```

24/9/2024

OUTPUT:

You are O

```
[[ O  X  O ]
 [ X  O  X ]
 [ X  O  O ]]
```

The winner is O.

URBAN EDGE 1/10/24

LAB 2 : Algorithm for Automated Vacuum cleaner. (2 & 4 QUADRANTS)

Algorithm : 2 Quads :

1) Input : Read room_A & room_B states
2) Initialize :
   → Set current_location to 'A'.
   → set total_cost to 0.
   → Define goal_state as both rooms being clean.
3) Define clean_room function :
   → If current_location is 'A':
      → If room_A is dirty (0) :
         clean room_A
         Add 2 to cost
         move to room_B
   → If current_location is 'B':
      If room_B is dirty (0):
         clean room_B
         Add 2 to cost
4) Define check_goal_state function :
   → If both rooms are clean, return goal achieved.
   → otherwise, call clean_room.
5) Run loop :
   → continue checking and cleaning
6) output : Display total cost and final state.

URBAN EDGE

output:
enter state for A : 1
enter state for B : 0
vacuum is placed in location A
location A is Dirty
Cleaning location A
cost for Suck : 2 units
moving right to location B
vacuum is placed in location B
location B is already clean
Both rooms are clean!
Goal state reached : ['A', 0, 'B', 0]
total cost : 2 units.

Algorithm: 4 Quadrants:

1) Input : Read room A & room B states.
room C & room D states

2) Initialize :
→ Set current_location to either 'A', 'B', 'C' or 'D'.
→ Set total_cost to 0.
→ Define goal state as both room being clean.

Input start room as A.

3) Define clean_room function :
→ If current_location is 'A':
If room A is dirty:
clean room A
Add 2 to cost
→ move to a+1 or a-1

URBAN EDGE

4) Define check_goal_status function:
   → If all rooms are clean, return goal achieved.
   → Otherwise, call clean_room

5) Run loop:
   → continuous checking for change in state.

6) Output: Display total cost and final state.

output:

Enter A state 1
Enter B state 0
Enter C state 0
Enter D state 1
Vacuum is placed in A
Location A is dirty
Cost for 'SUCK' : 2 units
Moving to B
Vacuum is placed in B
B is clean
Moving to C
Vacuum is placed in C
C is clean
Moving to D
Vacuum is placed in D
Location D is dirty
Cost for 'SUCK' : 2 units
All rooms are clean
Goal state : [A, 0, B, 0, C, 0, D, 0,]
Total cost : 4

3/10/2024

Code: TIC TAC TOE

```
import numpy as np
import random

array = np.full((3, 3), '-', dtype=object)

def generate(choice):
    for _ in range(9):
        a = random.randint(0, 2)
        b = random.randint(0, 2)
        if array[a][b] != "X" and array[a][b] != "O":
            array[a][b] = choice
            return
    print("No empty positions left! It's a draw")

def player(choice):
    row = int(input("Enter the Row Value (0-2): "))
    column = int(input("Enter the Column Value (0-2): "))
    if 0 <= row < 3 and 0 <= column < 3:
        if array[row][column] == "X" or array[row][column] == "O":
            print("Invalid row or column index. Position already taken.")
        else:
            array[row][column] = choice
    else:
        print("Invalid row or column index.")

def check_winner():
    for line in array:
        if line[0] == line[1] == line[2] and line[0] != '-':
            return line[0]
    for col in range(3):
        if array[0][col] == array[1][col] == array[2][col] and array[0][col] != '-':
            return array[0][col]
    if array[0][0] == array[1][1] == array[2][2] and array[0][0] != '-':
        return array[0][0]
    if array[0][2] == array[1][1] == array[2][0] and array[0][2] != '-':
        return array[0][2]
    return 0

choice = random.choice(["X", "O"])
print("You are", choice)

if choice == "X":
    generate("O")
else:
    generate("X")
```

```
print(array)

for i in range(1, 10):
    player(choice)
    if choice == "X":
        generate("O")
    else:
        generate("X")
    print(array)
    winner = check_winner()
    if winner != 0:
        print(f"The winner is: {winner}")
        break
```

OUTPUT:

```
You are X
[['-' 'O' '-']
 ['-' '-' '-']
 ['-' '-' '-']]
Enter the Row Value (0-2): 0
Enter the Column Value (0-2): 0
[['X' 'O' '-']
 ['-' '-' '-']
 ['-' '-' '-']]
[['X' 'O' '-']
 ['-' 'O' '-']
 ['-' '-' '-']]
Enter the Row Value (0-2): 1
Enter the Column Value (0-2): 1
[['X' 'O' '-']
 ['-' 'X' '-']
 ['-' '-' '-']]
[['X' 'O' '-']
 ['-' 'X' '-']
 ['-' 'O' '-']]
Enter the Row Value (0-2): 2
Enter the Column Value (0-2): 2
[['X' 'O' '-']
 ['-' 'X' '-']
 ['-' '-' 'X']]
The winner is: X
```

Code: VACCUUM WORLD (2 QUADS)

```
#2QUADS
room_A = int(input("Enter the state of Room A (1 for Dirty, 0 for Clean): "))
room_B = int(input("Enter the state of Room B (1 for Dirty, 0 for Clean): "))
current_location = 'A'
goal_state = ['A', 0, 'B', 0]
total_cost = 0

# Step 4: Define a function to clean the room
def clean_room():
    global room_A, room_B, current_location, total_cost
    # Check the current location
    if current_location == 'A':
        print("Vacuum is placed in Location A")
        if room_A == 1:
            print("Location A is Dirty")
            print("Cleaning Location A...")
            room_A = 0 # Clean the room
            total_cost += 2
            print("Location A has been Cleaned")
            print("COST for SUCK in Location A: 2 units")
        else:
            print("Location A is already clean")
        # Move to Location B (no cost for moving)
        print("Moving right to Location B")
        current_location = 'B'
    if current_location == 'B':
        print("Vacuum is placed in Location B")
        if room_B == 1:
            print("Location B is Dirty")
            print("Cleaning Location B...")
            room_B = 0 # Clean the room
            total_cost += 2 # Cost of cleaning (SUCK action)
            print("Location B has been Cleaned")
            print("COST for SUCK in Location B: 2 units")
        else:
            print("Location B is already clean")

# Step 5: Keep checking for the goal state
def check_goal_state():
    if room_A == goal_state[1] and room_B == goal_state[3]:
        print("Both rooms are clean! Goal state reached:", goal_state)
        return True # Goal state reached, stop the vacuum cleaner
    else:
        clean_room() # Continue cleaning until the goal state is reached
        return False
```

```
# Step 6: Run the vacuum cleaner
goal_reached = False # A flag to track if the goal is reached
while not goal_reached: # Loop until the goal state is reached
    goal_reached = check_goal_state()

# Final output after cleaning
print("Final goal state:", goal_state)
print("Total cost incurred:", total_cost, "units")
```

OUTPUT:

```
Enter the state of Room A (1 for Dirty, 0 for Clean): 1
Enter the state of Room B (1 for Dirty, 0 for Clean): 1
Vacuum is placed in Location A
Location A is Dirty
Cleaning Location A...
Location A has been Cleaned
COST for SUCK in Location A: 2 units
Moving right to Location B
Vacuum is placed in Location B
Location B is Dirty
Cleaning Location B...
Location B has been Cleaned
COST for SUCK in Location B: 2 units
Both rooms are clean! Goal state reached: ['A', 0, 'B', 0]
Final goal state: ['A', 0, 'B', 0]
Total cost incurred: 4 units
```

Code: VACCUUM WORLD(4 QUADS)

```
# 4-Quads Vacuum Cleaner Problem
room_A = int(input("Enter the state for room A (1 for Dirty, 0 for Clean): "))
room_B = int(input("Enter the state for room B (1 for Dirty, 0 for Clean): "))
room_C = int(input("Enter the state for room C (1 for Dirty, 0 for Clean): "))
room_D = int(input("Enter the state for room D (1 for Dirty, 0 for Clean): "))

current_location = 'A'
goal_state = ['A', 0, 'B', 0, 'C', 0, 'D', 0]
total_cost = 0

def clean_room():
    global room_A, room_B, room_C, room_D, current_location, total_cost
    if current_location == 'A':
        print("Vacuum is placed in Location A")
        if room_A == 1:
            print("Location A is Dirty")
            print("Cleaning Location A...")
            room_A = 0
```

```
            total_cost += 2
            print("Location A has been Cleaned")
            print("COST for SUCK in Location A: 2 units")
        else:
            print("Location A is already clean")
        print("Moving right to Location B")
        current_location = 'B'
    elif current_location == 'B':
        print("Vacuum is placed in Location B")
        if room_B == 1:
            print("Location B is Dirty")
            print("Cleaning Location B...")
            room_B = 0
            total_cost += 2
            print("Location B has been Cleaned")
            print("COST for SUCK in Location B: 2 units")
        else:
            print("Location B is already clean")
        print("Moving down to Location C")
        current_location = 'C'
    elif current_location == 'C':
        print("Vacuum is placed in Location C")
        if room_C == 1:
            print("Location C is Dirty")
            print("Cleaning Location C...")
            room_C = 0
            total_cost += 2
            print("Location C has been Cleaned")
            print("COST for SUCK in Location C: 2 units")
        else:
            print("Location C is already clean")
        print("Moving right to Location D")
        current_location = 'D'
    elif current_location ==  'D':
        print("Vacuum is placed in Location D")
        if room_D == 1:
            print("Location D is Dirty")
            print("Cleaning Location D...")
            room_D = 0
            total_cost += 2
            print("Location D has been Cleaned")
            print("COST for SUCK in Location D: 2 units")
        else:
            print("Location D is already clean")

def check_goal_state():
    if room_A == goal_state[1] and room_B == goal_state[3] and room_C == goal_state[5] and
```

```
room_D == goal_state[7]:
        print("All rooms are clean! Goal state reached:", goal_state)
        return True
    else:
        clean_room()
        return False

goal_reached = False
while not goal_reached:
    goal_reached = check_goal_state()

print("Final goal state:", goal_state)
print("Total cost incurred:", total_cost, "units")
```

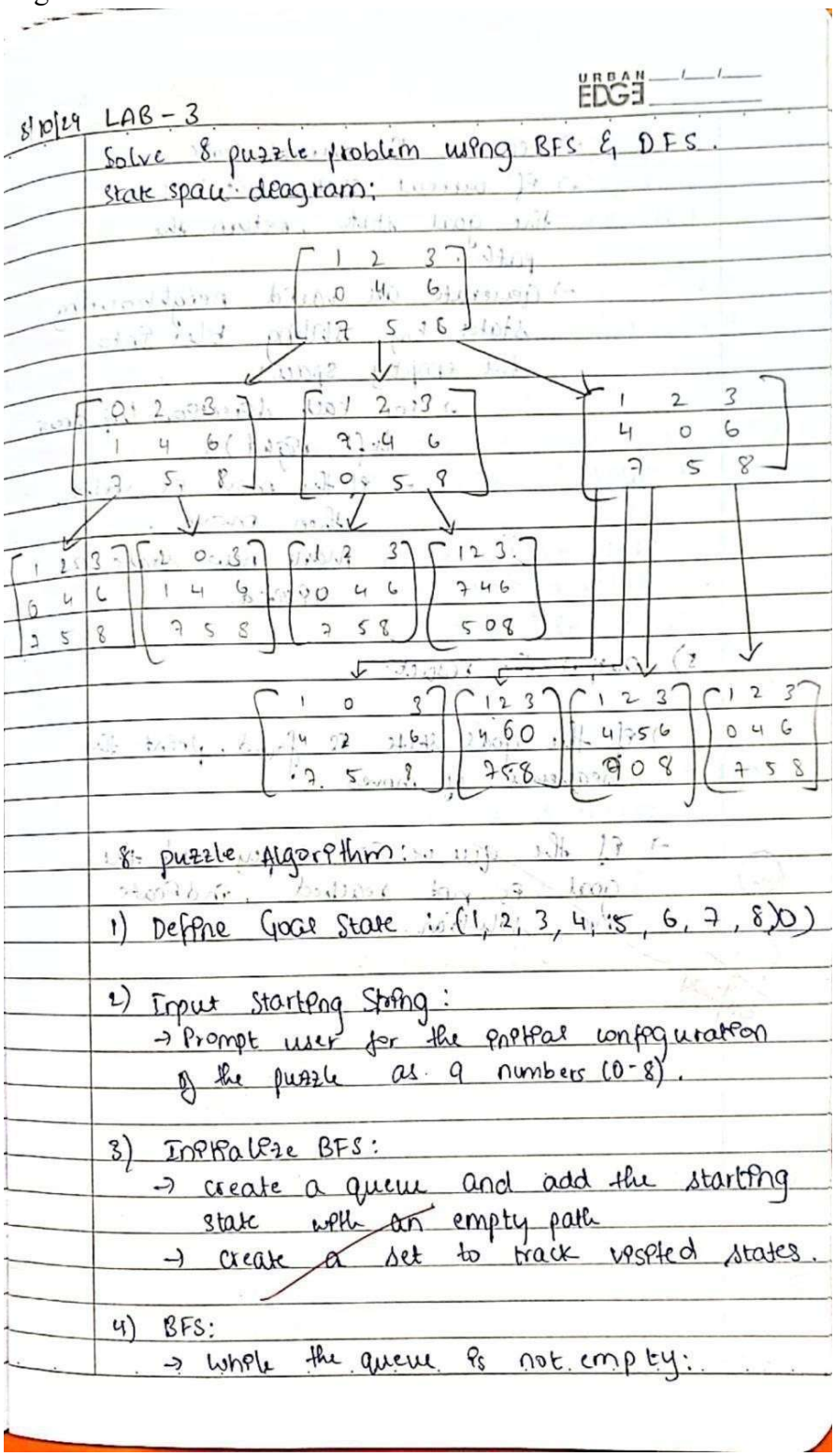
OUTPUT:

```
Enter the state for room A (1 for Dirty, 0 for Clean): 1
Enter the state for room B (1 for Dirty, 0 for Clean): 0
Enter the state for room C (1 for Dirty, 0 for Clean): 1
Enter the state for room D (1 for Dirty, 0 for Clean): 1
Vacuum is placed in Location A
Location A is Dirty
Cleaning Location A...
Location A has been Cleaned
COST for SUCK in Location A: 2 units
Moving right to Location B
Vacuum is placed in Location B
Location B is already clean
Moving down to Location C
Vacuum is placed in Location C
Location C is Dirty
Cleaning Location C...
Location C has been Cleaned
COST for SUCK in Location C: 2 units
Moving right to Location D
Vacuum is placed in Location D
Location D is Dirty
Cleaning Location D...
Location D has been Cleaned
COST for SUCK in Location D: 2 units
All rooms are clean! Goal state reached: ['A', 0, 'B', 0, 'C', 0, 'D', 0]
Final goal state: ['A', 0, 'B', 0, 'C', 0, 'D', 0]
Total cost incurred: 6 units
```

Program 2

SOLVE 8 PUZZLE PROBLEM USING BFS AND DFS

Algorithm:



→ Dequeue the current state
→ If current state matches the goal state, return the path.
→ Generate all valid neighbouring states by sliding tiles into the empty space:
→ For each direction (up, down, left, right):
→ If the move is valid, then move.
→ Mark new state as visited.

5) Output the result:

→ If the goal state is found, print the sequence of moves.

→ If the queue is empty and the goal is not reached, indicate no solution.

10

5/10/2024

Code:

```
from collections import deque

GOAL_STATE = (1, 2, 3, 4, 5, 6, 7, 8, 0)

def find_empty(state):
    return state.index(0)

def get_neighbors(state):
    neighbors = []
    empty_index = find_empty(state)
    row, col = divmod(empty_index, 3)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for dr, dc in directions:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_index = new_row * 3 + new_col
            new_state = list(state)
            new_state[empty_index], new_state[new_index] = new_state[new_index],
new_state[empty_index]
            neighbors.append(tuple(new_state))
    return neighbors

def bfs(initial_state):
    queue = deque([(initial_state, [])])
    visited = set()
    visited.add(initial_state)
    visited_count = 1  # Initialize visited count
    while queue:
        current_state, path = queue.popleft()
        if current_state == GOAL_STATE:
            return path, visited_count # Return path and count
        for neighbor in get_neighbors(current_state):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, path + [neighbor]))
                visited_count += 1  # Increment visited count
    return None, visited_count  # Return count if no solution found

def input_start_state():
    print("Enter the starting state as 9 numbers (0 for the empty space):")
    input_state = input("Format: 1 2 3 4 5 6 7 8 0\n")
    numbers = list(map(int, input_state.split()))
    if len(numbers) != 9 or set(numbers) != set(range(9)):
        print("Invalid input. Please enter numbers from 0 to 8 with no duplicates.")
        return input_start_state()
```

```
    return tuple(numbers)

def print_matrix(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])

if__name__== "_main_":
    initial_state = input_start_state()
    print("Initial state:")
    print_matrix(initial_state)
    solution, visited_count = bfs(initial_state)
    print(f"Number of states visited: {visited_count}")
    if solution:
        print("\nSolution found with the following steps:")
        for step in solution:
            print_matrix(step)
            print()
    else:
        print("No solution found.")
```

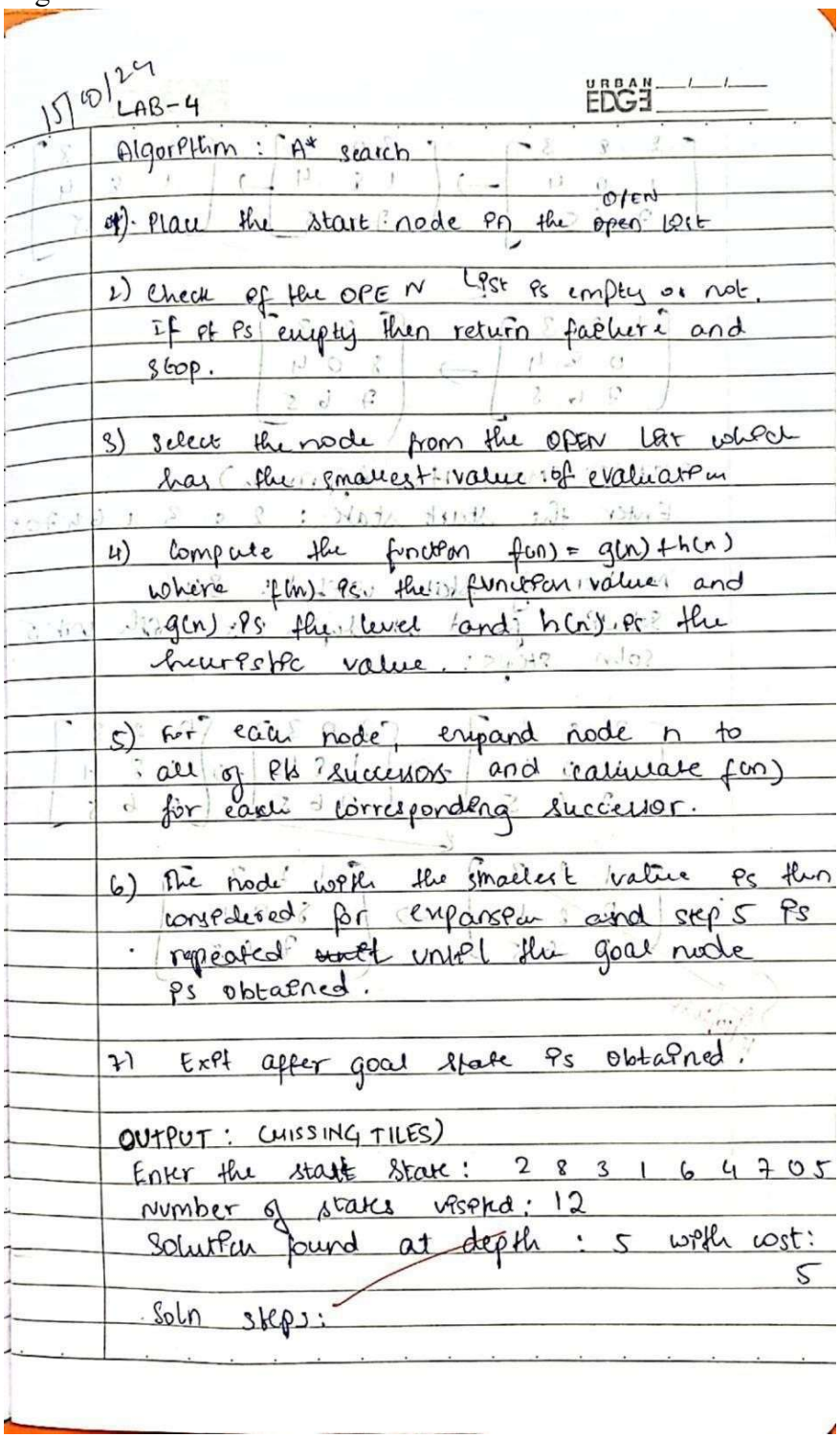
OUTPUT:

```
Enter the starting state as 9 numbers (0 for the empty space):
Format: 1 2 3 4 5 6 7 8 0
Initial state:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
Number of states visited: 3

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

Program 3

Implement A* SEARCH TO SOLVE 8 PUZZLE PROBLEM (MISSING TILES AND MANHATTAN DISTANCE APPROACH)

Algorithm:



15/10/24

LAB-4

URBAN EDGE

Algorithm : A* search

1). Place the start node on the OPEN open list

2) Check if the OPEN list is empty or not. If it is empty then return failure and stop.

3) Select the node from the OPEN list which has the smallest value of evaluation

4) Compute the function f(n) = g(n) + h(n) where f(n) is the function value and g(n) is the level and h(n) is the heuristic value.

5) For each node, expand node n to all of its successors and calculate f(n) for each corresponding successor.

6) The node with the smallest value is then considered for expansion and step 5 is repeated until the goal node is obtained.

7) Exit after goal state is obtained.

OUTPUT : (MISSING TILES)

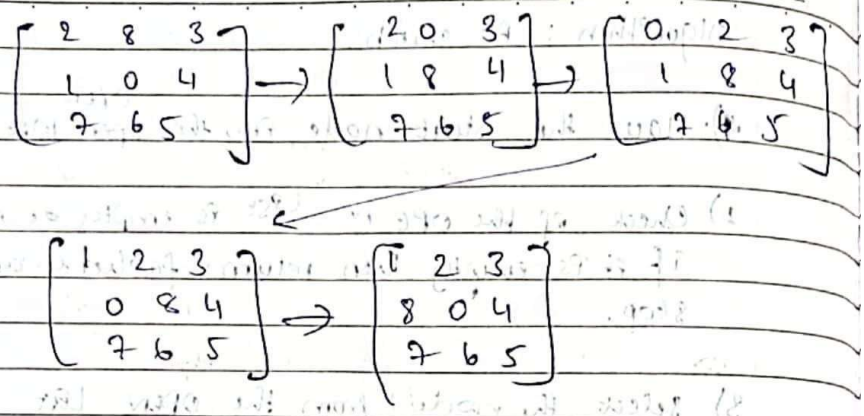
Enter the start State : 2 8 3 1 6 4 7 0 5

Number of states visited : 12

Solution found at depth : 5 with cost: 5

Soln steps :

URBAN EDGE



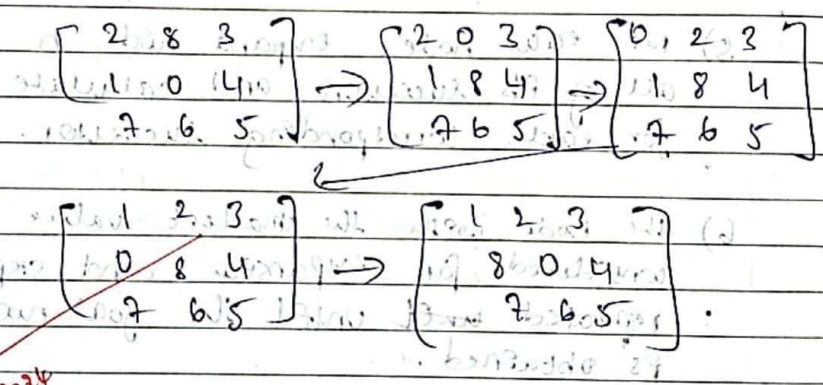
OUTPUT: (Manhatten Approach)

Enter the start state : 2 8 3 1 6 4 7 0 5

Number of states visited : 34

Solution found at depth : 5 path cost : 5

Soln steps:



15/10/2024

Code: MISSING TILES APPROACH

```
import heapq

# Define the goal state as a tuple
GOAL_STATE = (1, 2, 3, 8, 0, 4, 7, 6, 5)

# Function to find the index of the empty space (0)
def find_empty(state):
    return state.index(0)

# Heuristic function that counts the number of mismatched tiles
def mismatched_tiles(state):
    return sum(1 for i, tile in enumerate(state) if tile != 0 and tile != GOAL_STATE[i])

# Function to get neighbors (all possible moves from current state)
def get_neighbors(state):
    neighbors = []
    empty_index = find_empty(state)
    row, col = divmod(empty_index, 3)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # up, down, left, right
    for dr, dc in directions:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_index = new_row * 3 + new_col
            new_state = list(state)
            new_state[empty_index], new_state[new_index] = new_state[new_index],
new_state[empty_index]
            neighbors.append(tuple(new_state))
    return neighbors

# A* algorithm implementation using mismatched tiles heuristic
def a_star(initial_state):
    priority_queue = []
    heapq.heappush(priority_queue, (0, initial_state, [])) # (priority, state, path)
    visited = set()
    visited.add(initial_state)
    visited_count = 1  # Initialize visited count
    cost_so_far = {initial_state: 0} # Tracks the cost to reach each state

    while priority_queue:
        priority, current_state, path = heapq.heappop(priority_queue)

        if current_state == GOAL_STATE:
            depth = len(path)
            return path, visited_count, depth, cost_so_far[current_state]

        # Explore neighbors
```

```
        for neighbor in get_neighbors(current_state):
            new_cost = cost_so_far[current_state] + 1  # Cost to move to the neighbor

            if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                cost_so_far[neighbor] = new_cost
                priority = new_cost + mismatched_tiles(neighbor) # A* f = g + h
                heapq.heappush(priority_queue, (priority, neighbor, path + [neighbor]))
                if neighbor not in visited:
                    visited.add(neighbor)
                    visited_count += 1  # Increment visited count

    return None, visited_count, 0, 0  # No solution found

# Function to take user input for the initial state
def input_start_state():
    print("Enter the starting state as 9 numbers (0 for the empty space):")
    input_state = input("Format: 1 2 3 4 5 6 7 8 0\n")
    numbers = list(map(int, input_state.split()))
    if len(numbers) != 9 or set(numbers) != set(range(9)):
        print("Invalid input. Please enter numbers from 0 to 8 with no duplicates.")
        return input_start_state()
    return tuple(numbers)

# Function to print the state as a 3x3 matrix
def print_matrix(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])

# Main function
if __name__ == "_main_":
    initial_state = input_start_state()
    print("Initial state:")
    print_matrix(initial_state)

    # Run A* algorithm
    solution, visited_count, depth, cost = a_star(initial_state)

    print(f"Number of states visited: {visited_count}")
    if solution:
        print(f"Solution found at depth: {depth} with cost: {cost}")
        print("\nSolution steps:")
        for step in solution:
            print_matrix(step)
            print()
    else:
        print("No solution found.")
```

OUTPUT:

```
Enter the starting state as 9 numbers (0 for the empty space):
Format: 1 2 3 4 5 6 7 8 0
2 8 3 1 6 4 7 0 5
Initial state:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)
Number of states visited: 12
Solution found at depth: 5 with cost: 5

Solution steps:
(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)
```

Code: MANHATTAN DISTANCE APPROACH

```
import heapq

# Define the goal state as a tuple
GOAL_STATE = (1, 2, 3, 8, 0, 4, 7, 6, 5)

# Function to find the index of the empty space (0)
def find_empty(state):
    return state.index(0)

# Heuristic function that calculates the Manhattan distance
def manhattan_distance(state):
    distance = 0
    for i, tile in enumerate(state):
        if tile != 0: # Skip the blank tile (0)
            # Calculate the correct position (row, col) in the goal state
```

```
            correct_row, correct_col = divmod(tile - 1, 3)
            # Calculate the current position (row, col)
            current_row, current_col = divmod(i, 3)
            # Add the Manhattan distance for this tile
            distance += abs(correct_row - current_row) + abs(correct_col - current_col)
    return distance

# Function to get neighbors (all possible moves from current state)
def get_neighbors(state):
    neighbors = []
    empty_index = find_empty(state)
    row, col = divmod(empty_index, 3)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # up, down, left, right
    for dr, dc in directions:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_index = new_row * 3 + new_col
            new_state = list(state)
            new_state[empty_index], new_state[new_index] = new_state[new_index],
new_state[empty_index]
            neighbors.append(tuple(new_state))
    return neighbors

# A* algorithm implementation using Manhattan distance heuristic
def a_star(initial_state):
    priority_queue = []
    heapq.heappush(priority_queue, (0, initial_state, [])) # (priority, state, path)
    visited = set()
    visited.add(initial_state)
    visited_count = 1  # Initialize visited count
    cost_so_far = {initial_state: 0} # Tracks the cost to reach each state

    while priority_queue:
        priority, current_state, path = heapq.heappop(priority_queue)

        if current_state == GOAL_STATE:
            depth = len(path)
            return path, visited_count, depth, cost_so_far[current_state]

        # Explore neighbors
        for neighbor in get_neighbors(current_state):
            new_cost = cost_so_far[current_state] + 1  # Cost to move to the neighbor

            if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                cost_so_far[neighbor] = new_cost
                priority = new_cost + manhattan_distance(neighbor) # A* f = g + h
                heapq.heappush(priority_queue, (priority, neighbor, path + [neighbor]))
```

```
            if neighbor not in visited:
                visited.add(neighbor)
                visited_count += 1  # Increment visited count

    return None, visited_count, 0, 0  # No solution found

# Function to take user input for the initial state
def input_start_state():
    print("Enter the starting state as 9 numbers (0 for the empty space):")
    input_state = input("Format: 1 2 3 4 5 6 7 8 0\n")
    numbers = list(map(int, input_state.split()))
    if len(numbers) != 9 or set(numbers) != set(range(9)):
        print("Invalid input. Please enter numbers from 0 to 8 with no duplicates.")
        return input_start_state()
    return tuple(numbers)

# Function to print the state as a 3x3 matrix
def print_matrix(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])

# Main function
if __name__ == "_main_":
    initial_state = input_start_state()
    print("Initial state:")
    print_matrix(initial_state)

    # Run A* algorithm
    solution, visited_count, depth, cost = a_star(initial_state)

    print(f"Number of states visited: {visited_count}")
    if solution:
        print(f"Solution found at depth: {depth} with cost: {cost}")
        print("\nSolution steps:")
        for step in solution:
            print_matrix(step)
            print()
    else:
        print("No solution found.")
```

OUTPUT:

```
Enter the starting state as 9 numbers (0 for the empty space):
Format: 1 2 3 4 5 6 7 8 0
2 8 3 1 6 4 7 0 5
Initial state:
(2, 8, 3)
(1, 6, 4)
```

(7, 0, 5)
Number of states visited: 34
Solution found at depth: 5 with cost: 5

Solution steps:
(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)

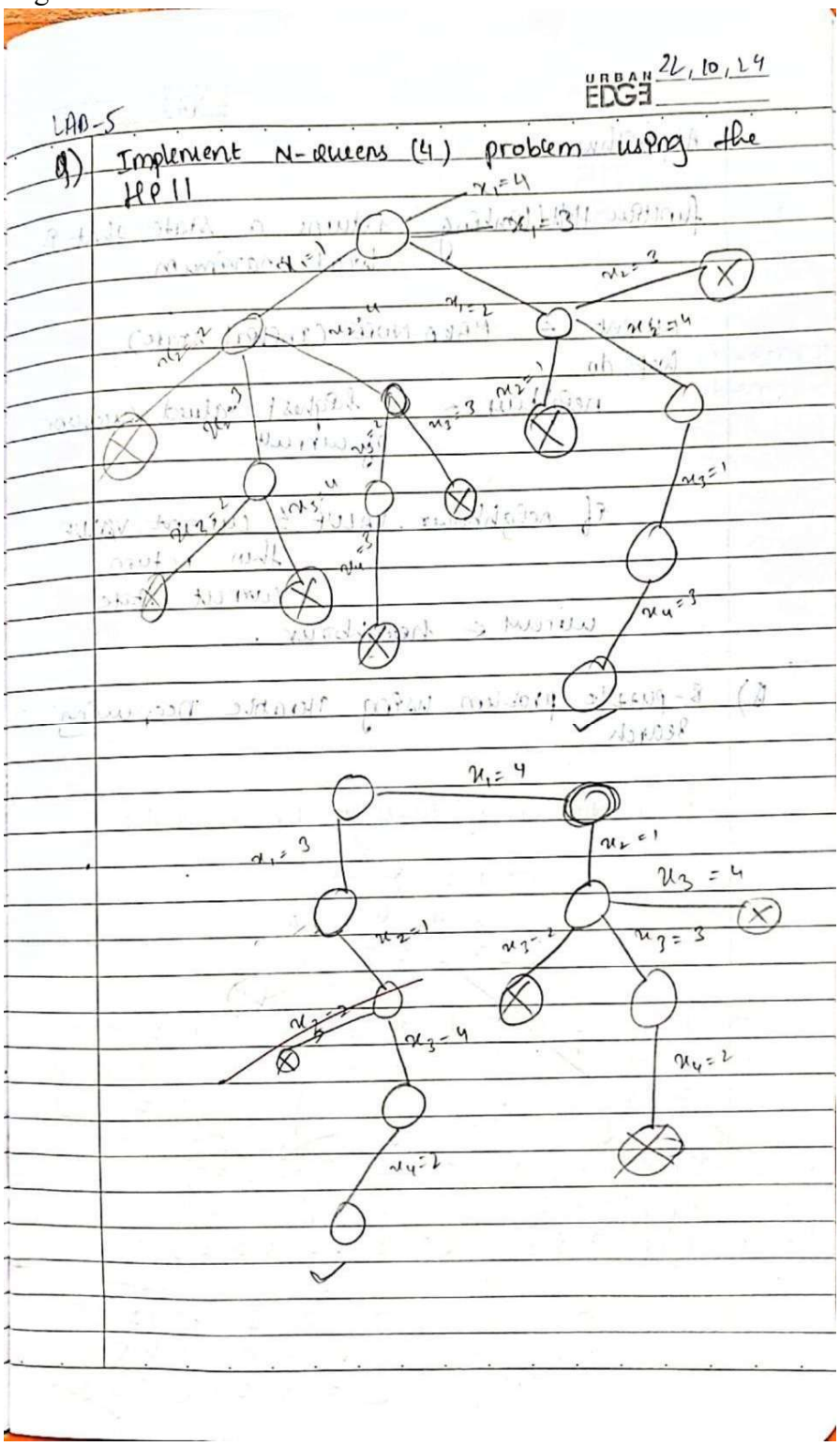
(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

(1, 2, 3)
(8, 0, 4)
(7, 6, 5)

Program 4

Implement N-QUEENS PROBLEM USING HILL CLIMBING ALGORITHM

Algorithm:



URBAN EDGE

Algorithm:

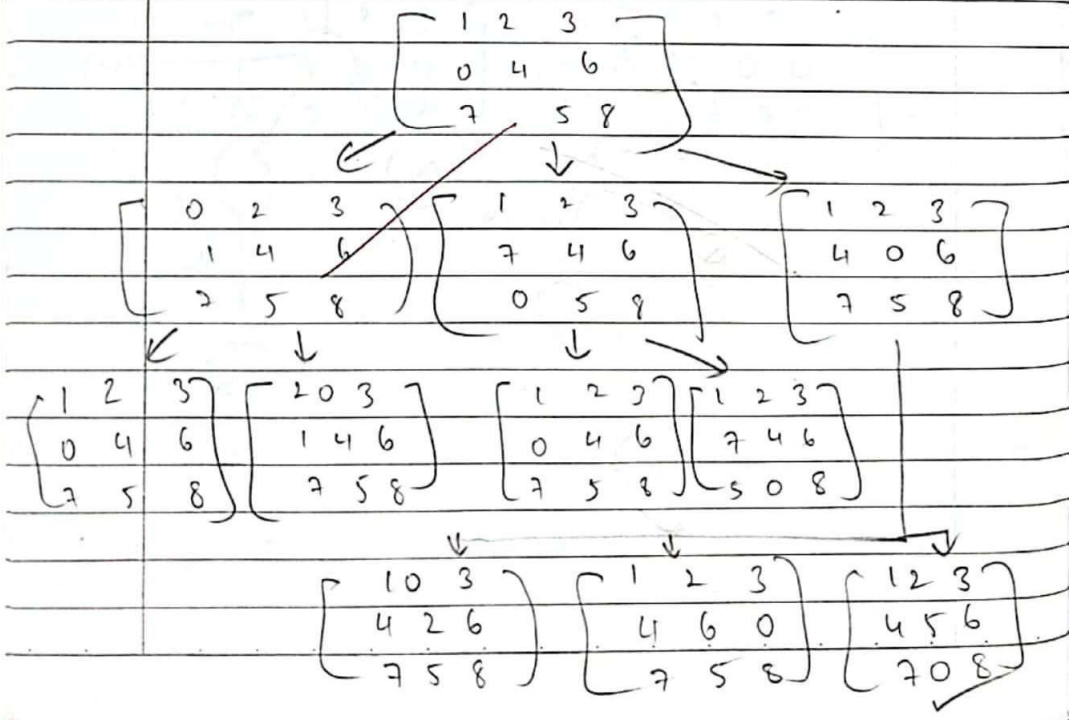
function Hill Climbing : returns a state that is local maximum

current ← MAKE-NODE (Initial state)
loop do
neighbour ← a highest valued successor of current

if neighbour.VALUE ≤ current.VALUE then return current.State
current ← neighbour.

8) 8-puzzle problem using Iterative Deepening Search

Enter Number of level to be explored : 3



URBAN EDGE

Algorithm:

function ITDS(problem) returns a soln or failure.

for depth = 0 to ∞ do
result ← DEPTH-LIMITED-SEARCH(problem, depth)
~~return result~~

Code:

```
import random

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

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

        while True:
            # Generate neighbors by moving each queen to a different position
            found_better = False

            for i in range(n):
                for j in range(n):
                    if j != current_board[i]: # Only consider different positions
                        neighbor_board  = list(current_board)
                        neighbor_board[i] = j
                        neighbor_conflicts = calculate_conflicts(neighbor_board)

                        if neighbor_conflicts < current_conflicts:
                            print_board(current_board)
                            print("Current Conflicts:", current_conflicts)
                            print_board(neighbor_board)
                            print("Neighbor Conflicts:", neighbor_conflicts)

                            current_board = neighbor_board
                            current_conflicts = neighbor_conflicts
                            cost += 1
                            found_better = True
                            break
                if found_better:
                    break
```

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

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

# Function to print the board
def print_board(board):
    n = len(board)
    for i in range(n):
        row = ['.'] * n
        row[board[i]] = 'Q'  # Place a queen
        print(' '.join(row))
    print("\n===============")

# Example Usage
if __name__ == "_main_":
    n = 4  # Change N here to test for different sizes
    solution, conflicts, cost = hill_climbing(n)
    print("Final Board Configuration:")
    print_board(solution)
    print("Number of Conflicts:", conflicts)
    print("Number of Steps (Cost):", cost)
```

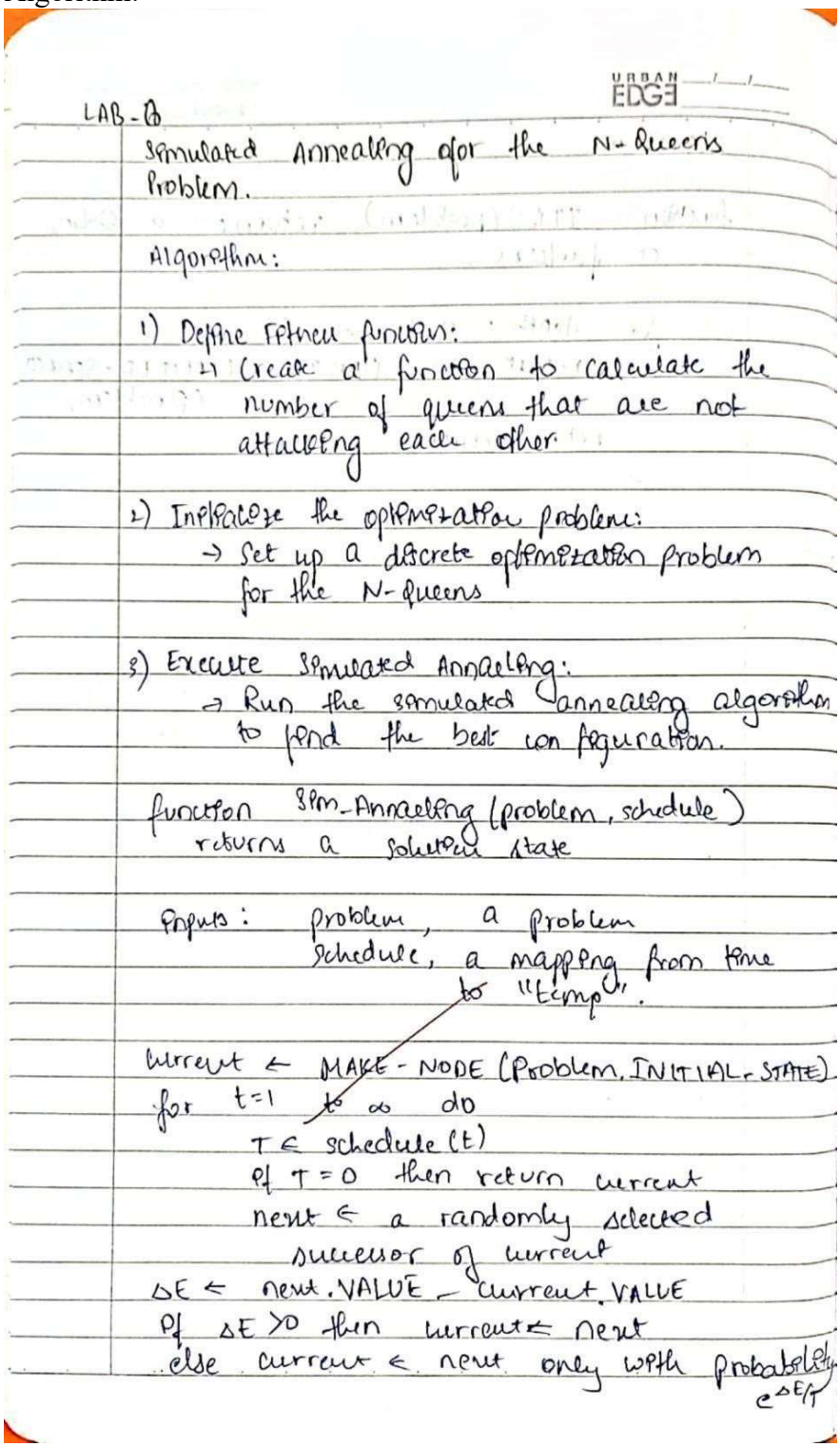
OUTPUT:

```
. Q . .
. . . Q
Q . . .
. . Q .

Current Conflicts: 3
Q . . .
. . . Q
. Q . .
. . Q .
Neighbor Conflicts: 1
Final Board Configuration:
. Q . .
. . . Q
Q . . .
. . Q .

Number of Conflicts: 0
Number of Steps (Cost): 3
```

Program 5

Implement SIMULATED ANNEALING

Algorithm:



LAB-6

Simulated Annealing for the N-Queens Problem.

Algorithm:

1) Define Fitness function:
   → Create a function to calculate the number of queens that are not attacking each other.

2) Initialize the optimization problem:
   → Set up a discrete optimization problem for the N-Queens

3) Execute Simulated Annealing:
   → Run the simulated annealing algorithm to find the best configuration.

function SIM-ANNEALING (problem, schedule)
returns a solution state

inputs: problem, a problem
schedule, a mapping from time to "temp".

current ← MAKE-NODE (Problem.INITIAL-STATE)
for t=1 to ∞ do
  T ← schedule (t)
  if T=0 then return current
  next ← a randomly selected successor of current
ΔE ← next.VALUE − current.VALUE
if ΔE > 0 then current ← next
else current ← next only with probability $e^{\Delta E/T}$

URBAN EDGE

OUTPUT:

The best position found: [5 3 6 0 7 1 4 2]
The number of queens that are not attacking each other: 8.

29/10/2024

Code:

```
!sed -i 's/from joblib.my_exceptions import WorkerInterrupt/import joblib/'
/usr/local/lib/python3.10/dist-packages/mlrose_hiive/runners/_nn_runner_base.py
import mlrose_hiive as mlrose
import numpy as np

def queens_max(position):
    no_attack_on_j = 0
    queen_not_attacking = 0
    for i in range(len(position) - 1):
        no_attack_on_j = 0
        for j in range(i + 1, len(position)):
            if (position[i] != position[j]) and (position[i] != position[j] + (j - i)) and (position[i] !=
position[j] - (j - i)):
                no_attack_on_j += 1
        if no_attack_on_j == len(position) - 1 - i:
            queen_not_attacking += 1
    if queen_not_attacking == 7:
        queen_not_attacking += 1
    return queen_not_attacking

objective = mlrose.CustomFitness(queens_max)

problem = mlrose.DiscreteOpt(length=8, fitness_fn=objective, maximize=True, max_val=8)
T = mlrose.ExpDecay()

initial_position = np.array([4, 6, 1, 5, 2, 0, 3, 7])

best_position, best_objective, _ = mlrose.simulated_annealing(
    problem=problem, schedule=T, max_attempts=500, max_iters=5000, init_state=initial_position
)

print('The best position found is:', best_position)
print('The number of queens that are not attacking each other is:', best_objective)
```

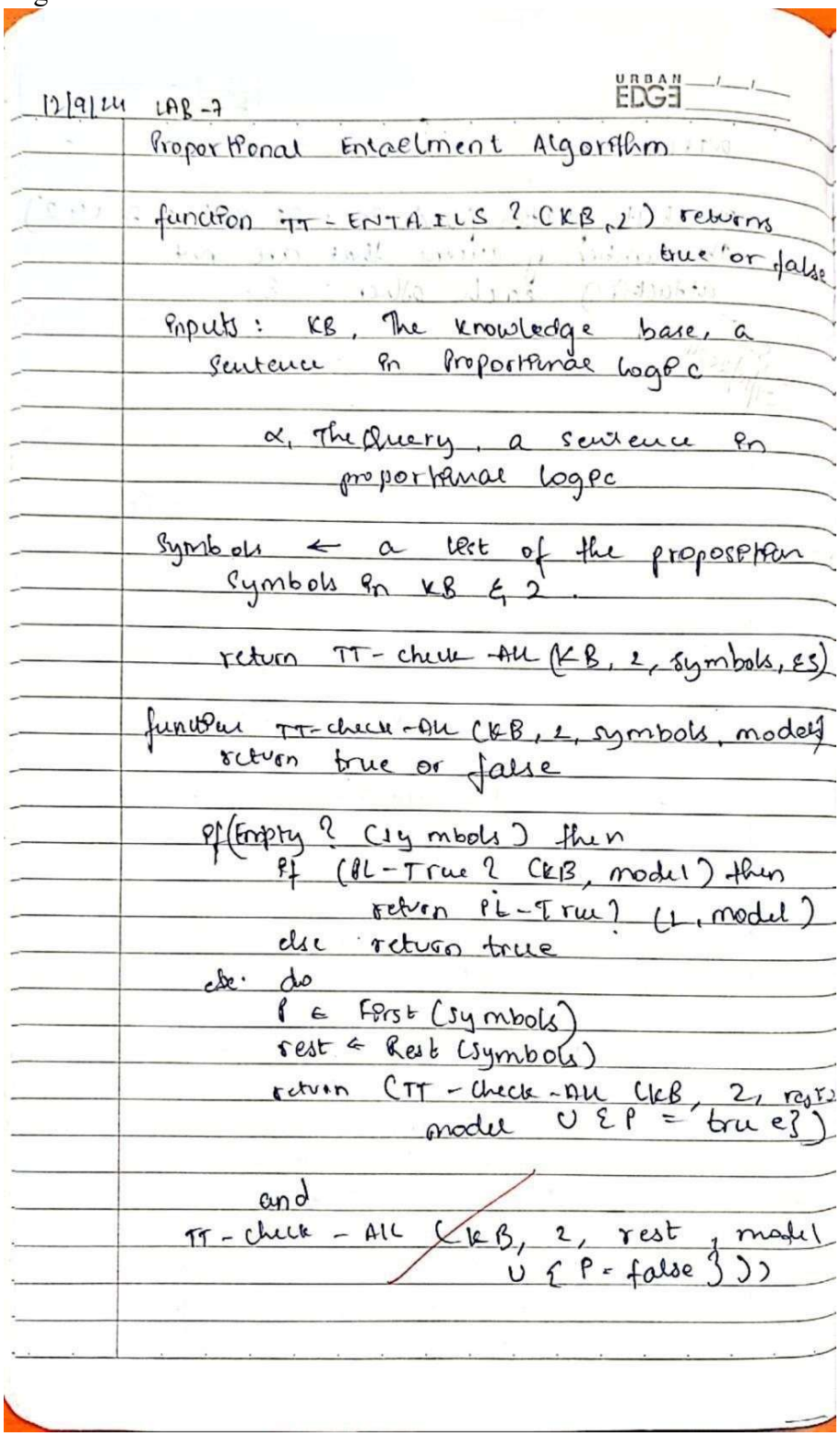
OUTPUT:

```
The best position found is: [5 2 6 1 3 7 0 4]
The number of queens that are not attacking each other is: 8.0
```

Program 6

Implement PROPORTIONAL ENTAILMENT ALGORITHM

Algorithm:



12/9/24 LAB-7

URBAN EDGE

Propositional Entailment Algorithm

function TT-ENTAILS ? (KB, α) returns true or false

Inputs : KB, The knowledge base, a sentence in propositional logic

α, The query, a sentence in propositional logic

Symbols ← a list of the proposition symbols in KB & α.

return TT-check-All (KB, α, symbols, {})

function TT-check-All (KB, α, symbols, model)
return true or false

If (Empty ? (symbols) then
If (PL-True ? (KB, model) then
return PL-True? (α, model)
else return true
else do
P ← First (symbols)
rest ← Rest (symbols)
return (TT-Check-All (KB, α, rest,
model ∪ {P = true}))

and
TT-check-All (KB, α, rest, model
∪ {P = false}))

URBAN EDGE

Propositional Inference : Enumeration Method

$\alpha = A \vee B$ $KB = (A \vee C) \wedge (B \vee \neg C)$

Checking that $KB \models \alpha$:

A	B	C	A∨C	B∨¬C	KB	α
false	f	f	f	t	f	f
f	f	t	t	f	f	f
f	t	f	f	t	f	t
f	t	t	t	t	t	t
t	f	f	t	t	t	t
t	f	t	t	f	t	t
t	t	f	t	t	t	t
t	t	t	t	t	t	t

Output : [('AND', ('OR', 'A', 'B'), ('OR', ('NOT', 'C'), 'B'))]
('OR', 'A', 'B')

Does KB entail alpha? True

14/4/2024

Code:

```
import itertools

# Function to evaluate an expression
def evaluate_expression(a, b, c, expression):
    # Use eval() to evaluate the logical expression
    return eval(expression)

# Function to generate the truth table and evaluate a logical expression
def truth_table_and_evaluation(kb, query):
    # All possible combinations of truth values for a, b, and c
    truth_values = [True, False]
    combinations = list(itertools.product(truth_values, repeat=3))
    # Reverse the combinations to start from the bottom (False -> True)
    combinations.reverse()

    # Header for the full truth table
    print(f"{'a':<5} {'b':<5} {'c':<5} {'KB':<20} {'Query':<20}")
    print("-" * 50)

    # Evaluate the expressions for each combination
    for combination in combinations:
        a, b, c = combination

        # Evaluate the knowledge base (KB) and query expressions
        kb_result = evaluate_expression(a, b, c, kb)
        query_result = evaluate_expression(a, b, c, query)

        # Convert boolean values of a, b, c, KB, and Query to strings "True"/"False"
        a_str = "True" if a else "False"
        b_str = "True" if b else "False"
        c_str = "True" if c else "False"
        kb_result_str = "True" if kb_result else "False"
        query_result_str = "True" if query_result else "False"

        # Print the results for the knowledge base and the query
        print(f"{a_str:<5} {b_str:<5} {c_str:<5} {kb_result_str:<20} {query_result_str:<20}")

    # Additional output for combinations where both KB and Query are true
    print("\nCombinations where both KB and Query are True:")
    print(f"{'a':<5} {'b':<5} {'c':<5} {'KB':<20} {'Query':<20}")
    print("-" * 50)

    # Print only the rows where both KB and Query are True
    for combination in combinations:
        a, b, c = combination
```

```
        # Evaluate the knowledge base (KB) and query expressions
        kb_result = evaluate_expression(a, b, c, kb)
        query_result = evaluate_expression(a, b, c, query)

        # If both KB and query are True, print the combination
        if kb_result and query_result:
            a_str = "True" if a else "False"
            b_str = "True" if b else "False"
            c_str = "True" if c else "False"
            kb_result_str = "True" if kb_result else "False"
            query_result_str = "True" if query_result else "False"
            print(f"{a_str:<5} {b_str:<5} {c_str:<5} {kb_result_str:<20} {query_result_str:<20}")

# Define the logical expressions as strings
kb = "(a or c) and (b or not c)"  # Knowledge Base
query = "a or b" # Query to evaluate

# Generate the truth table and evaluate the knowledge base and query
truth_table_and_evaluation(kb, query)
```

OUTPUT:

```
a     b     c     KB                  Query
-------------------------------------------------------
False False False False               False
False False True  False              False
False True  False True               True
False True  True  True               True
True  False False False              True
True  False True  False              True
True  True  False True               True
True  True  True  True               True
```

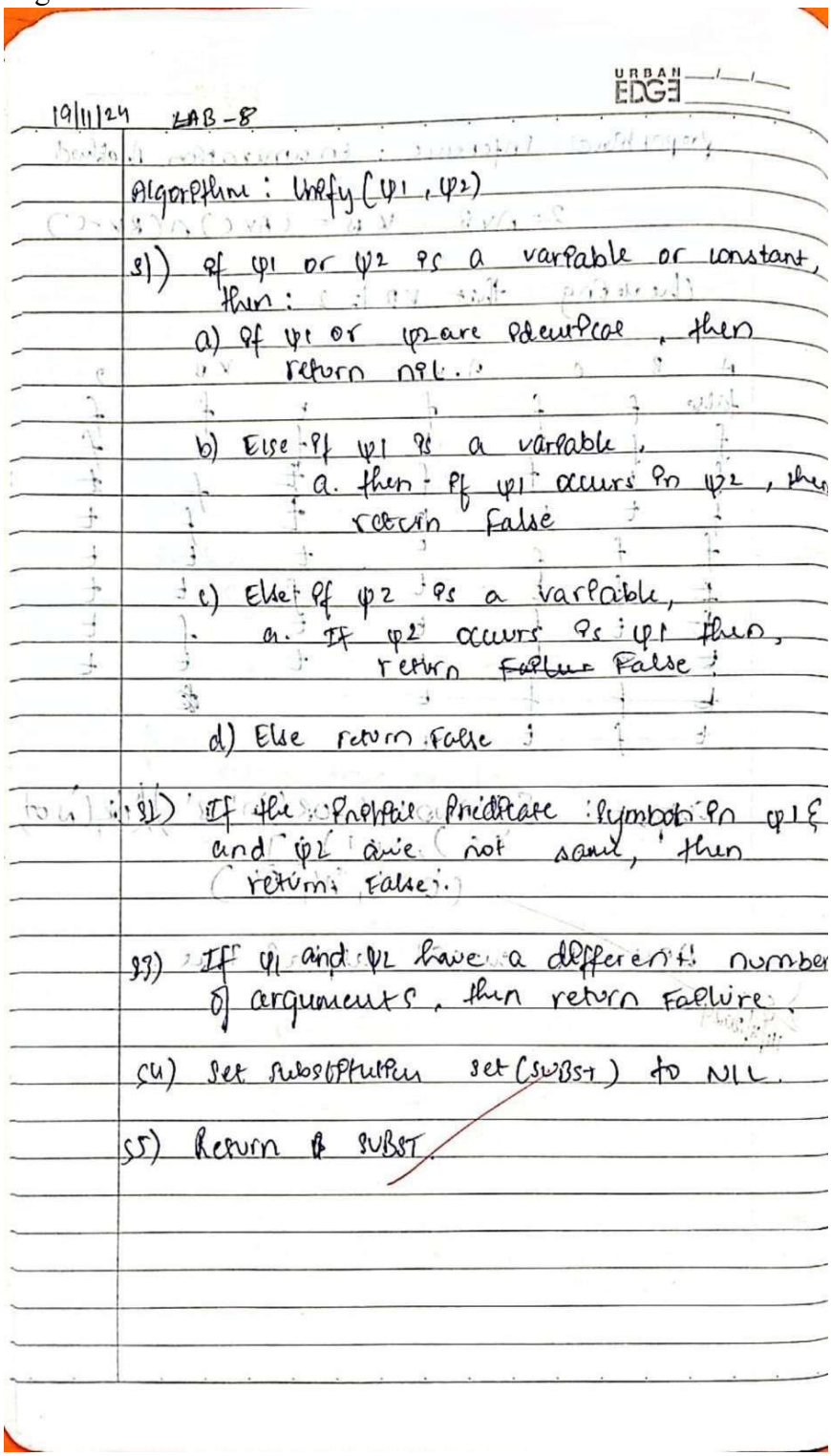
Combinations where both KB and Query are True:

```
a     b     c     KB                  Query
-------------------------------------------------------
False True  False True               True
False True  True  True               True
True  True  False True               True
True  True  True  True               True
```

Program 7

Implement UNIFICATION ALGORITHM

Algorithm:



19/11/24 LAB-8

Algorithm : Unify (ψ_1, ψ_2)

S1) If ψ_1 or ψ_2 is a variable or constant, then:

a) If ψ_1 or ψ_2 are identical, then return NIL.

b) Else if ψ_1 is a variable,

a. then if ψ_1 occurs in ψ_2, then return False

c) Else if ψ_2 is a variable,

a. If ψ_2 occurs in ψ_1 then, return ~~Failure~~ False;

d) Else return False;

S2) If the initial Predicate Symbol in ψ_1 & and ψ_2 are not same, then return False.

S3) If ψ_1 and ψ_2 have a different number of arguments, then return Failure.

S4) Set Substitution set (SUBST) to NIL.

S5) Return ~~0~~ SUBST.

URBAN EDGE

Q1) $\psi_1 = P(b, x, f(g(z)))$ — ①

$\psi_2 = P(z, f(y), f(y))$ — ④

Replace ② with b,

$P(b, f(y), f(y))^{b/z}$

Replace x in ① with f(y)

~~$P(b, f(y), f(y))$~~

$P(b, f(y), f(g(z)))^{f(y)/x}$

Replace y in ② with g(z)

$P(b, f(y), f(g(z)))^{g(z)/y}$

$= P(b, f(y), f(g(z)))$ — ③

$= P(b, f(y), f(g(z)))$ — ④

Unification possible

Q2) $\psi_1 = P(f(a), g(y))$

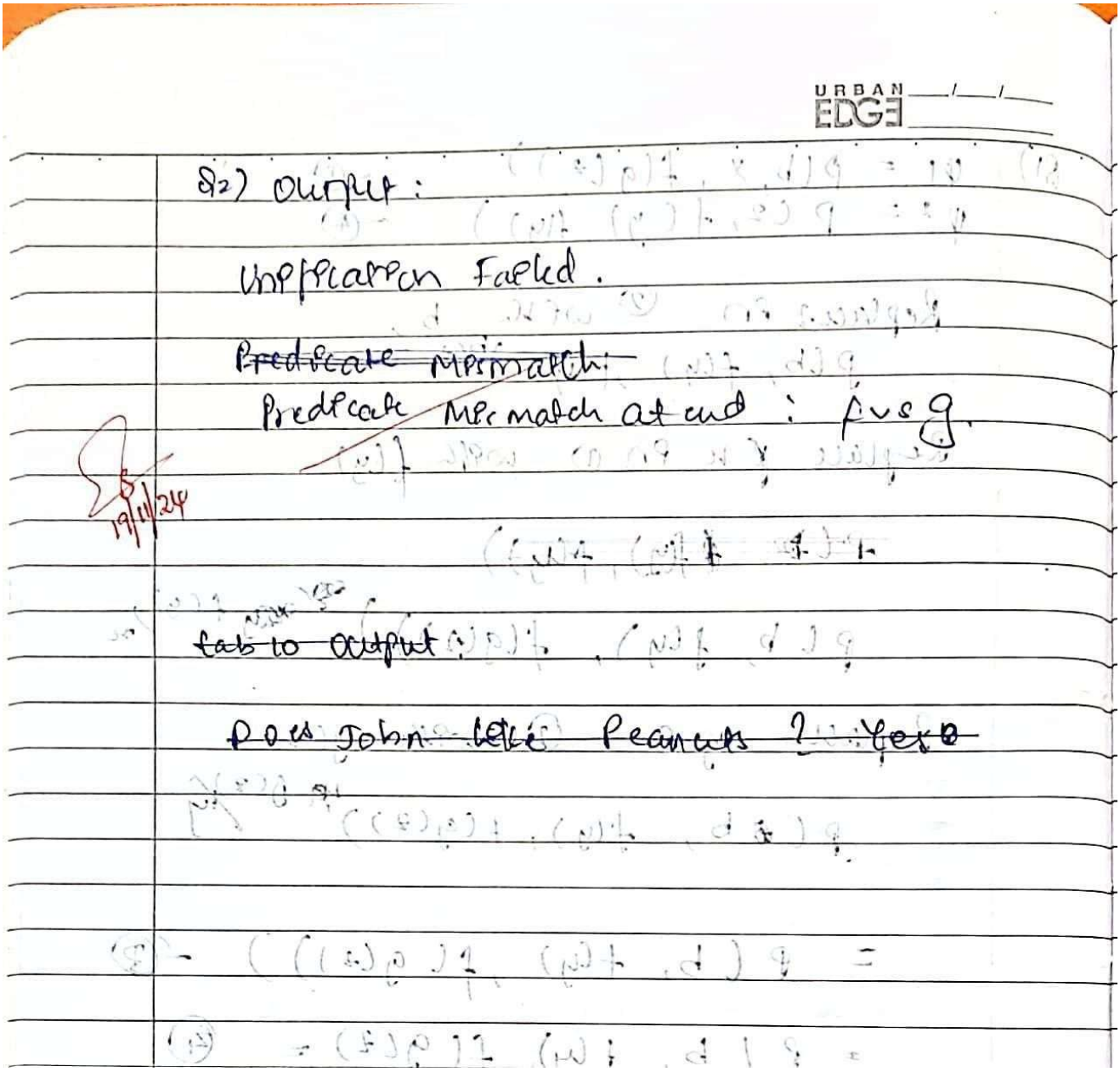
$\psi_2 = P(x, x)$

Unification not possible as x cannot be replaced.

OUTPUT:

Q1) Unification successful

Substitution : {'b': 'z', 'x': ('f', 'y'), 'y': ('g', 'z')}



Code:

```
def unify(phi1, phi2, subst={}):
    print(f"Attempting to unify: {phi1} with {phi2}")
    print(f"Current substitution: {subst}")

    if isinstance(phi1, str) and phi1.islower():
        return unify_var(phi1, phi2, subst)
    if isinstance(phi2, str) and phi2.islower():
        return unify_var(phi2, phi1, subst)

    if phi1 == phi2:
        return subst
```

```
    if isinstance(phi1, tuple) and isinstance(phi2, tuple):
        if phi1[0] != phi2[0]:
            print("Predicate mismatch:", phi1[0], "vs", phi2[0])
            return None

        if len(phi1) != len(phi2):
            print("Argument count mismatch")
            return None

        for arg1, arg2 in zip(phi1[1:], phi2[1:]):
            subst = unify(arg1, arg2, subst)
            if subst is None:
                return None
        return subst

    return None

def unify_var(var, value, subst):
    print(f"Unifying variable: {var} with value: {value}")

    if var in subst:
        return unify(subst[var], value, subst)
    elif value in subst:
        return unify(var, subst[value], subst)
    elif occurs_check(var, value, subst):
        print(f"Occurs check failed: {var} in {value}")
        return None
    else:
        subst[var] = value
        return subst

def occurs_check(var, value, subst):
    if var == value:
        return True
    elif isinstance(value, tuple):
        return any(occurs_check(var, arg, subst) for arg in value)
    elif value in subst:
        return occurs_check(var, subst[value], subst)
    return False

phi1 = ("q", "b", "x", ("f", ("g", "z")))
phi2 = ("q", "z", ("f", "y"), ("f", "y"))

result = unify(phi1, phi2)
if result:
    print("\nUnification Successful!")
```

```
    print("Substitution:", result)
else:
    print("\nUnification Failed!")
```

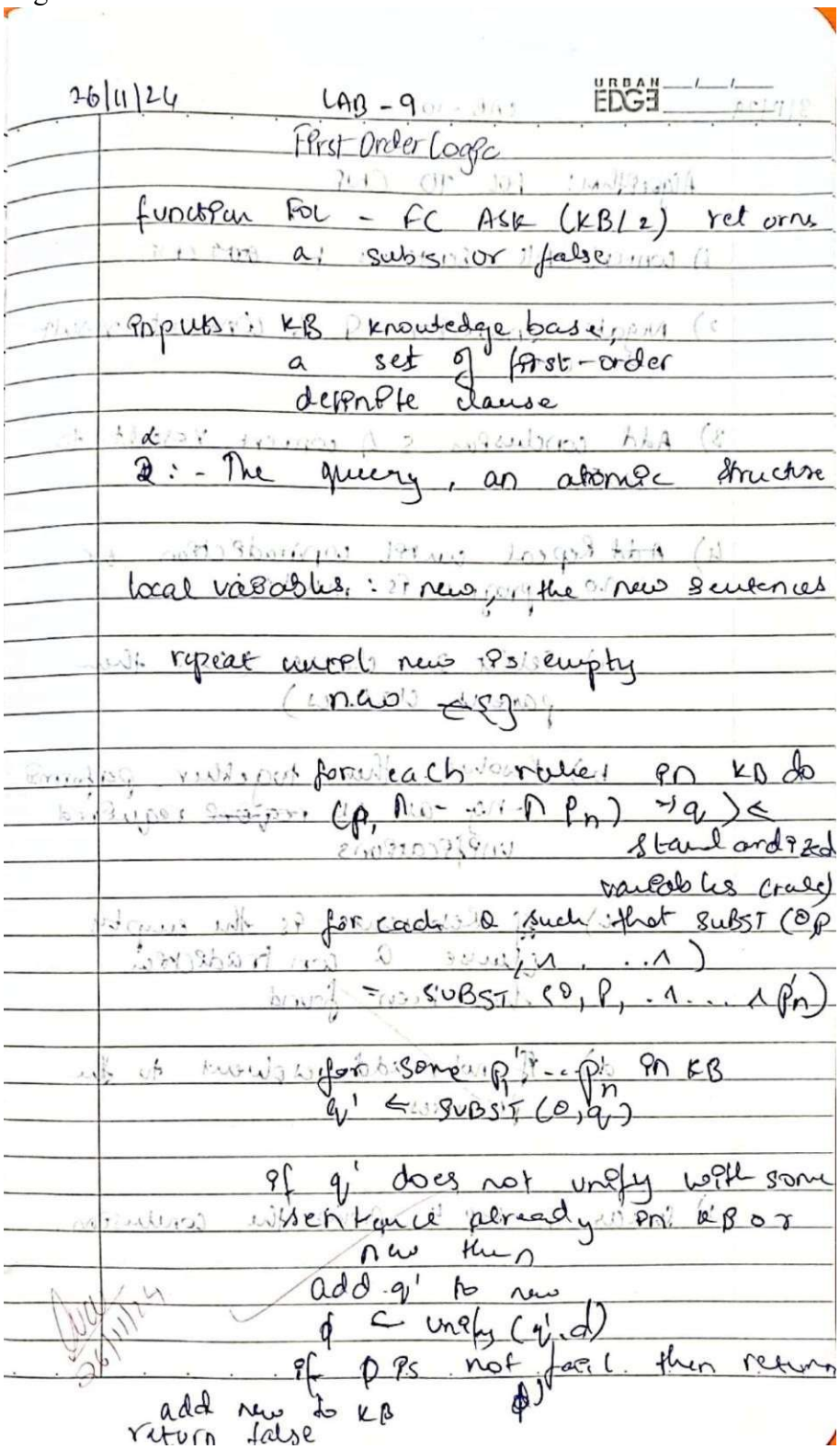
OUTPUT:

```
Attempting to unify: ('q', 'b', 'x', ('f', ('g', 'z'))) with ('q', 'z', ('f', 'y'), ('f', 'y'))
Current substitution: {}
Attempting to unify: b with z
Current substitution: {}
Unifying variable: b with value: z
Attempting to unify: x with ('f', 'y')
Current substitution: {'b': 'z'}
Unifying variable: x with value: ('f', 'y')
Attempting to unify: ('f', ('g', 'z')) with ('f', 'y')
Current substitution: {'b': 'z', 'x': ('f', 'y')}
Attempting to unify: ('g', 'z') with y
Current substitution: {'b': 'z', 'x': ('f', 'y')}
Unifying variable: y with value: ('g', 'z')

Unification Successful!
Substitution: {'b': 'z', 'x': ('f', 'y'), 'y': ('g', 'z')}
```

Program 8

Implement FIRST ORDER LOGIC – 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. Check if the goal is reached: Criminal(Robert)
    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 the conclusion has been reached
    if 'Criminal(Robert)' in KB:
```

```
        print("Conclusion: Robert is a criminal!")
    else:
        print("No more inferences can be made.")

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

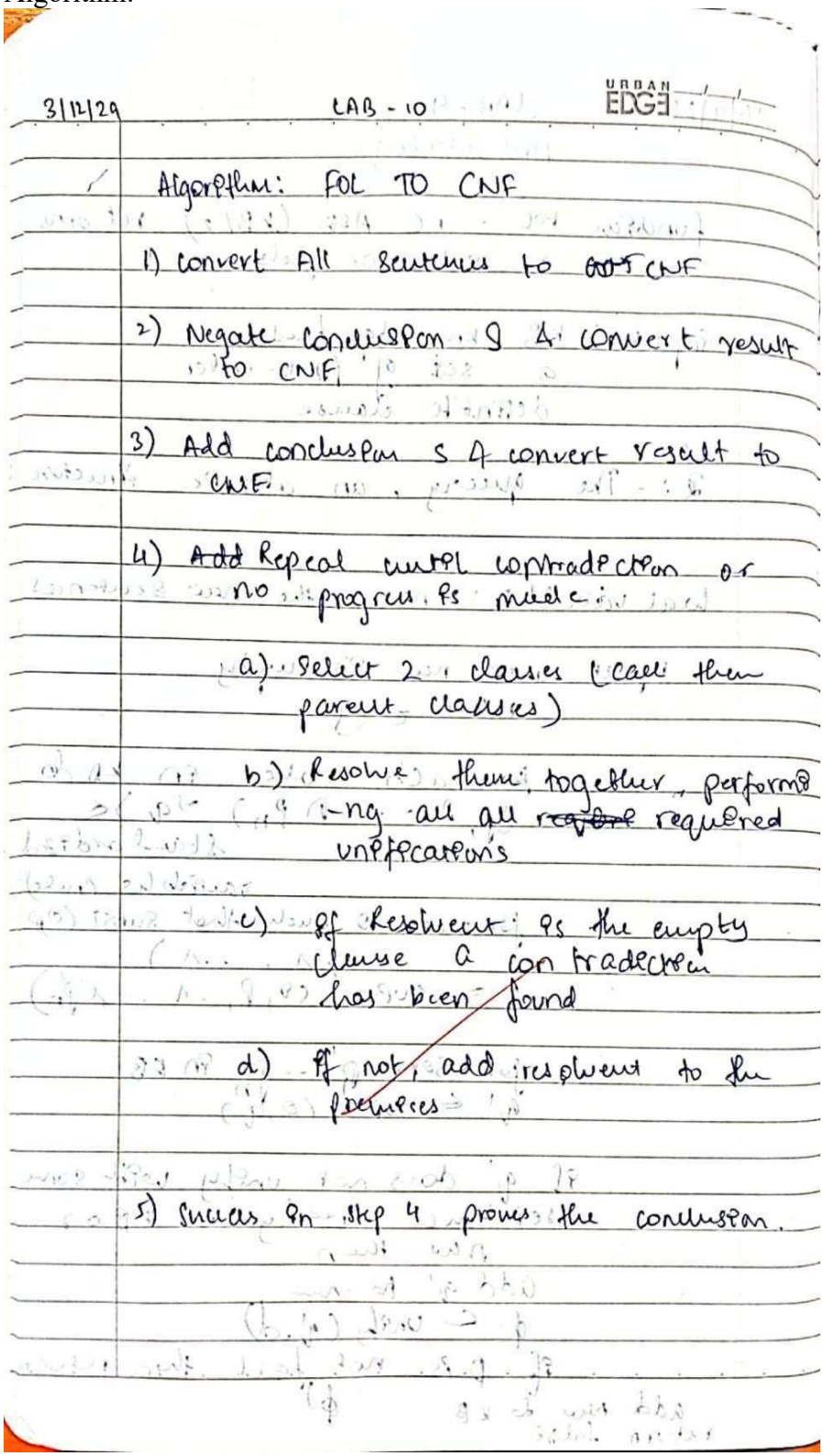
OUTPUT:

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

Program 9

Convert a given first order logic statement into Conjunctive Normal Form (CNF).

Algorithm:



3/12/24 LAB - 10

Algorithm: FOL TO CNF

1) Convert All Sentences to CNF

2) Negate conclusion S & convert result to CNF

3) Add conclusion S & convert result to CNF.

4) Repeat until contradiction or no progress is made:

a) Select 2 clauses (call them parent clauses)

b) Resolve them together, performing all required unifications

c) If Resolvent is the empty clause a contradiction has been found

d) If not, add resolvent to the premises

5) Success on step 4 proves the conclusion.

URBAN EDGE

Lab 9 output:

Inferred : Weap on (T1)
Inferred : Sells (Robert, T1, A)
Inferred : Hostile (A)
Inferred : Criminal (Robert)
Robert is a criminal!

Lab 10 output:

Does John like Peanuts? (Yes).

Code:

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

# Function to evaluate if a predicate is true based on the KB
def resolve(predicate):
    # If it's a direct fact in KB
    if predicate in KB and isinstance(KB[predicate], bool):
        return KB[predicate]

    # If it's a derived rule
    if predicate in KB:
        rule = KB[predicate]

        if " and " in rule: # Handle conjunction
            sub_preds = rule.split(" and ")
            return all(resolve(sub.strip()) for sub in sub_preds)
        elif " or " in rule: # Handle disjunction
            sub_preds = rule.split(" or ")
            return any(resolve(sub.strip()) for sub in sub_preds)
        elif "not " in rule: # Handle negation
            sub_pred = rule[4:]  # Remove "not "
            return not resolve(sub_pred.strip())
        else:  # Handle single predicate
            return resolve(rule.strip())

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

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

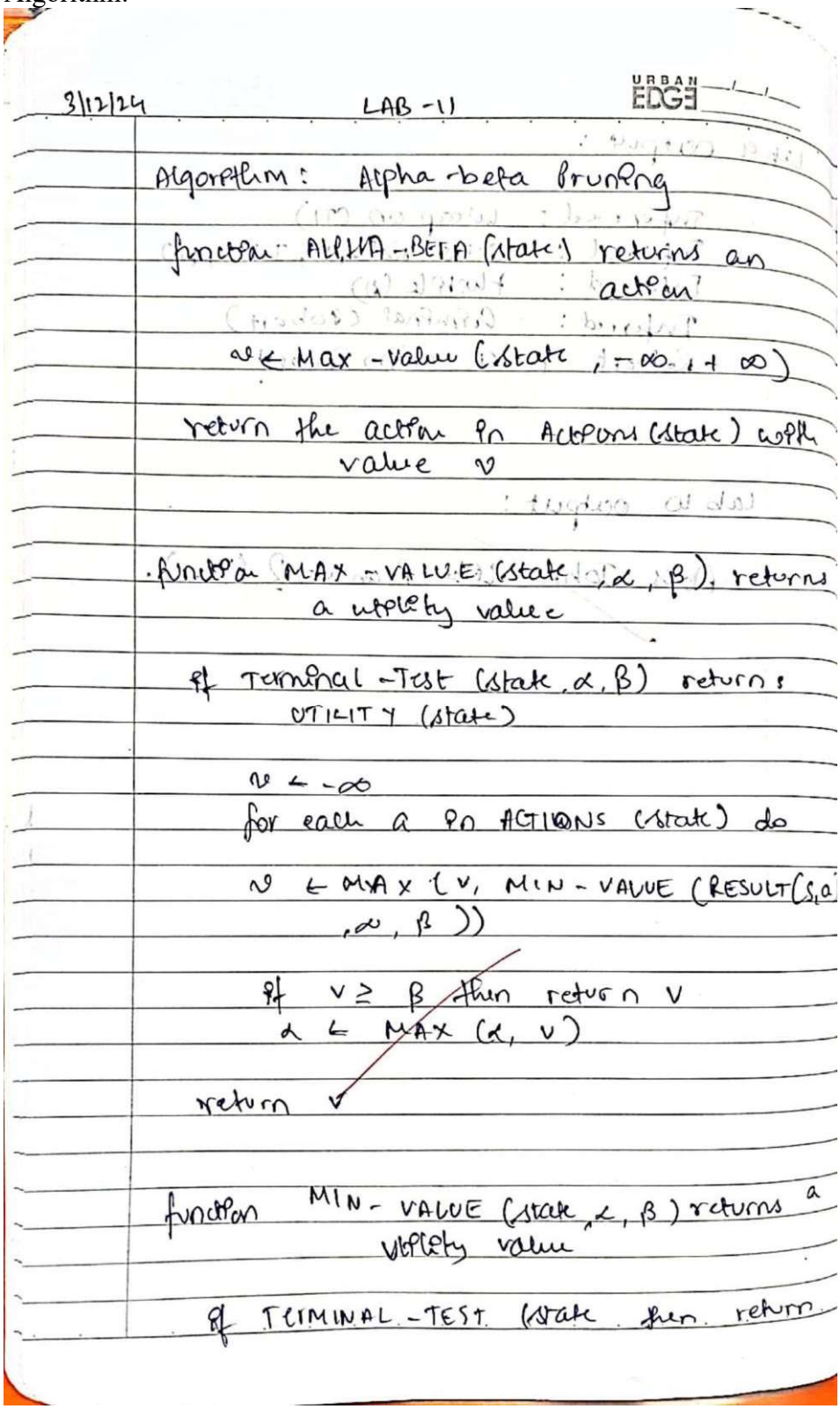
```
    return False

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

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

OUTPUT:
Does John like peanuts? Yes

Program 10

Implement Alpha-Beta Pruning.

Algorithm:



3/12/24 LAB -11

URBAN EDGE

Algorithm: Alpha-beta pruning

function ALPHA-BETA (state) returns an action

$v \leftarrow$ MAX-VALUE (state, $-\infty$, $+\infty$)

return the action in ACTIONS (state) with value v

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

if TERMINAL-TEST (state, α, β) return UTILITY (state)

$v \leftarrow -\infty$

for each a in ACTIONS (state) do

$v \leftarrow$ MAX (v, MIN-VALUE (RESULT(s,a), α, β))

if $v \geq \beta$ then return v

$\alpha \leftarrow$ MAX (α, v)

return v

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

if TERMINAL-TEST (state then return

URBAN EDGE

```
UTILITY (state)

v ∈ +∞

for each a in ACTIONS(state) do

    v ← MIN(v, MAX-VALUE(RESULTS,
        a), α, β))

    if v ≤ α then return v
        β ← MIN(β, v)

return v
```

Output :

~~Do~~

Enter numbers for the game Tree (space-seperated) : 10 9 14
18 5 4 50 3

Final Result of Alpha-beta Pruning : 50.

3/12/2024

Code:

```
# Alpha-Beta Pruning Implementation
def alpha_beta_pruning(node, alpha, beta, maximizing_player):
    # Base case: If it's a leaf node, return its value (simulating evaluation of the node)
    if type(node) is int:
        return node

    # If not a leaf node, explore the children
    if maximizing_player:
        max_eval = -float('inf')
        for child in node: # Iterate over children of the maximizer node
            eval = alpha_beta_pruning(child, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval) # Maximize alpha
            if beta <= alpha: # Prune the branch
                break
        return max_eval
    else:
        min_eval = float('inf')
        for child in node: # Iterate over children of the minimizer node
            eval = alpha_beta_pruning(child, alpha, beta, True)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)  # Minimize beta
            if beta <= alpha: # Prune the branch
                break
        return min_eval

# Function to build the tree from a list of numbers
def build_tree(numbers):
    # We need to build a tree with alternating levels of maximizers and minimizers
    # Start from the leaf nodes and work up
    current_level = [[n] for n in numbers]
    while len(current_level) > 1:
        next_level = []
        for i in range(0, len(current_level), 2):
            if i + 1 < len(current_level):
                next_level.append(current_level[i] + current_level[i + 1]) # Combine two nodes
            else:
                next_level.append(current_level[i]) # Odd number of elements, just carry forward
        current_level = next_level
    return current_level[0] # Return the root node, which is a maximizer

# Main function to run alpha-beta pruning
def main():
    # Input: User provides a list of numbers
    numbers = list(map(int, input("Enter numbers for the game tree (space-separated): ").split()))
```

```
    # Build the tree with the given numbers
    tree = build_tree(numbers)

    # Parameters: Tree, initial alpha, beta, and the root node is a maximizing player
    alpha = -float('inf')
    beta = float('inf')
    maximizing_player = True  # The root node is a maximizing player

    # Perform alpha-beta pruning and get the final result
    result = alpha_beta_pruning(tree, alpha, beta, maximizing_player)
    print("Final Result of Alpha-Beta Pruning:", result)

# Run the main function
if __name__ == "__main__":
    main()
```

OUTPUT:

```
Enter numbers for the game tree (space-separated): 3 5 6 9 1 2 0 -1
Final Result of Alpha-Beta Pruning: 6
```