VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Sanjana Shetty (1BM22CS238)

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 **Sanjana Shetty (1BM22CS238),** 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.

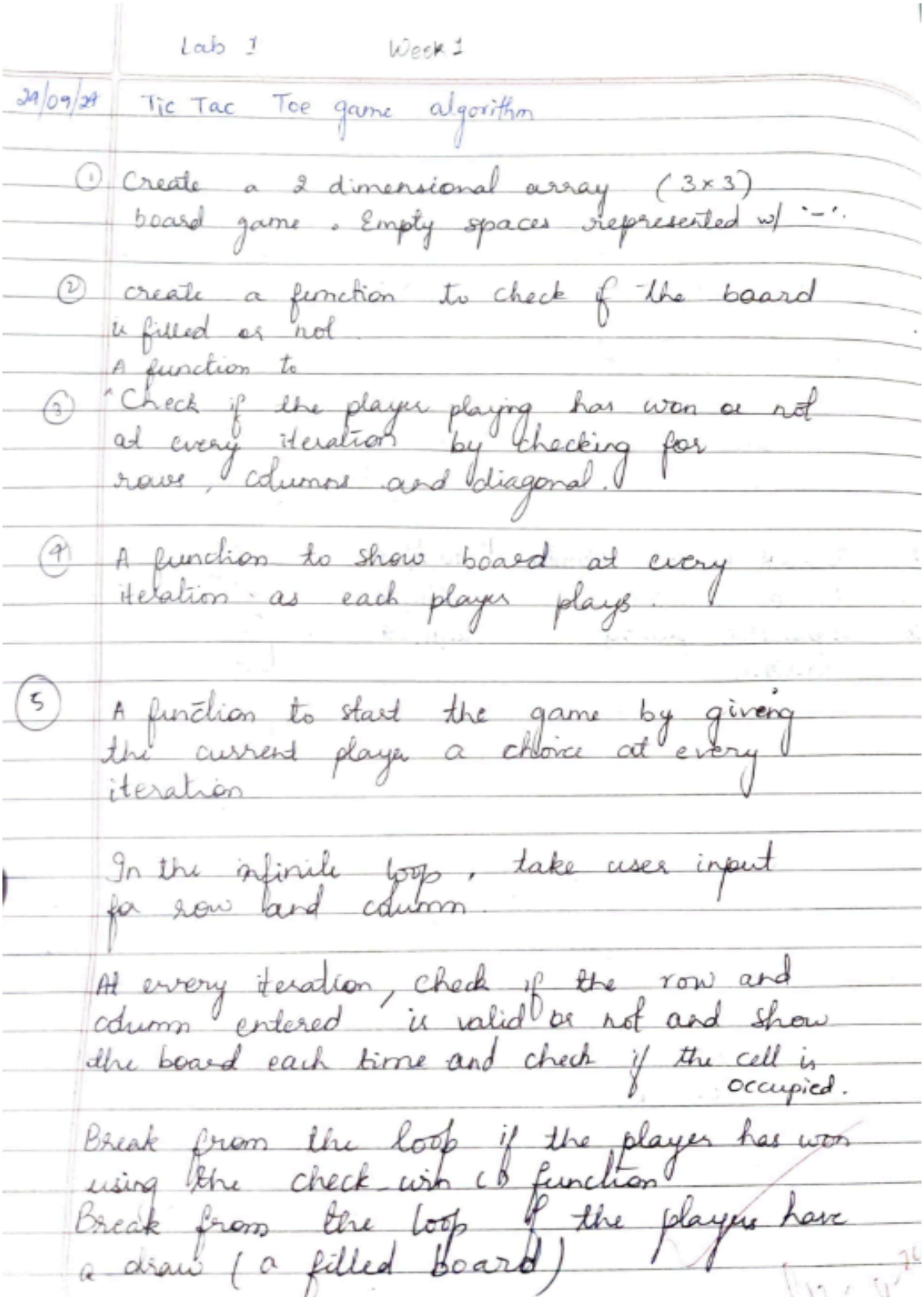
Dr. Seema Patil Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/sanjanashettybmsce/AI

Program 1:

Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent



Lab 1 Week 1

29/09/24 Tic Tac Toe game algorithm

① Create a 2 dimensional array (3×3) board game. Empty spaces represented w/ '-'.

② create a function to check if the board is filled or not.

A function to

③ Check if the player playing has won or not at every iteration by checking for rows, columns and diagonal.

④ A function to show board at every iteration as each player plays.

⑤ A function to start the game by giving the current player a choice at every iteration

In the infinite loop, take user input for row and column.

At every iteration, check if the row and column entered is valid or not and show the board each time and check if the cell is occupied.

Break from the loop if the player has won using the check_win() function

Break from the loop if the players have a draw (a filled board)

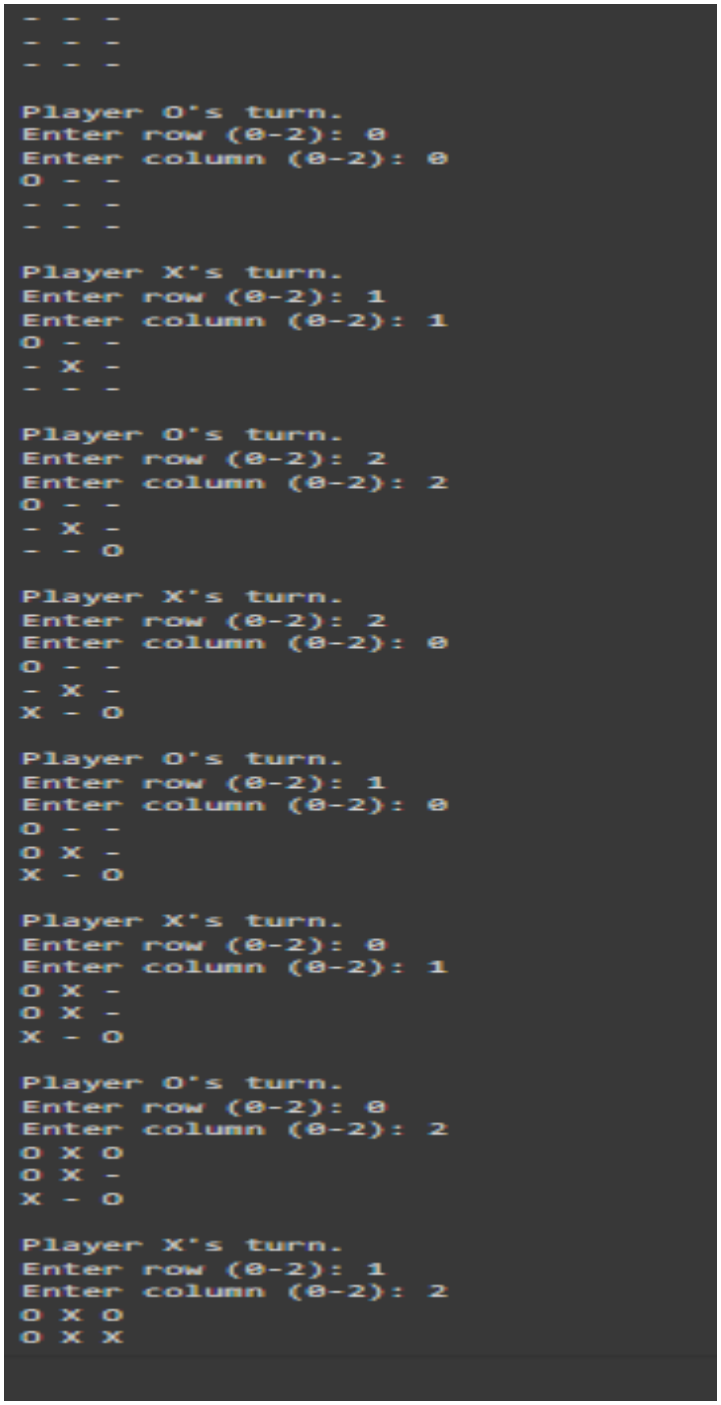
Code:

```
board = [['-' for _ in range(3)] for _ in range(3)]

def is_board_filled(board):
    for row in board:
        for element in row:
            if element == '-':
                return False
    return True

def check_win(board, player):
    # Check rows
    for row in board:
        if all(element == player for element in row):
            return True

    # Check columns
    for col in range(len(board[0])):
        if all(board[row][col] == player for row in range(len(board))):
            return True

    # Check diagonals
    if all(board[i][i] == player for i in range(len(board))):
        return True
    if all(board[i][len(board) - i - 1] == player for i in
range(len(board))):
        return True

    return False

def show_board(board):
    for row in board:
        print(" ".join(row))
    print()  # Add a blank line after the board

import random

def start_game(board):
```

```
    current_player = random.choice(['X', 'O'])
    while True:
        print(f"Player {current_player}'s turn.")

        try:
            row = int(input("Enter row (0-2): "))
            col = int(input("Enter column (0-2): "))
        except ValueError:
            print("Invalid input. Please enter integers between 0 and 2.")
            continue

        if 0 <= row <= 2 and 0 <= col <= 2:
            if board[row][col] == '-':
                board[row][col] = current_player

                show_board(board)

                if check_win(board, current_player):
                    print(f"Player {current_player} wins!")
                    break

                if is_board_filled(board):
                    print("It's a draw!")
                    break

                current_player = 'O' if current_player == 'X' else 'X'
            else:
                print("Invalid move. Cell already occupied. Try again.")
        else:
            print("Invalid input. Please enter numbers between 0 and 2.")

# Start the game
show_board(board)
start_game(board)
```

OUTPUT:

- Draw


- - -
- - -
- - -
Player O's turn.
Enter row (0-2): 0
Enter column (0-2): 0
O - -
- - -
- - -
Player X's turn.
Enter row (0-2): 1
Enter column (0-2): 1
O - -
- X -
- - -
Player O's turn.
Enter row (0-2): 2
Enter column (0-2): 2
O - -
- X -
- - O
Player X's turn.
Enter row (0-2): 2
Enter column (0-2): 0
O - -
- X -
X - O
Player O's turn.
Enter row (0-2): 1
Enter column (0-2): 0
O - -
O X -
X - O
Player X's turn.
Enter row (0-2): 0
Enter column (0-2): 1
O X -
O X -
X - O
Player O's turn.
Enter row (0-2): 0
Enter column (0-2): 2
O X O
O X -
X - O
Player X's turn.
Enter row (0-2): 1
Enter column (0-2): 2
O X O
O X X

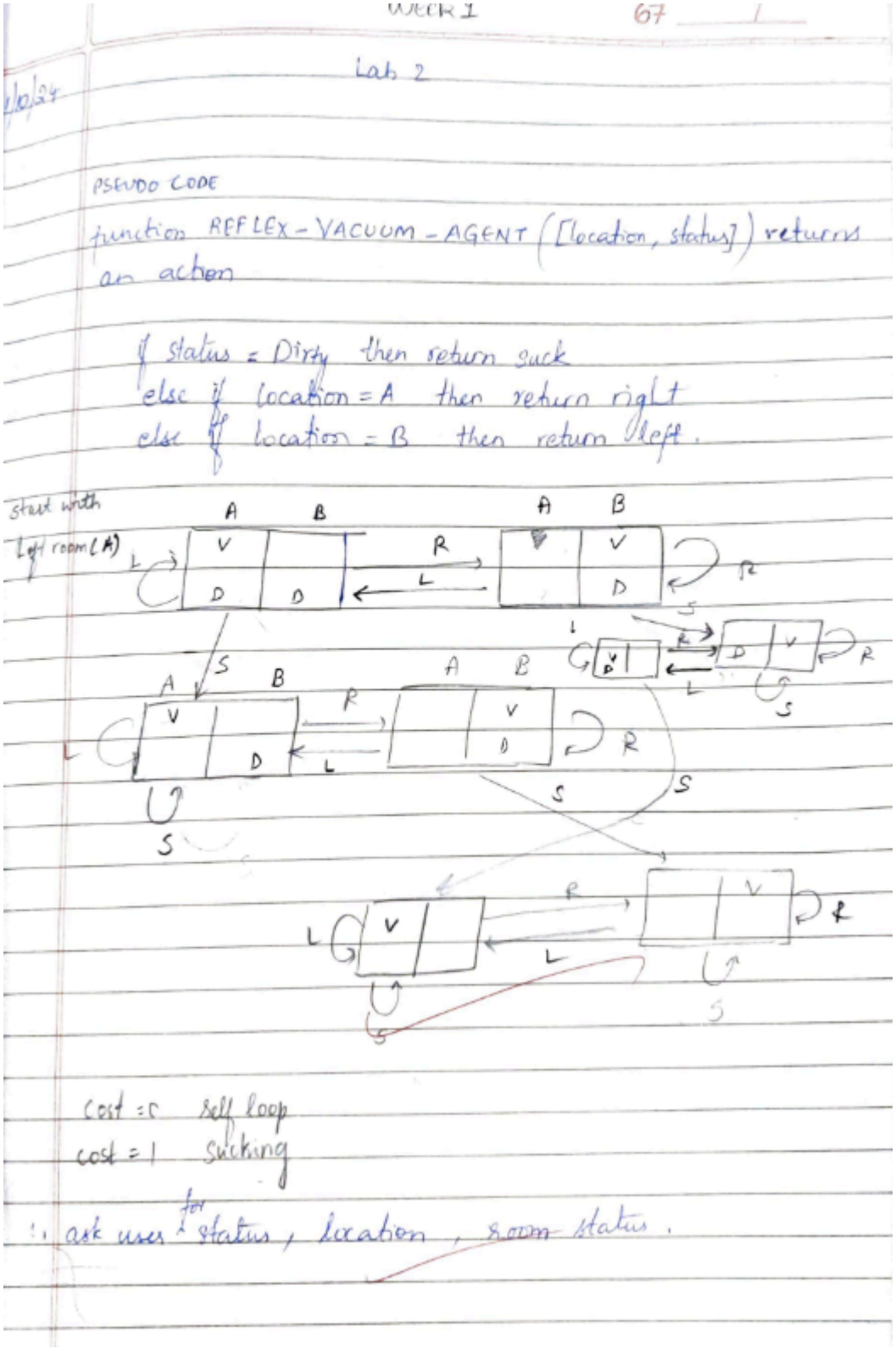
2. Win

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

Player X's turn.
Enter row (0-2): 0
Enter column (0-2): 0
X - -
- - -
- - -

Player O's turn.
Enter row (0-2): 1
Enter column (0-2): 0
X - -
O - -
- - -

Player X's turn.
Enter row (0-2): 1
Enter column (0-2): 1
X - -
O X -
- - -

Player O's turn.
Enter row (0-2): 2
Enter column (0-2): 1
X - -
O X -
- O -

Player X's turn.
Enter row (0-2): 2
Enter column (0-2): 2
X - -
O X -
- O X

Player X wins!
```

3. Occupying the same position

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

Player X's turn.
Enter row (0-2): 0
Enter column (0-2): 0
X - -
- - -
- - -

Player O's turn.
Enter row (0-2): 1
Enter column (0-2): 2
X - -
- - O
- - -

Player X's turn.
Enter row (0-2): 1
Enter column (0-2): 1
X - -
- X O
- - -

Player O's turn.
Enter row (0-2): 2
Enter column (0-2): 2
X - -
- X O
- - O

Player X's turn.
Enter row (0-2): 1
Enter column (0-2): 1
Invalid move. Cell already occupied. Try again.
Player X's turn.
Enter row (0-2):
```

Algorithm:


WEEK 1
67
Lab 2
PSEUDO CODE
function REFLEX-VACUUM-AGENT ([location, status]) returns an action
if status = Dirty then return suck
else if location = A then return right
else if location = B then return left.
start with
Left room (A)
A
B
V
D
R
L
S
Cost = 0 Self loop
cost = 1 Sucking
for
:. ask user status, location, room status.

Algorithm: Vacuum World problem for 2 quadrants

1. Initializing state:
   a) Set goal state of locations A, B as {'A': '0', 'B': '0'}
   b) initialize cost to 0

2. User input:
   a) prompt user to enter location of vacuum (A or B)
   b) ~~Initialize cost~~ prompt user to enter the status of selected location (A or B).
   c) prompt user to enter status of the other location (A or B)

3. Determining actions based on ~~input~~.
   a) If the vacuum is in Location A:
      i) Status of A == 1
         - print location A is dirty
         - set goal_state['A'] to '0' and increase cost by '1'

      ii) If status of B == 1
         - print location B is dirty
         - goal_state['B'] = 0 set
         - cost ++

      else
         - no action

iii) Status of A == 0
print location of A as clean

iv) Status of B == 1
print B as ~~clean~~ dirty
goal_state['B'] = 0
cost ++

b) If vacuum location is in B:
i) Status of B == 1
location B is dirty
goal_state['B'] = '0' and cost ++

ii) Status of A == 1
move to A (cost ++)
clean loc A

iii) Status B == 0
already clean

iv) Status of A == 1
move to A; cost ++
update goal_state

4. Output Result
: goal_state
print cost

10/12

Code:

```
#INSTRUCTIONS
#Enter LOCATION A/B in captial letters
#Enter Status O/1 accordingly where 0 means CLEAN and 1 means DIRTY

def vacuum_world_2q():

        # initializing goal_state
        # 0 indicates Clean and 1 indicates Dirty
    goal_state = {'A': '0', 'B': '0'}
    cost = 0

    location_input = input("Enter Location of Vacuum ") #user_input of
location vacuum is placed
    status_input = input("Enter status of " + location_input) #user_input if
location is dirty or clean
    other_location = 'B' if location_input == 'A' else 'A'
    status_input_complement = input("Enter status of " + other_location + "
(0 for CLEAN, 1 for DIRTY): ")

   # Initialize status dictionary
    initial_status = {location_input: status_input, other_location:
status_input_complement}
    print("Initial Status: " + str(initial_status))
    print("Initial Location Condition: " + str(goal_state))

    if location_input == 'A':
        # Location A is Dirty.
        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")
            # suck the dirt  and mark it as clean
            goal_state['A'] = '0'
            cost += 1                       #cost for suck
            print("Cost for CLEANING A " + str(cost))
            print("Location A has been Cleaned.")

            if status_input_complement == '1':
                # if B is Dirty
                print("Location B is Dirty.")
```

```
            print("Moving right to the Location B. ")
            cost += 1                        #cost for moving right
            print("COST for moving RIGHT" + str(cost))
            # suck the dirt and mark it as clean
            goal_state['B'] = '0'
            cost += 1                        #cost for suck
            print("COST for SUCK " + str(cost))
            print("Location B has been Cleaned. ")
        else:
            print("No action" + str(cost))
            # suck and mark clean
            print("Location B is already clean.")

    if status_input == '0':
        print("Location A is already clean ")
        if status_input_complement == '1':# if B is Dirty
            print("Location B is Dirty.")
            print("Moving RIGHT to the Location B. ")
            cost += 1                        #cost for moving right
            print("COST for moving RIGHT " + str(cost))
            # suck the dirt and mark it as clean
            goal_state['B'] = '0'
            cost += 1                        #cost for suck
            print("Cost for SUCK" + str(cost))
            print("Location B has been Cleaned. ")
        else:
            print("No action " + str(cost))
            print(cost)
            # suck and mark clean
            print("Location B is already clean.")

else:
    print("Vacuum is placed in location B")
    # Location B is Dirty.
    if status_input == '1':
        print("Location B is Dirty.")
        # suck the dirt  and mark it as clean
        goal_state['B'] = '0'
        cost += 1  # cost for suck
        print("COST for CLEANING " + str(cost))
        print("Location B has been Cleaned.")
```

```
            if status_input_complement == '1':
                # if A is Dirty
                print("Location A is Dirty.")
                print("Moving LEFT to the Location A. ")
                cost += 1  # cost for moving right
                print("COST for moving LEFT" + str(cost))
                # suck the dirt and mark it as clean
                goal_state['A'] = '0'
                cost += 1  # cost for suck
                print("COST for SUCK " + str(cost))
                print("Location A has been Cleaned.")

        else:
            print(cost)
            # suck and mark clean
            print("Location B is already clean.")

            if status_input_complement == '1':  # if A is Dirty
                print("Location A is Dirty.")
                print("Moving LEFT to the Location A. ")
                cost += 1  # cost for moving right
                print("COST for moving LEFT " + str(cost))
                # suck the dirt and mark it as clean
                goal_state['A'] = '0'
                cost += 1  # cost for suck
                print("Cost for SUCK " + str(cost))
                print("Location A has been Cleaned. ")
            else:
                print("No action " + str(cost))
                # suck and mark clean
                print("Location A is already clean.")

    # done cleaning
    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))


vacuum_world_2q()
```

OUTPUT:

1.

```
Enter Location of Vacuum a
Enter status of a1
Enter status of A (0 for CLEAN, 1 for DIRTY): 1
Initial Status: {'a': '1', 'A': '1'}
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in location B
Location B is Dirty.
COST for CLEANING 1
Location B has been Cleaned.
Location A is Dirty.
Moving LEFT to the Location A.
COST for moving LEFT2
COST for SUCK 3
Location A has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
```

2.

```
Enter Location of Vacuum B
Enter status of B0
Enter status of A (0 for CLEAN, 1 for DIRTY): 1
Initial Status: {'B': '0', 'A': '1'}
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in location B
0
Location B is already clean.
Location A is Dirty.
Moving LEFT to the Location A.
COST for moving LEFT 1
Cost for SUCK 2
Location A has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 2
```

3.

```
Enter Location of Vacuum B
Enter status of B0
Enter status of A (0 for CLEAN, 1 for DIRTY): 1
Initial Status: {'B': '0', 'A': '1'}
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in location B
0
Location B is already clean.
Location A is Dirty.
Moving LEFT to the Location A.
COST for moving LEFT 1
Cost for SUCK 2
Location A has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 2
```

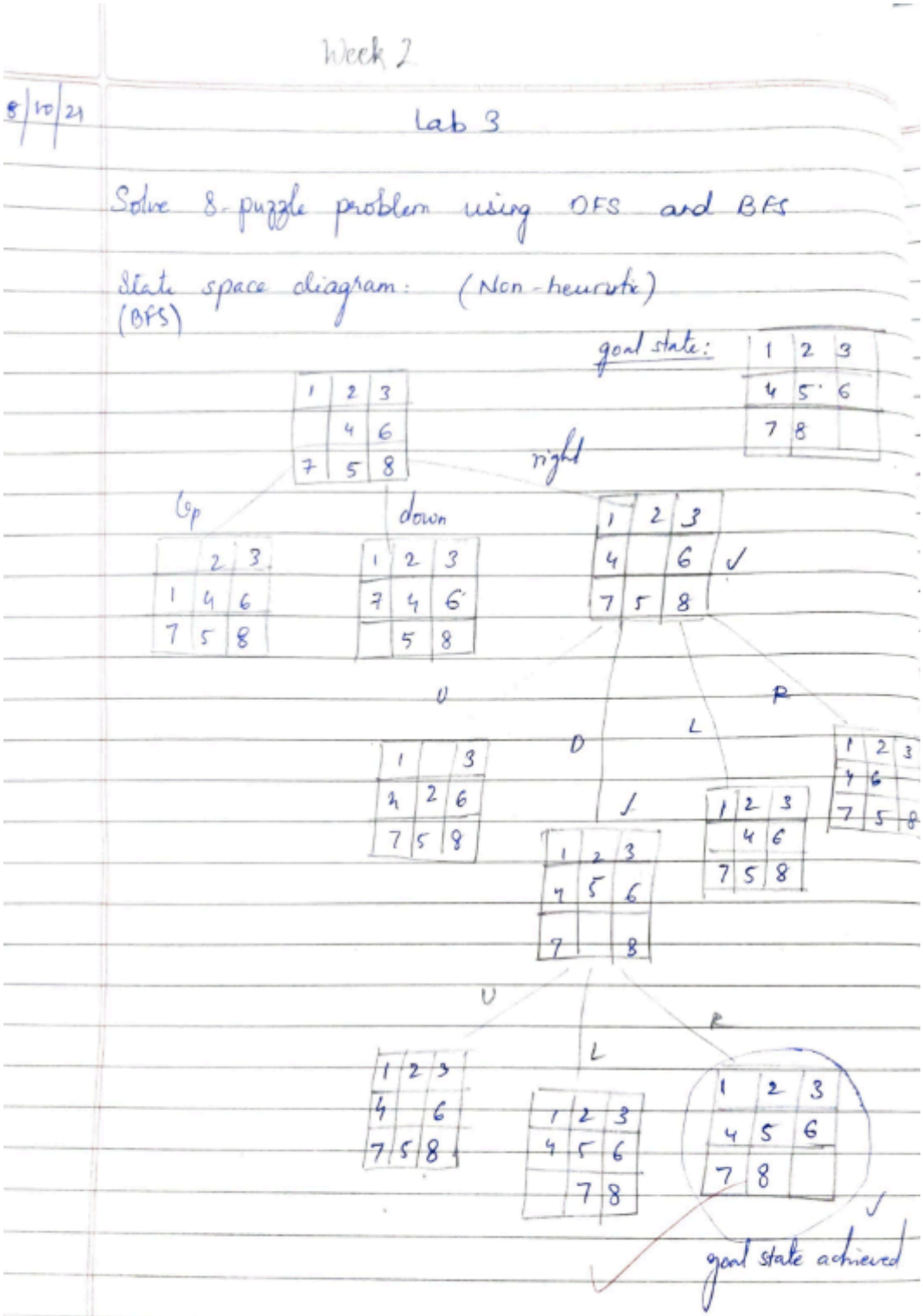
4.

```
Enter Location of Vacuum B
Enter status of B1
Enter status of A (0 for CLEAN, 1 for DIRTY): 0
Initial Status: {'B': '1', 'A': '0'}
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in location B
Location B is Dirty.
COST for CLEANING 1
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 1
```

Program 2:

Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm

Algorithm:



Code:

```
#DEPTH FIRST SEARCH
cnt = 0;
def print_state(in_array):
    global cnt
    cnt += 1
    for row in in_array:
        print(' '.join(str(num) for num in row))
    print()  # Print a blank line for better readability

def helper(goal, in_array, row, col, vis):
    # Mark the current position as visited
    vis[row][col] = 1
    drow = [-1, 0, 1, 0]  # Directions for row movements: up, right, down,
left
    dcol = [0, 1, 0, -1]  # Directions for column movements
    dchange = ['U', 'R', 'D', 'L']

    # Print the current state
    print("Current state:")
    print_state(in_array)

    # Check if the current state is the goal state
    if in_array == goal:
        print_state(in_array)
        print(f"Number of states : {cnt}")
        return True

    # Explore all possible directions
    for i in range(4):
        nrow = row + drow[i]
        ncol = col + dcol[i]

        # Check if the new position is within bounds and not visited
        if 0 <= nrow < len(in_array) and 0 <= ncol < len(in_array[0]) and not
vis[nrow][ncol]:
            # Make the move (swap the empty space with the adjacent tile)
            print(f"Took a {dchange[i]} move")
```

```
            in_array[row][col], in_array[nrow][ncol] = in_array[nrow][ncol],
in_array[row][col]

            # Recursive call
            if helper(goal, in_array, nrow, ncol, vis):
                return True

            # Backtrack (undo the move)
            in_array[row][col], in_array[nrow][ncol] = in_array[nrow][ncol],
in_array[row][col]

    # Mark the position as unvisited before returning
    vis[row][col] = 0
    return False

# Example usage
initial_state = [[1, 2, 3], [0, 4, 6], [7, 5, 8]]  # 0 represents the empty
space
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
visited = [[0] * 3 for _ in range(3)]  # 3x3 visited matrix
empty_row, empty_col = 1, 0  # Initial position of the empty space

found_solution = helper(goal_state, initial_state, empty_row, empty_col,
visited)
print("Solution found:", found_solution)
```

OUTPUT:

```
Current state:
1 2 3
0 4 6
7 5 8

Took a U move
Current state:
0 2 3
1 4 6
7 5 8

Took a R move
Current state:
2 0 3
1 4 6
7 5 8

Took a R move
Current state:
2 3 0
1 4 6
7 5 8

Took a D move
Current state:
2 3 6
1 4 0
7 5 8

Took a D move
Current state:
2 3 6
1 4 8
7 5 0

Took a L move
Current state:
2 3 6
1 4 8
7 0 5

Took a U move
Current state:
2 3 6
1 0 8
7 4 5

Took a L move
Current state:
2 3 6
1 4 8
0 7 5

Took a L move
Current state:
2 3 6
1 0 4
7 5 8
```

Algorithm:

Week 2

Lab 5 store 67

22/10/24 Implement Iterative ~~Deepening~~ search algorithm (DFS) Deepening

(8 puzzle)

(DFS in BFS manner)

1. for each child of the current node
2. if it is the target node, return
3. if the current maximum depth is reached, return
4. set the current node to this node & go back to 1
5. After having gone through all children, go to the next child of the parent (the next sibling)
6. After having gone through all children of the start node, increase the maximum depth and go back to 1
7. If we have reached all leaf (bottom) nodes, the goal node doesn't exist.

function ITERATIVE-DEEPENING-SEARCH (problem) returns a solution or failure

for depth=0 to ∞ do

result ← DEPTH-LIMITED-SEARCH (problem, depth)

if result ≠ cutoff then return result.

Code:

```
import copy

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

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

    def expand(self):
        children = []
        row, col = self.find_blank()
        possible_actions = []

        if row > 0:  # Can move the blank tile up
            possible_actions.append('Up')
        if row < 2:  # Can move the blank tile down
            possible_actions.append('Down')
        if col > 0:  # Can move the blank tile left
            possible_actions.append('Left')
        if col < 2:  # Can move the blank tile right
            possible_actions.append('Right')

        for action in possible_actions:
            new_state = copy.deepcopy(self.state)
            if action == 'Up':
                new_state[row][col], new_state[row - 1][col] = new_state[row
- 1][col], new_state[row][col]
            elif action == 'Down':
                new_state[row][col], new_state[row + 1][col] = new_state[row
+ 1][col], new_state[row][col]
            elif action == 'Left':
                new_state[row][col], new_state[row][col - 1] =
new_state[row][col - 1], new_state[row][col]
            elif action == 'Right':
```

```
                new_state[row][col], new_state[row][col + 1] =
new_state[row][col + 1], new_state[row][col]

                children.append(Node(new_state, self, action, self.depth + 1))
        return children

    def find_blank(self):
        for row in range(3):
            for col in range(3):
                if self.state[row][col] == 0:
                    return row, col
        raise ValueError("No blank tile found")

def depth_limited_search(node, goal_state, limit):
    if node.state == goal_state:
        return node
    if node.depth >= limit:
        return None
    for child in node.expand():
        result = depth_limited_search(child, goal_state, limit)
        if result is not None:
            return result
    return None

def iterative_deepening_search(initial_state, goal_state, max_depth):
    for depth in range(max_depth):
        result = depth_limited_search(Node(initial_state), goal_state, depth)
        if result is not None:
            return result
    return None

def print_solution(node):
    path = []
    while node is not None:
        path.append((node.action, node.state))
        node = node.parent
    path.reverse()

    for action, state in path:
        if action:
            print(f"Action: {action}")
```

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

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

max_depth = 20
solution = iterative_deepening_search(initial_state, goal_state, max_depth)

if solution:
    print("Solution found:")
    print_solution(solution)
else:
    print("Solution not found.")
```

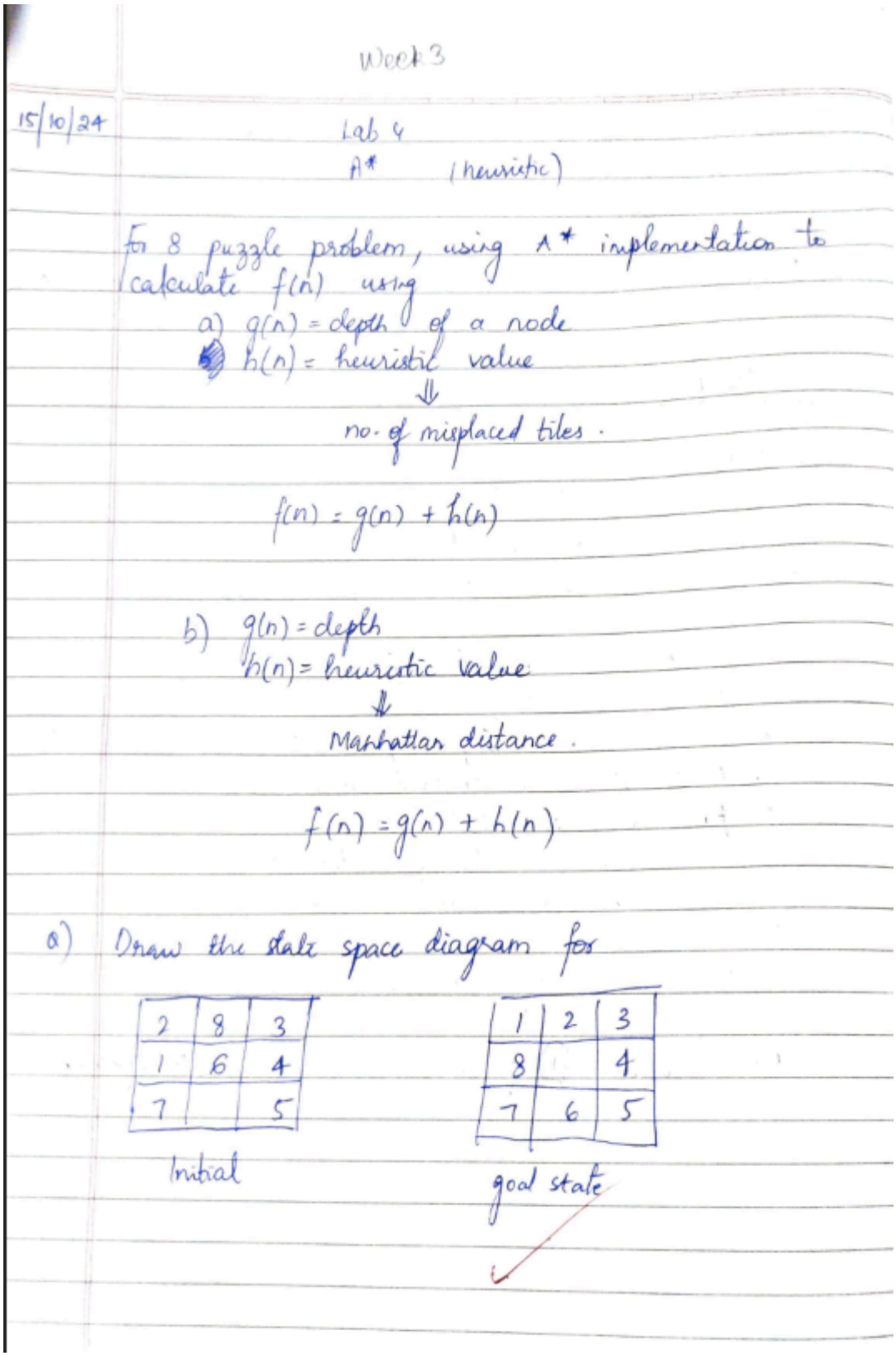
OUTPUT:

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

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

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

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

Program 3:

Implement A* search algorithm

Algorithm



Week 3

15/10/24

Lab 4

A* (heuristic)

To 8 puzzle problem, using A* implementation to calculate f(n) using

a) g(n) = depth of a node
h(n) = heuristic value
⇓
no. of misplaced tiles.

f(n) = g(n) + h(n)

b) g(n) = depth
h(n) = heuristic value
⇓
Manhattan distance.

f(n) = g(n) + h(n)

a) Draw the state space diagram for

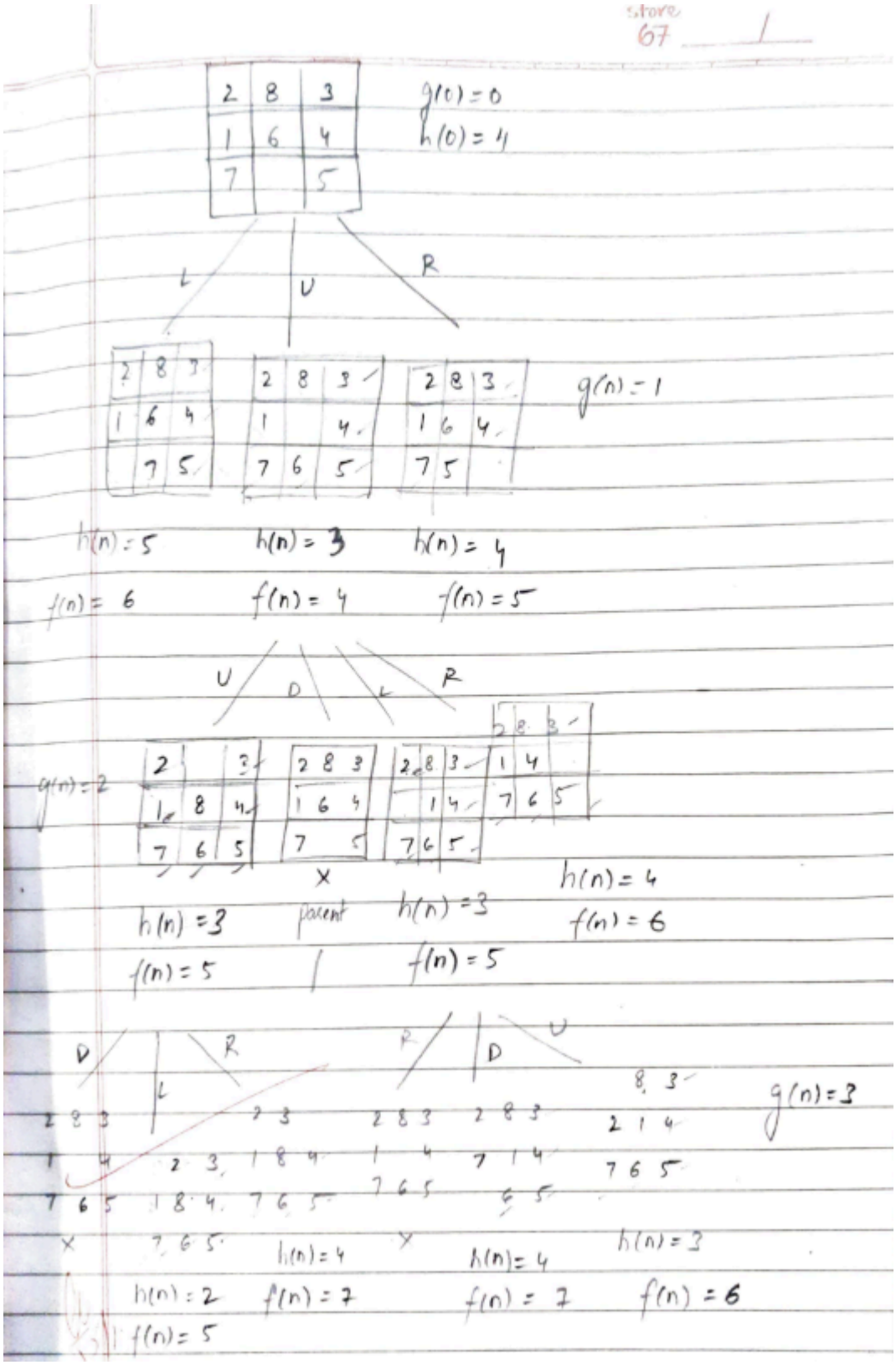
2	8	3
1	6	4
7		5

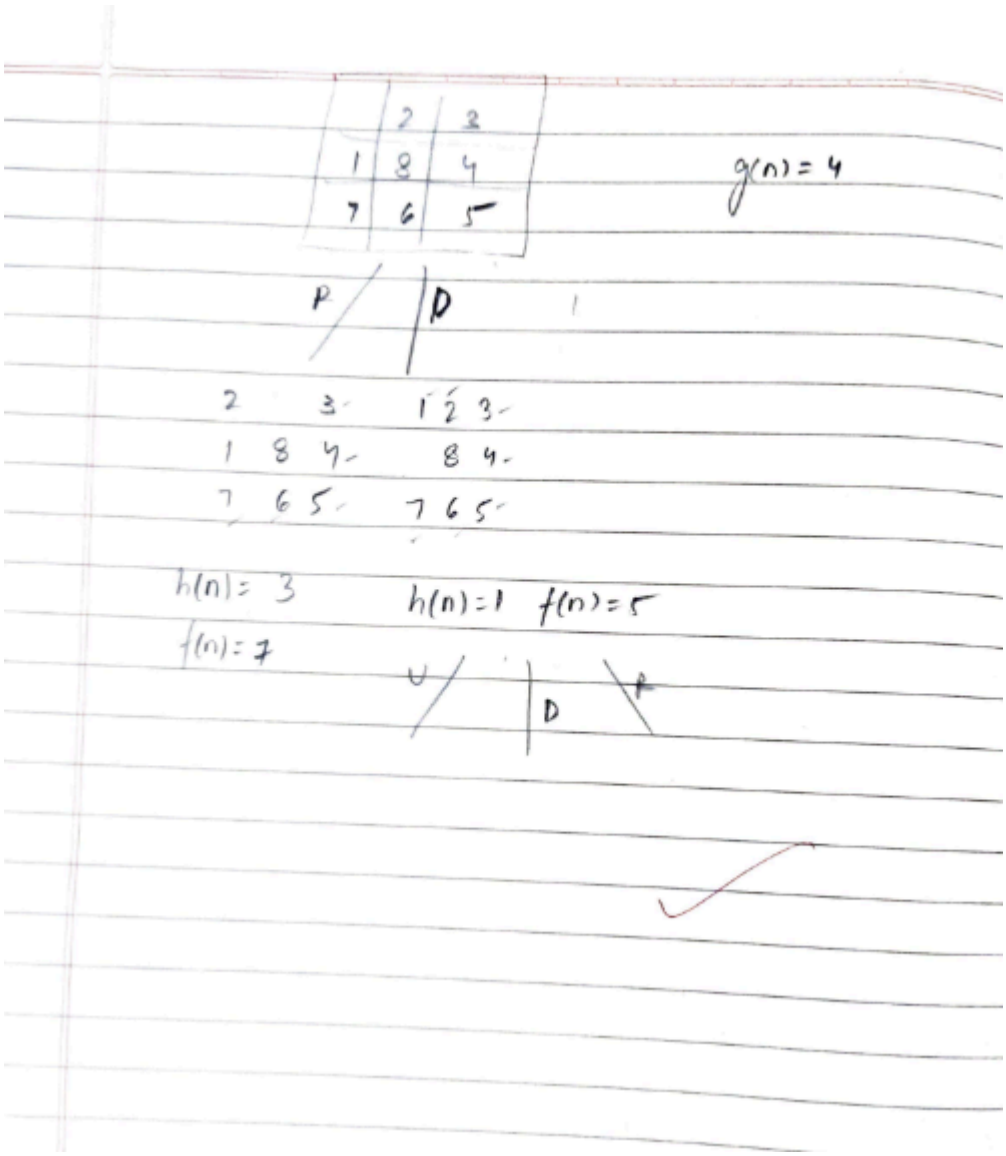
Initial

1	2	3
8		4
7	6	5

goal state

store
67 /


g(0) = 0
h(0) = 4
L
U
R
g(n) = 1
h(n) = 5
h(n) = 3
h(n) = 4
f(n) = 6
f(n) = 4
f(n) = 5
U
D
L
R
g(n) = 2
X
parent
h(n) = 3
f(n) = 5
h(n) = 3
f(n) = 5
h(n) = 4
f(n) = 6
D
R
L
R
D
U
g(n) = 3
X
h(n) = 4
f(n) = 7
h(n) = 4
f(n) = 7
h(n) = 3
f(n) = 6
h(n) = 2
f(n) = 5


2 3
1 8 4
7 6 5
g(n) = 4
R
D
2 3
1 8 4
7 6 5
1 2 3
8 4
7 6 5
h(n) = 3
f(n) = 7
h(n) = 1 f(n) = 5
U
D
R

Code:

```
import heapq

# Define the goal state for the 8 puzzle
GOAL_STATE = [1, 2, 3, 8, 0, 4, 7, 6, 5]  # 0 represents the blank space

# Find the index of the blank space (0)
def find_blank(state):
    return state.index(0)

# Define valid moves based on the blank's position
def possible_moves(blank_index):
    moves = []
    # Move blank up (swap with element above)
    if blank_index > 2:
        moves.append(blank_index - 3)
    # Move blank down (swap with element below)
    if blank_index < 6:
        moves.append(blank_index + 3)
    # Move blank left (swap with element to the left)
    if blank_index % 3 > 0:
        moves.append(blank_index - 1)
    # Move blank right (swap with element to the right)
    if blank_index % 3 < 2:
        moves.append(blank_index + 1)
    return moves

# Swap blank space (0) with the target position
def swap(state, blank_index, target_index):
    new_state = state[:]
    new_state[blank_index], new_state[target_index] =
new_state[target_index], new_state[blank_index]
    return new_state

# Heuristic function: Count the number of misplaced tiles
def heuristic(state):
    return sum([1 if state[i] != GOAL_STATE[i] and state[i] != 0 else 0 for i
in range(9)])

# A* Search Algorithm
```

```
def a_star(start_state):
    open_list = []  # priority queue to maintain nodes to be explored
    closed_list = set()  # to store already explored nodes

    # Initial state setup
    g = 0  # Cost to reach the current node (depth/level)
    h = heuristic(start_state)  # Heuristic (misplaced tiles)
    f = g + h  # Total cost (f = g + h)

    heapq.heappush(open_list, (f, g, h, start_state, []))  # Add initial node
to open list

    while open_list:
        # Get the node with the lowest f value
        f, g, h, current_state, path = heapq.heappop(open_list)

        # If current state is the goal, return the solution path
        if current_state == GOAL_STATE:
            return path + [(current_state, g, h)]

        # Add current state to the closed list
        closed_list.add(tuple(current_state))

        # Find blank's position and generate all possible moves
        blank_index = find_blank(current_state)
        for move in possible_moves(blank_index):
            new_state = swap(current_state, blank_index, move)

            # If the new state is already explored, skip it
            if tuple(new_state) in closed_list:
                continue

            # Calculate g, h, and f for the new state
            new_g = g + 1
            new_h = heuristic(new_state)
            new_f = new_g + new_h

            # Add the new state to the open list
            heapq.heappush(open_list, (new_f, new_g, new_h, new_state, path +
[(current_state, g, h)]))
```

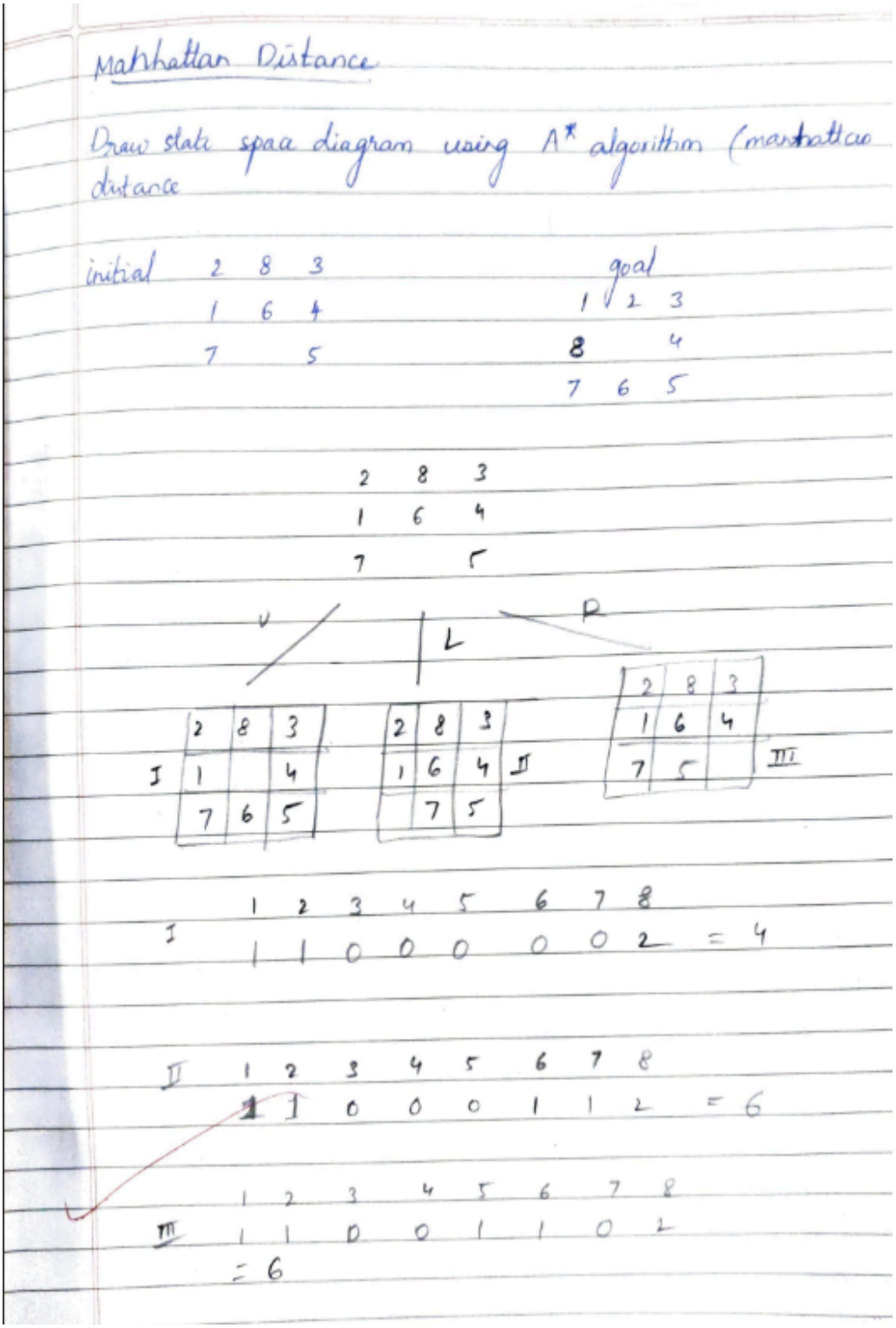
```
    return None  # If no solution is found

# Example usage
start_state = [2,8,3,1,6,4,7,0,5]  # Start state of the puzzle
solution = a_star(start_state)

if solution:
    print("Solution found:")
    for state_info in solution:
        state, g, h = state_info
        for i in range(0, 9, 3):
            print(state[i:i+3])
        print(f"Level (g): {g}, Heuristic (h): {h}, Total Cost (f = g + h):
{g + h}\n")
else:
    print("No solution exists.")
```

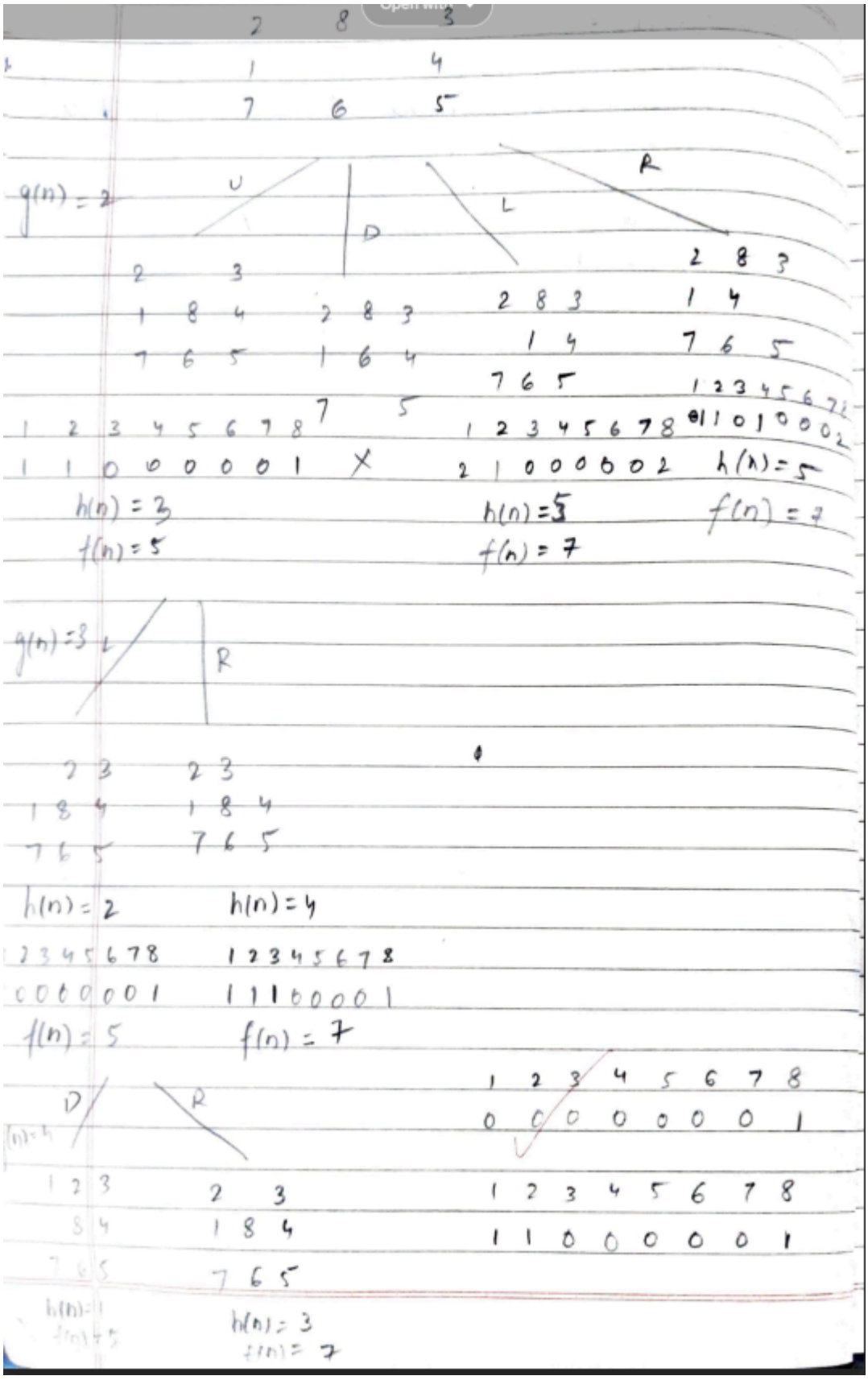
OUTPUT:

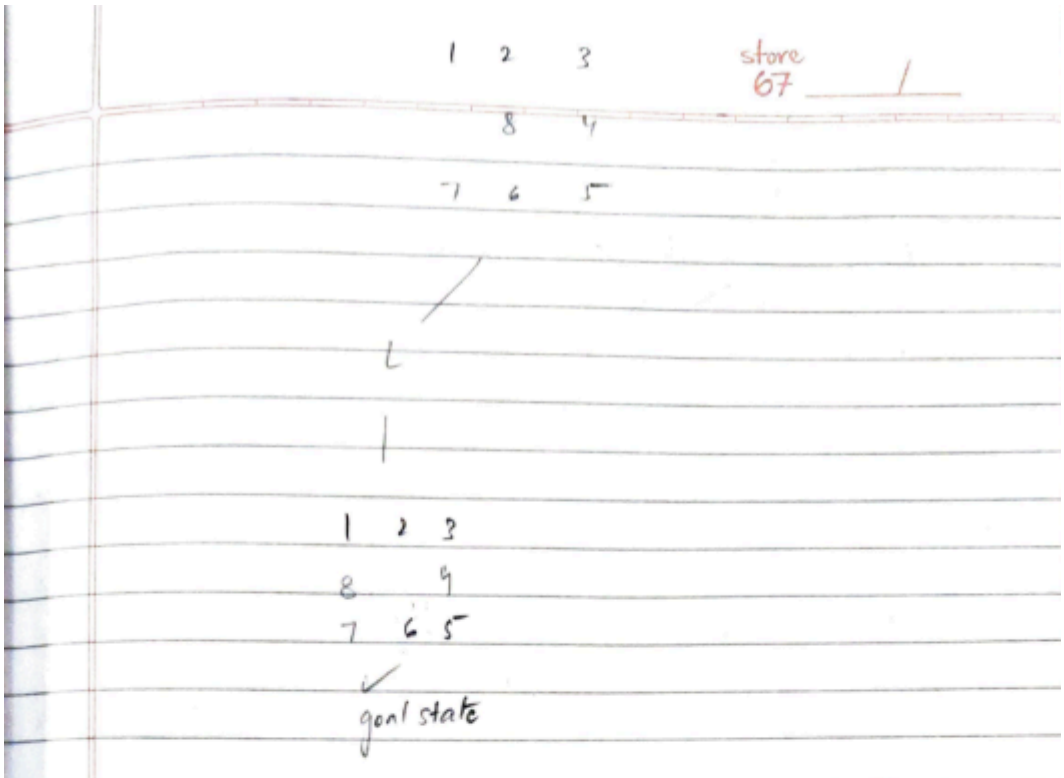
```
Solution found:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
Level (g): 0, Heuristic (h): 4, Total Cost (f = g + h): 4

[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
Level (g): 1, Heuristic (h): 3, Total Cost (f = g + h): 4

[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
Level (g): 2, Heuristic (h): 3, Total Cost (f = g + h): 5

[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
Level (g): 3, Heuristic (h): 2, Total Cost (f = g + h): 5

[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
Level (g): 4, Heuristic (h): 1, Total Cost (f = g + h): 5

[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
Level (g): 5, Heuristic (h): 0, Total Cost (f = g + h): 5
```

Manhattan Distance

Algorithm




2 8 3
1 4
7 6 5
g(n) = 2
U
D
L
R
2 3
1 8 4
7 6 5
2 8 3
1 6 4
7 5
1 2 3 4 5 6 7 8
1 1 0 0 0 0 0 1
h(n) = 3
f(n) = 5
2 8 3
1 4
7 6 5
1 2 3 4 5 6 7 8
2 1 0 0 0 0 0 2
h(n) = 5
f(n) = 7
2 8 3
1 4
7 6 5
1 2 3 4 5 6 7 8
0 1 1 0 1 0 0 0
h(n) = 5
f(n) = 7
g(n) = 3
L
R
2 3
1 8 4
7 6 5
h(n) = 2
1 2 3 4 5 6 7 8
0 0 0 0 0 0 0 1
f(n) = 5
2 3
1 8 4
7 6 5
h(n) = 4
1 2 3 4 5 6 7 8
1 1 1 0 0 0 0 1
f(n) = 7
D
R
1 2 3
8 4
7 6 5
2 3
1 8 4
7 6 5
h(n) = 3
f(n) = 7
1 2 3 4 5 6 7 8
0 0 0 0 0 0 0 1
1 2 3 4 5 6 7 8
1 1 0 0 0 0 0 1



store 67 1

Algorithm:

① Create an array with the numbers & space. (Initial node) (1D)

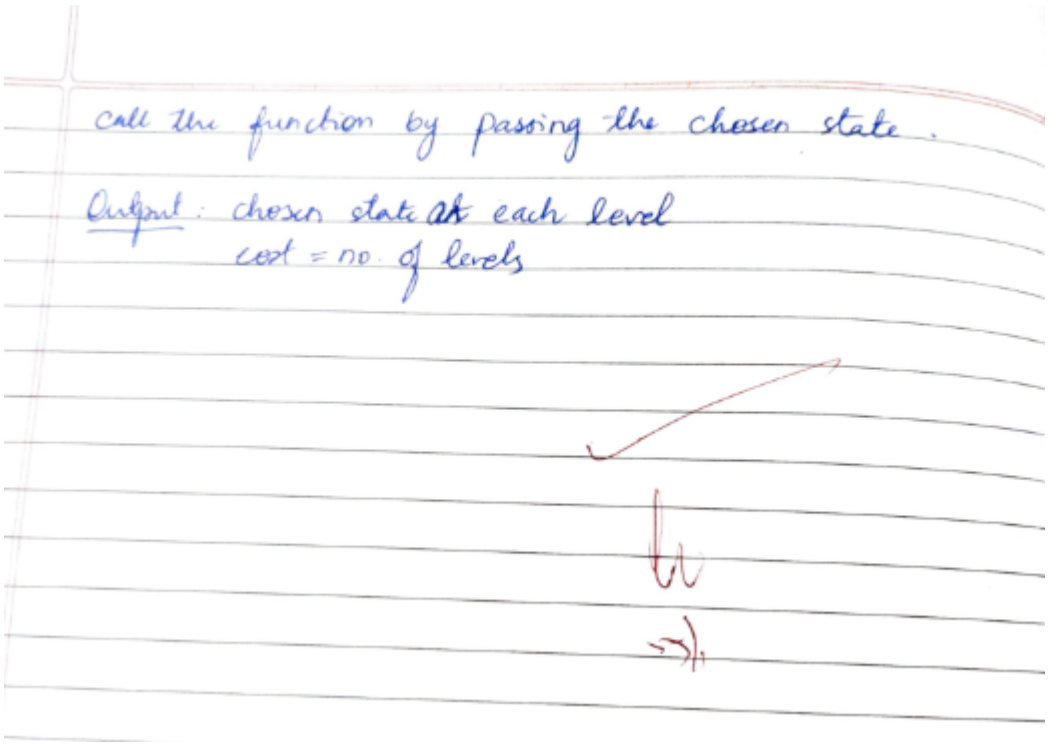
Create another array which is the goal state

② For each node (puzzle), all possible directions are checked for (U, D, L, R) & check if state is matching with goal ~~If it is~~ For creating the move, a swapping function can be used. b/w the number & space.

③ Each number in the current state is compared with goal state. Count is incremented when it is not matching.

④ Create minimum variable to ~~free~~ store the state with least count value.

⑤ Choose the node/state with the least value & recursively



Code:

```
#MANHATTAN DISTANCE

import heapq

# Define the goal state for the 8 puzzle
GOAL_STATE = [1, 2, 3, 4, 5, 6, 7, 8, 0]  # 0 represents the blank space

# Find the index of the blank space (0)
def find_blank(state):
    return state.index(0)

# Define valid moves based on the blank's position
def possible_moves(blank_index):
```

```
    moves = []
    # Move blank up (swap with element above)
    if blank_index > 2:
        moves.append(blank_index - 3)
    # Move blank down (swap with element below)
    if blank_index < 6:
        moves.append(blank_index + 3)
    # Move blank left (swap with element to the left)
    if blank_index % 3 > 0:
        moves.append(blank_index - 1)
    # Move blank right (swap with element to the right)
    if blank_index % 3 < 2:
        moves.append(blank_index + 1)
    return moves

# Swap blank space (0) with the target position
def swap(state, blank_index, target_index):
    new_state = state[:]
    new_state[blank_index], new_state[target_index] =
new_state[target_index], new_state[blank_index]
    return new_state

# Manhattan distance heuristic function
def manhattan_distance(state):
    distance = 0
    for i in range(9):
        if state[i] != 0:  # Ignore the blank space (0)
            # Find the target position in the goal state
            target_index = GOAL_STATE.index(state[i])
            # Calculate Manhattan distance
            current_row, current_col = divmod(i, 3)
            target_row, target_col = divmod(target_index, 3)
            distance += abs(current_row - target_row) + abs(current_col -
target_col)
    return distance

# A* Search Algorithm
def a_star(start_state):
    open_list = []  # priority queue to maintain nodes to be explored
    closed_list = set()  # to store already explored nodes
```

```
    # Initial state setup
    g = 0  # Cost to reach the current node (depth/level)
    h = manhattan_distance(start_state)  # Manhattan distance heuristic
    f = g + h  # Total cost (f = g + h)
 heapq.heappush(open_list, (f, g, h, start_state, []))  # Add initial node to
open list

    while open_list:
        # Get the node with the lowest f value
        f, g, h, current_state, path = heapq.heappop(open_list)

        # If current state is the goal, return the solution path
        if current_state == GOAL_STATE:
            return path + [(current_state, g, h)]

        # Add current state to the closed list
        closed_list.add(tuple(current_state))

        # Find blank's position and generate all possible moves
        blank_index = find_blank(current_state)
        for move in possible_moves(blank_index):
            new_state = swap(current_state, blank_index, move)

            # If the new state is already explored, skip it
            if tuple(new_state) in closed_list:
                continue

            # Calculate g, h, and f for the new state
            new_g = g + 1
            new_h = manhattan_distance(new_state)
            new_f = new_g + new_h

            # Add the new state to the open list
            heapq.heappush(open_list, (new_f, new_g, new_h, new_state, path +
[(current_state, g, h)]))

    return None  # If no solution is found

# Example usage
start_state = [1, 2, 3, 4, 0, 5, 6, 7, 8]  # Start state of the puzzle
solution = a_star(start_state)
```

```
if solution:
    print("Solution found:")
    for state_info in solution:
        state, g, h = state_info
        for i in range(0, 9, 3):
            print(state[i:i+3])
        print(f"Level (g): {g}, Heuristic (h): {h}, Total Cost (f = g + h):
{g + h}\n")
else:
    print("No solution exists.")
```

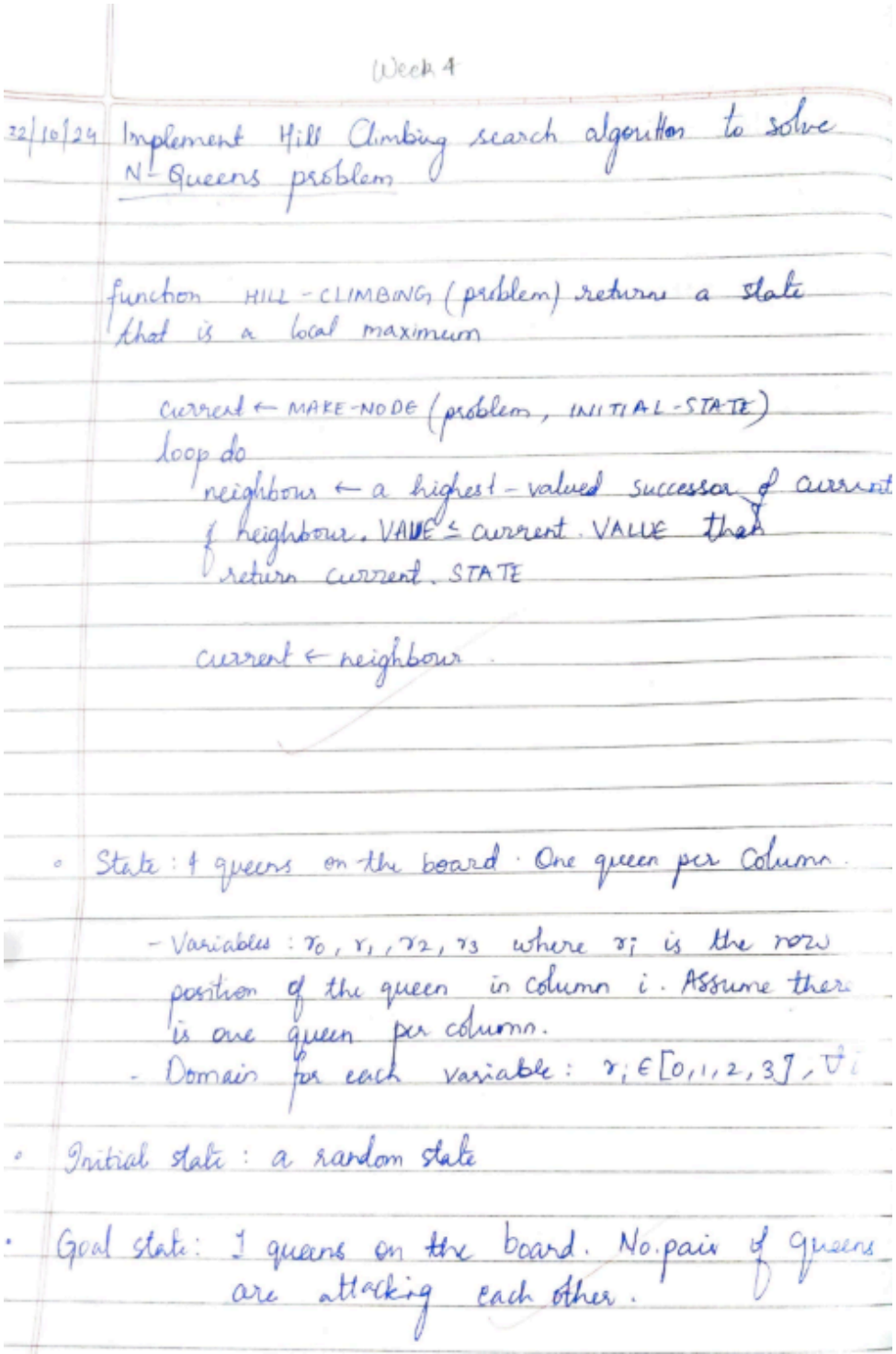
OUTPUT:

```
Solution found:
[1, 2, 3]
[4, 0, 5]
[6, 7, 8]
Level (g): 0, Heuristic (h): 6, Total Cost (f = g + h): 6

[1, 2, 3]
[4, 5, 0]
[6, 7, 8]
Level (g): 1, Heuristic (h): 5, Total Cost (f = g + h): 6

[1, 2, 3]
[4, 5, 8]
[6, 7, 0]
Level (g): 2, Heuristic (h): 6, Total Cost (f = g + h): 8

[1, 2, 3]
[4, 5, 8]
[6, 0, 7]
Level (g): 3, Heuristic (h): 7, Total Cost (f = g + h): 10

[1, 2, 3]
[4, 5, 8]
[0, 6, 7]
Level (g): 4, Heuristic (h): 6, Total Cost (f = g + h): 10

[1, 2, 3]
[0, 5, 8]
[4, 6, 7]
Level (g): 5, Heuristic (h): 7, Total Cost (f = g + h): 12

[1, 2, 3]
[5, 0, 8]
[4, 6, 7]
Level (g): 6, Heuristic (h): 8, Total Cost (f = g + h): 14

[1, 2, 3]
[5, 6, 8]
[4, 0, 7]
Level (g): 7, Heuristic (h): 7, Total Cost (f = g + h): 14

[1, 2, 3]
[5, 6, 8]
[4, 7, 0]
Level (g): 8, Heuristic (h): 6, Total Cost (f = g + h): 14

[1, 2, 3]
[5, 6, 0]
[4, 7, 8]
Level (g): 9, Heuristic (h): 5, Total Cost (f = g + h): 14

[1, 2, 3]
[5, 0, 6]
[4, 7, 8]
Level (g): 10, Heuristic (h): 4, Total Cost (f = g + h): 14

[1, 2, 3]
[0, 5, 6]
```

Program 4:

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



Week 4

22/10/24 Implement Hill Climbing search algorithm to solve N-Queens problem

function HILL-CLIMBING (problem) returns a state that is a local maximum

current ← MAKE-NODE (problem, INITIAL-STATE)
loop do
neighbour ← a highest-valued successor of current
if neighbour.VALUE ≤ current.VALUE then
return current.STATE

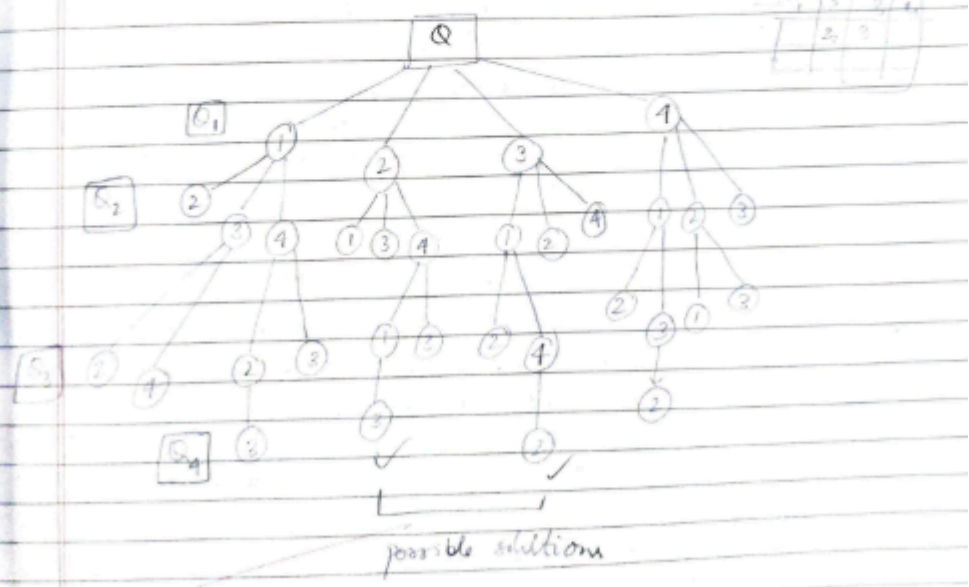
current ← neighbour

- State : 4 queens on the board. One queen per column.
  - Variables : x_0, x_1, x_2, x_3 where x_i is the row position of the queen in column i. Assume there is one queen per column.
  - Domain for each variable : $x_i \in [0,1,2,3], \forall i$
- Initial state : a random state
- Goal state: 4 queens on the board. No pair of queens are attacking each other.

- Neighbour relation :
  Swap the row position of 2 queens

- Cost function : The number of pairs of queens attacking each other, direct or indirectly.

State space diagram (HILL CLIMBING - N Queens)



Code:

```
#HILL_CLIMBING

import random

def calculate_cost(state):
    """Calculate the number of conflicts in the current state."""
    cost = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i -
j):
                cost += 1
    return cost

def get_neighbors(state):
    """Generate all possible neighbors by moving each queen in its column."""
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if state[col] != row:  # Move the queen in column `col` to a
different row
                new_state = list(state)
                new_state[col] = row
                neighbors.append(new_state)
    return neighbors

def hill_climbing(n, max_iterations=1000):
    """Perform hill climbing search to solve the N-Queens problem."""
    current_state = [random.randint(0, n - 1) for _ in range(n)]
    current_cost = calculate_cost(current_state)

    for iteration in range(max_iterations):
        if current_cost == 0:  # Found a solution
            return current_state

        neighbors = get_neighbors(current_state)
```

```
        neighbor_costs = [(neighbor, calculate_cost(neighbor)) for neighbor
in neighbors]
        next_state, next_cost = min(neighbor_costs, key=lambda x: x[1])

        if next_cost >= current_cost:  # No improvement found
            print(f"Local maximum reached at iteration {iteration}.
Restarting...")
            return None  # Restart with a new random state

        current_state, current_cost = next_state, next_cost
        print(f"Iteration {iteration}: Current state: {current_state}, Cost:
{current_cost}")

    print(f"Max iterations reached without finding a solution.")
    return None

# Get user-defined input for the number of queens
try:
    n = int(input("Enter the number of queens (N): "))
    if n <= 0:
        raise ValueError("N must be a positive integer.")
except ValueError as e:
    print(e)
    n = 4  # Default to 4 if input is invalid

solution = None

# Keep trying until a solution is found
while solution is None:
    solution = hill_climbing(n)

print(f"Solution found: {solution}")
```

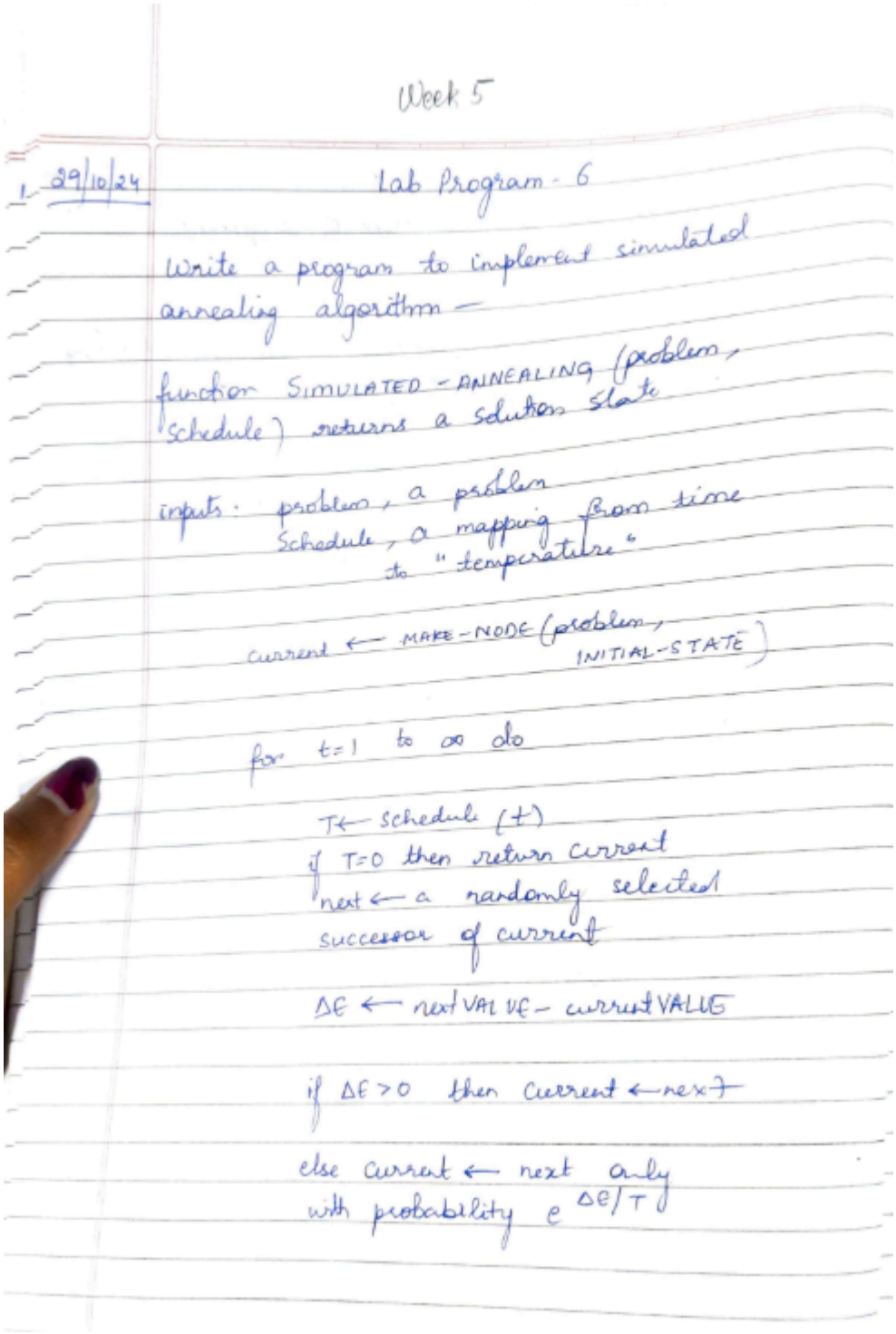
OUTPUT:

```
Enter the number of queens (N): 4
Iteration 0: Current state: [3, 1, 0, 2], Cost: 1
Local maximum reached at iteration 1. Restarting...
Local maximum reached at iteration 0. Restarting...
Iteration 0: Current state: [0, 3, 0, 1], Cost: 3
Iteration 1: Current state: [0, 3, 0, 2], Cost: 1
Iteration 2: Current state: [1, 3, 0, 2], Cost: 0
Solution found: [1, 3, 0, 2]
```

Program 5:

Stimulated Annealing to solve 8 queens problem.

Algorithm:



Week 5

1. 29/10/24 Lab Program - 6

Write a program to implement simulated annealing algorithm —

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

inputs: problem, a problem
schedule, a mapping from time to "temperature"

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 $e^{\Delta E/T}$

store
67 /

* 8 Queen problem using Simulated Annealing.

mlrose - python package

mlrose for simulated annealing

① Import the mlrose and numpy libraries.

$$P(x, x_j, T) = \begin{cases} 1 & f(x_j) \geq f(x) \\ e^{\frac{f(x_j) - f(x)}{T}} & f(x_j) < f(x) \end{cases}$$

OUTPUT

① initial position = [4, 6, 1, 5, 2, 0, 3, 7]
best position = [0, 6, 4, 7, 1, 3, 5, 2]

The number of queens that are not ~~attending~~ attacking each other is 8.0.

② The best position found is [2 5 7 1 3 0 6 4]
The number of queens that are not attacking each other is : 28.0

5 * Application of Simulated Annealing :

Job Scheduling Problems

job_times = [2, 14, 4, 16, 6, 5, 3, 12]
num_machines = 3

o/p Best job-to-machine assignment
= [1 2 0 1 2 0 1 0]

Minimum makespan : 21.0

Machine 1 jobs : [(2,4), (5,5), (7,12)]
Total time : 21

Machine 2 jobs : [(0,2), (3,16), (6,3)]
Total time : 21

Machine 3 jobs : [(1,14), (4,6)]
Total time : 20

Code:

```
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[j] != position[i]) and (position[j] != position[i] +
(j - i)) and (position[j] != position[i] - (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])


result = mlrose.simulated_annealing(problem=problem, schedule=T,
max_attempts=500, max_iters=5000, init_state=initial_position)
best_position, best_objective = result[0], result[1]


print('The best position found is: ', best_position)
```

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

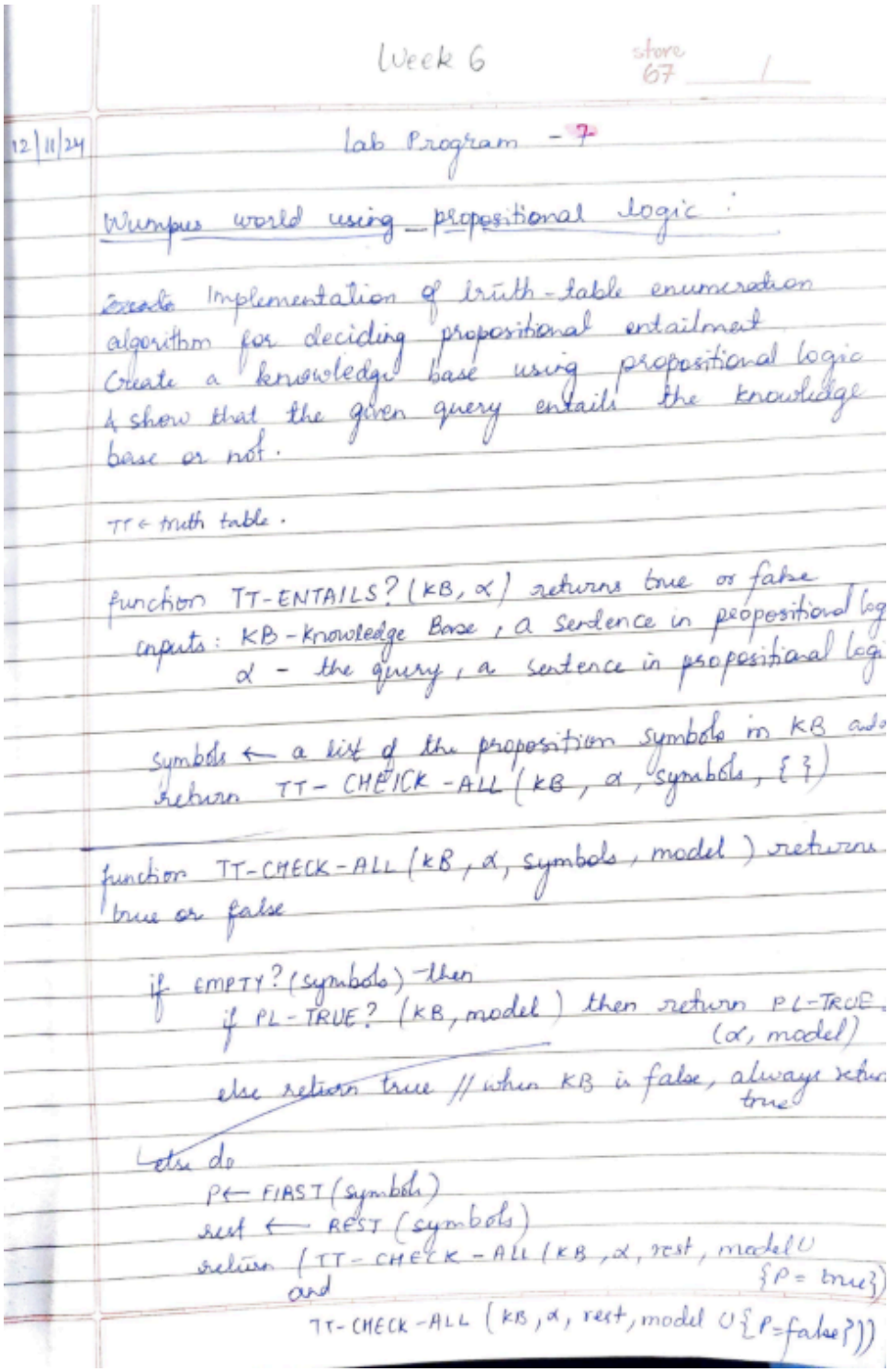
OUTPUT:

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

Program 6:

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

Algorithm:



Week 6 store 67

12/11/24 Lab Program - 7

Wumpus world using propositional logic :

Create Implementation of truth-table enumeration algorithm for deciding propositional entailment
Create a knowledge base using propositional logic & show that the given query entails the knowledge base or not.

TT ← truth table.

function TT-ENTAILS? (KB, α) returns true or false
inputs: KB - knowledge Base, a sentence in propositional log
α - the query, a sentence in propositional logi

symbols ← a list of the proposition symbols in KB and
return TT-CHECK-ALL (KB, α, symbols, {})

function TT-CHECK-ALL (KB, α, symbols, model) returns true or false

if EMPTY? (symbols) then
if PL-TRUE? (KB, model) then return PL-TRUE (α, model)
else return true // when KB is false, always 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}))

Example:

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

Checking that $KB \models \alpha$

A	B	C	$A \vee C$	$A \vee \neg C$	KB	α
false	F	F	F	T	F	F
false	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	F	T
T	T	F	T	T	T	T
T	T	T	T	T	T	T

Explanation

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Leftrightarrow Q$	$P \Rightarrow Q$
F	F	T	F	F	T	T
F	T	F	F	T	F	T
T	F	F	F	T	F	F
T	T	F	T	T	T	T

Code:

```
from itertools import product

def pl_true(sentence, model):
    """Evaluates if a sentence is true in a given model."""
    if isinstance(sentence, str):
        return model.get(sentence, False)
    elif isinstance(sentence, tuple) and len(sentence) == 2:  # NOT operation
        operator, operand = sentence
        if operator == "NOT":
            return not pl_true(operand, model)
    elif isinstance(sentence, tuple) and len(sentence) == 3:
        operator, left, right = sentence
        if operator == "AND":
            return pl_true(left, model) and pl_true(right, model)
        elif operator == "OR":
            return pl_true(left, model) or pl_true(right, model)
        elif operator == "IMPLIES":
            return not pl_true(left, model) or pl_true(right, model)
        elif operator == "IFF":
            return pl_true(left, model) == pl_true(right, model)

def print_truth_table(kb, query, symbols):
    """Generates and prints the truth table for KB and Query."""
    # Define headers with spaces for alignment
    headers = ["A      ", "B      ", "C      ", "A ∨ C  ", "B ∨ ¬C ", "KB
", "α     "]
    print(" | ".join(headers))
    print("-" * (len(headers) * 9))  # Separator line

    # Generate all combinations of truth values
    for values in product([False, True], repeat=len(symbols)):
        model = dict(zip(symbols, values))

        # Evaluate sub-expressions and main expressions
        a_or_c = pl_true(("OR", "A", "C"), model)
        b_or_not_c = pl_true(("OR", "B", ("NOT", "C")), model)
        kb_value = pl_true(("AND", ("OR", "A", "C"), ("OR", "B", ("NOT",
"C"))), model)
        alpha_value = pl_true(("OR", "A", "B"), model)
```

```
        # Print the truth table row
        row = values + (a_or_c, b_or_not_c, kb_value, alpha_value)
        row_str = " | ".join(str(v).ljust(7) for v in row)

        # Highlight rows where both KB and α are true
        if kb_value and alpha_value:
            print(f"\033[92m{row_str}\033[0m")  # Green color for rows where
KB and α are true
        else:
            print(row_str)

# Define the knowledge base and query
symbols = ["A", "B", "C"]
kb = ("AND", ("OR", "A", "C"), ("OR", "B", ("NOT", "C")))
query = ("OR", "A", "B")

# Print the truth table
print_truth_table(kb, query, symbols)
```

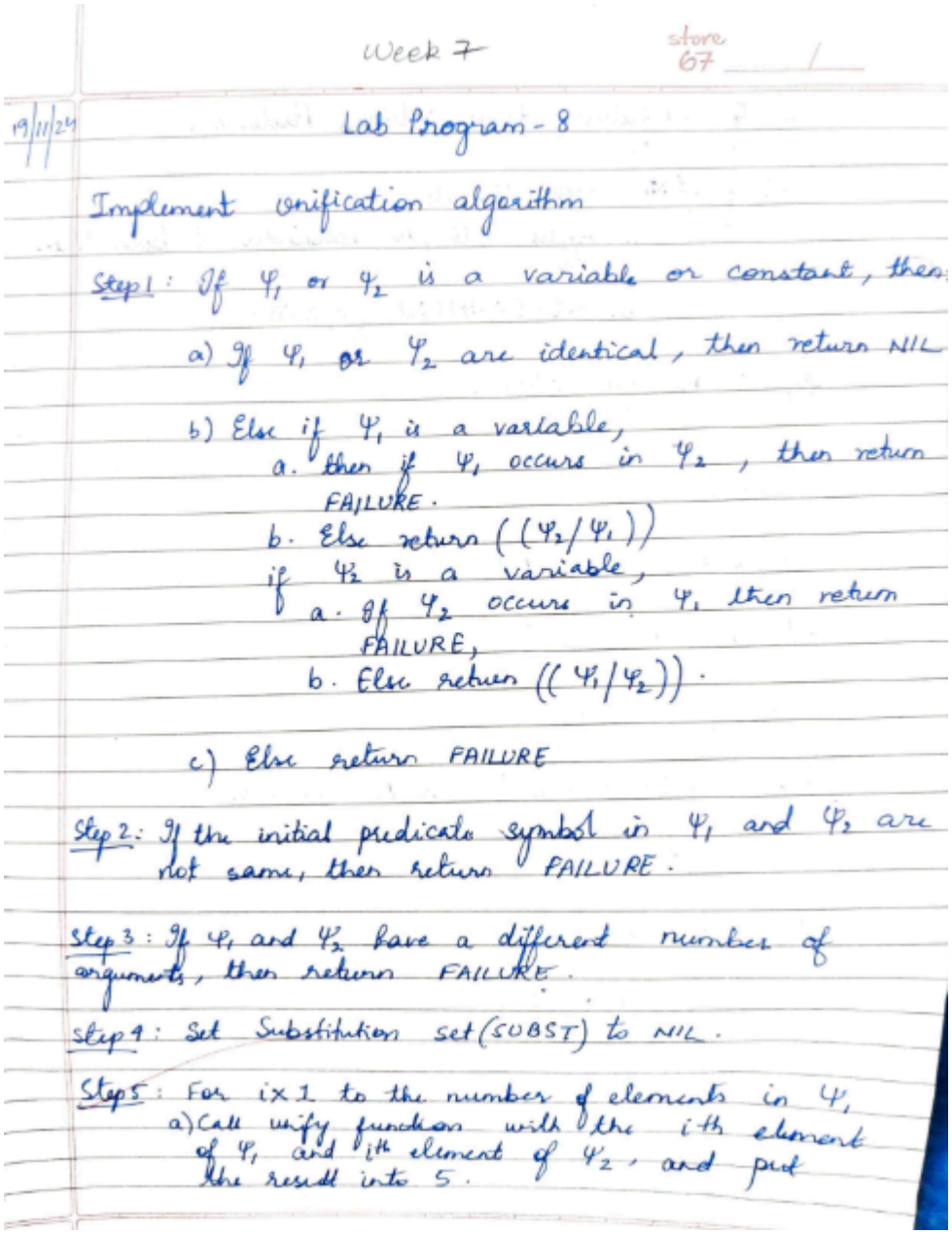
OUTPUT:

```
A       | B       | C       | A ∨ C   | B ∨ ¬C  | KB      | α
-----------------------------------------------------------------
False   | False   | False   | False   | True    | False   | False
False   | False   | True    | True    | False   | False   | False
False   | True    | False   | False   | True    | False   | True
False   | True    | True    | True    | True    | True    | True
True    | False   | False   | True    | True    | True    | True
True    | False   | True    | True    | False   | False   | True
True    | True    | False   | True    | True    | True    | True
True    | True    | True    | True    | True    | True    | True
```

Program 7:

Implement unification in first order logic

Algorithm:



Week 7 store 67

19/11/24 Lab Program - 8

Implement unification algorithm

Step 1: 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 FAILURE.
   b. Else return $((\Psi_2/\Psi_1))$
   if Ψ_2 is a variable,
   a. If Ψ_2 occurs in Ψ_1 then return FAILURE,
   b. Else return $((\Psi_1/\Psi_2))$.

c) Else return FAILURE

Step 2: If the initial predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE.

Step 3: If Ψ_1 and Ψ_2 have a different number of arguments, then return FAILURE.

Step 4: Set Substitution set (SUBST) to NIL.

Step 5: For i = 1 to the number of elements in Ψ_1,
a) Call unify function with the ith element of Ψ_1 and ith element of Ψ_2, and put the result into S.

b) If S = failure then returns Failure

c) If S = NIL then do,

a. Apply S to the remainder of both L1 and L2

b. SUBST = APPEND (S, SUBST).

Step 6: Return SUBST.

Examples:

(I) P (x, F(y)) ⟶ (1)

P (a, F(g(x)) ⟶ (2)

Same predicate P

(1) & (2) are identical if 'x' is replaced with 'a'

P(a, F(y))

If y is replaced w/ g(x)

P (a, F(g(x)))

store
67

(II) $Q(a, g(x,a), f(y))$ — ①
$Q(a, g(f(b), a), x)$ — ②

Replace x with $f(b)$ in ① ⇒ $Q(a, g(f(b), a), f(y))$

Replace $f(y)$ with x in ② ⇒ $Q(a, g(f(b), a), x)$

(Q1) $\Psi_1 = P(f(a), g(y))$
$\Psi_2 = P(x, x)$

In Ψ_1, f & g are distinct predicates, so they can't have the same values.

Unification fails.

(Q2) $\Psi_1 = P(b, x, f(g(z)))$ — ①
$\Psi_2 = P(z, f(y), f(y))$ — ②

Replace x with $f(y)$ in ① & $g(z)$ with y & b with z

$\Psi_1 = P(z, f(y), f(y))$

$\Psi_1 = \Psi_2$

Successful

Code:

```
def unify(expr1, expr2, subst=None):
    if subst is None:
        subst = {}

    # Apply substitutions to both expressions
    expr1 = apply_substitution(expr1, subst)
    expr2 = apply_substitution(expr2, subst)

    # Base case: Identical expressions
    if expr1 == expr2:
        return subst

    # If expr1 is a variable
    if is_variable(expr1):
        return unify_variable(expr1, expr2, subst)

    # If expr2 is a variable
    if is_variable(expr2):
        return unify_variable(expr2, expr1, subst)

    # If both are compound expressions (e.g., f(a), P(x, y))
    if is_compound(expr1) and is_compound(expr2):
        if expr1[0] != expr2[0] or len(expr1[1]) != len(expr2[1]):
            return None  # Predicate/function symbols or arity mismatch
        for arg1, arg2 in zip(expr1[1], expr2[1]):
            subst = unify(arg1, arg2, subst)
            if subst is None:
                return None
        return subst

    # If they don't unify
    return None
```

```
def unify_variable(var, expr, subst):
    """Handle variable unification."""
    if var in subst:  # Variable already substituted
        return unify(subst[var], expr, subst)
    if occurs_check(var, expr, subst):  # Occurs-check
        return None
    subst[var] = expr
    return subst

def apply_substitution(expr, subst):
    """Apply the current substitution set to an expression."""
    if is_variable(expr) and expr in subst:
        return apply_substitution(subst[expr], subst)
    if is_compound(expr):
        return (expr[0], [apply_substitution(arg, subst) for arg in expr[1]])
    return expr

def occurs_check(var, expr, subst):
    """Check for circular references."""
    if var == expr:
        return True
    if is_compound(expr):
        return any(occurs_check(var, arg, subst) for arg in expr[1])
    if is_variable(expr) and expr in subst:
        return occurs_check(var, subst[expr], subst)
    return False

def is_variable(expr):
    """Check if the expression is a variable."""
    return isinstance(expr, str) and expr.islower()


def is_compound(expr):
    """Check if the expression is a compound expression."""
    return isinstance(expr, tuple) and len(expr) == 2 and isinstance(expr[1],
list)

# Testing the algorithm with the given cases
if __name__ == "__main__":
    # Case 1: p(f(a), g(b)) and p(x, x)
```

```
expr1 = ("p", [("f", ["a"]), ("g", ["b"])])
expr2 = ("p", ["x", "x"])
result = unify(expr1, expr2)
print("Case 1 Result:", result)


# Case 2: p(b, x, f(g(z))) and p(z, f(y), f(y))
expr2 = ("p", ["a", ("f", [("g", ["x"])])])
expr1 = ("p", ["x", ("f", ["y"])])
result = unify(expr1, expr2)
print("Case 2 Result:", result)
```

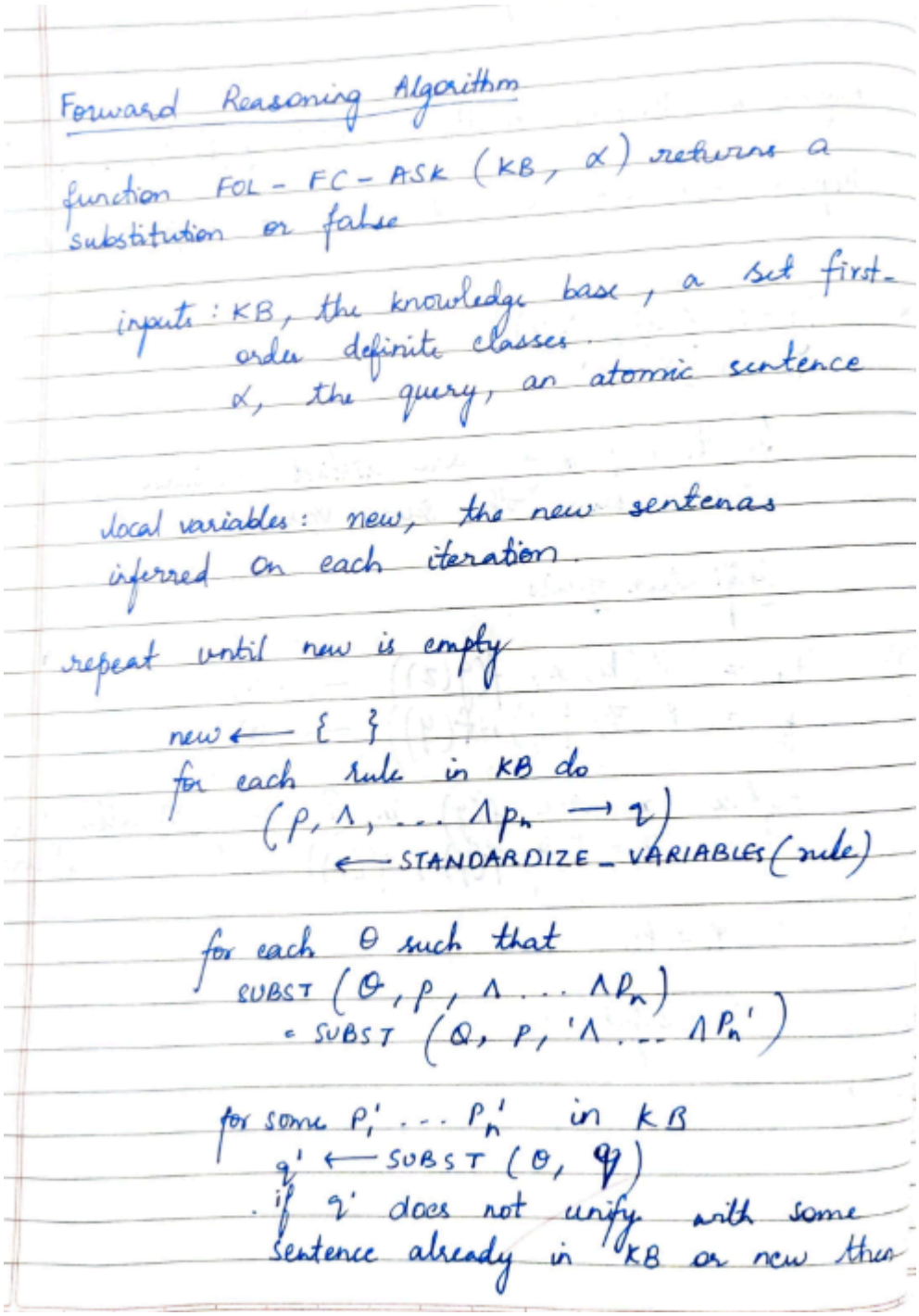
OUTPUT:

```
Case 1 Result: None
Case 2 Result: {'x': 'a', 'y': ('g', ['a'])}
```

Program 8:

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

Algorithm:



Forward Reasoning Algorithm

function FOL - FC - ASK (KB, α) returns a substitution or false

inputs : KB, the knowledge base, a set first-order definite clauses
α, the query, an atomic sentence

local variables : new, the new sentences inferred on each iteration

repeat until new is empty

new ← { }
for each rule in KB do
(p1 ∧ ... ∧ pn → q)
← STANDARDIZE_VARIABLES(rule)

for each θ such that
SUBST (θ, p1 ∧ ... ∧ pn)
= SUBST (θ, p1' ∧ ... ∧ pn')

for some p1' ... pn' in KB
q' ← SUBST (θ, q)
if q' does not unify with some sentence already in KB or new then

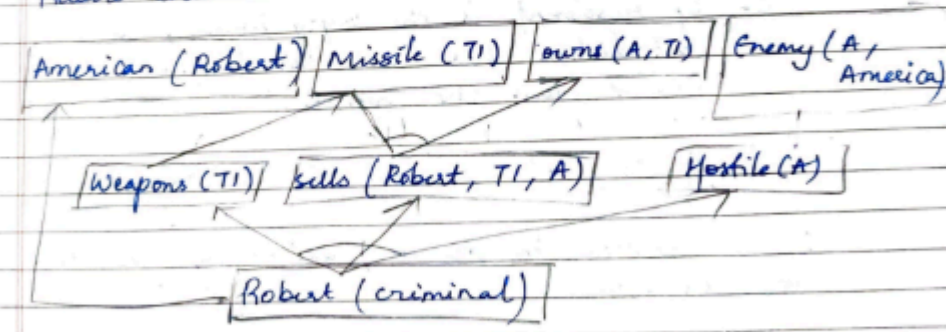
store
67

add q' to new
ϕ ← UNIFY (q', α)
if ϕ is not fail then
return ϕ

add new to KB
return false

Example : As per law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, and all the missiles were sold to it by Robert, who is an American citizen.

Prove that robert is criminal.



American (P) ∧ Weapon (q) ∧ sells (p, q, r)
∧ Hostile (r)
→ Criminal (P)

∃ x owns (A, x) ∧ Missiles (x)

Owns (A, T1)

Missile (T1)

∀ x Missile (x) ∧ Owns (A, x) → sells (Robert, x, A)

Missile (x) ⇒ Weapon (x)

∀ x Enemy (x, American) → Hostile (x)

Code:

```
# Define the knowledge base with facts and rules
knowledge_base = [
    # Rule: Selling weapons to a hostile nation makes one a criminal
    {
        "type": "rule",
        "if": [
            {"type": "sells", "seller": "?X", "item": "?Z", "buyer": "?Y"},
            {"type": "hostile_nation", "nation": "?Y"},
            {"type": "citizen", "person": "?X", "country": "america"}
        ],
        "then": {"type": "criminal", "person": "?X"}
    },
    # Facts
    {"type": "hostile_nation", "nation": "CountryA"},
    {"type": "sells", "seller": "Robert", "item": "missiles", "buyer":
"CountryA"},
    {"type": "citizen", "person": "Robert", "country": "america"}
]


# Forward chaining function
def forward_reasoning(kb, query):
    inferred = []  # Track inferred facts
    while True:
        new_inferences = []
        for rule in [r for r in kb if r["type"] == "rule"]:
            conditions = rule["if"]
            conclusion = rule["then"]
            substitutions = {}
            if match_conditions(conditions, kb, substitutions):
                inferred_fact = substitute(conclusion, substitutions)
                if inferred_fact not in kb and inferred_fact not in
new_inferences:
                    new_inferences.append(inferred_fact)
        if not new_inferences:
            break
        kb.extend(new_inferences)
        inferred.extend(new_inferences)
    return query in kb
```

```
# Helper to match conditions
def match_conditions(conditions, kb, substitutions):
    for condition in conditions:
        if not any(match_fact(condition, fact, substitutions) for fact in
kb):
            return False
    return True


# Helper to match a single fact
def match_fact(condition, fact, substitutions):
    if condition["type"] != fact["type"]:
        return False
    for key, value in condition.items():
        if key == "type":
            continue
        if isinstance(value, str) and value.startswith("?"):  # Variable
            variable = value
            if variable in substitutions:
                if substitutions[variable] != fact[key]:
                    return False
            else:
                substitutions[variable] = fact[key]
        elif fact[key] != value:  # Constant
            return False
    return True


# Substitute variables with their values
def substitute(conclusion, substitutions):
    result = conclusion.copy()
    for key, value in conclusion.items():
        if isinstance(value, str) and value.startswith("?"):
            result[key] = substitutions[value]
    return result


# Query: Is Robert a criminal?
query = {"type": "criminal", "person": "Robert"}
```

```
# Run the reasoning algorithm
if forward_reasoning(knowledge_base, query):
    print("Robert is a criminal.")
else:
    print("Could not prove that Robert is a criminal.")
```

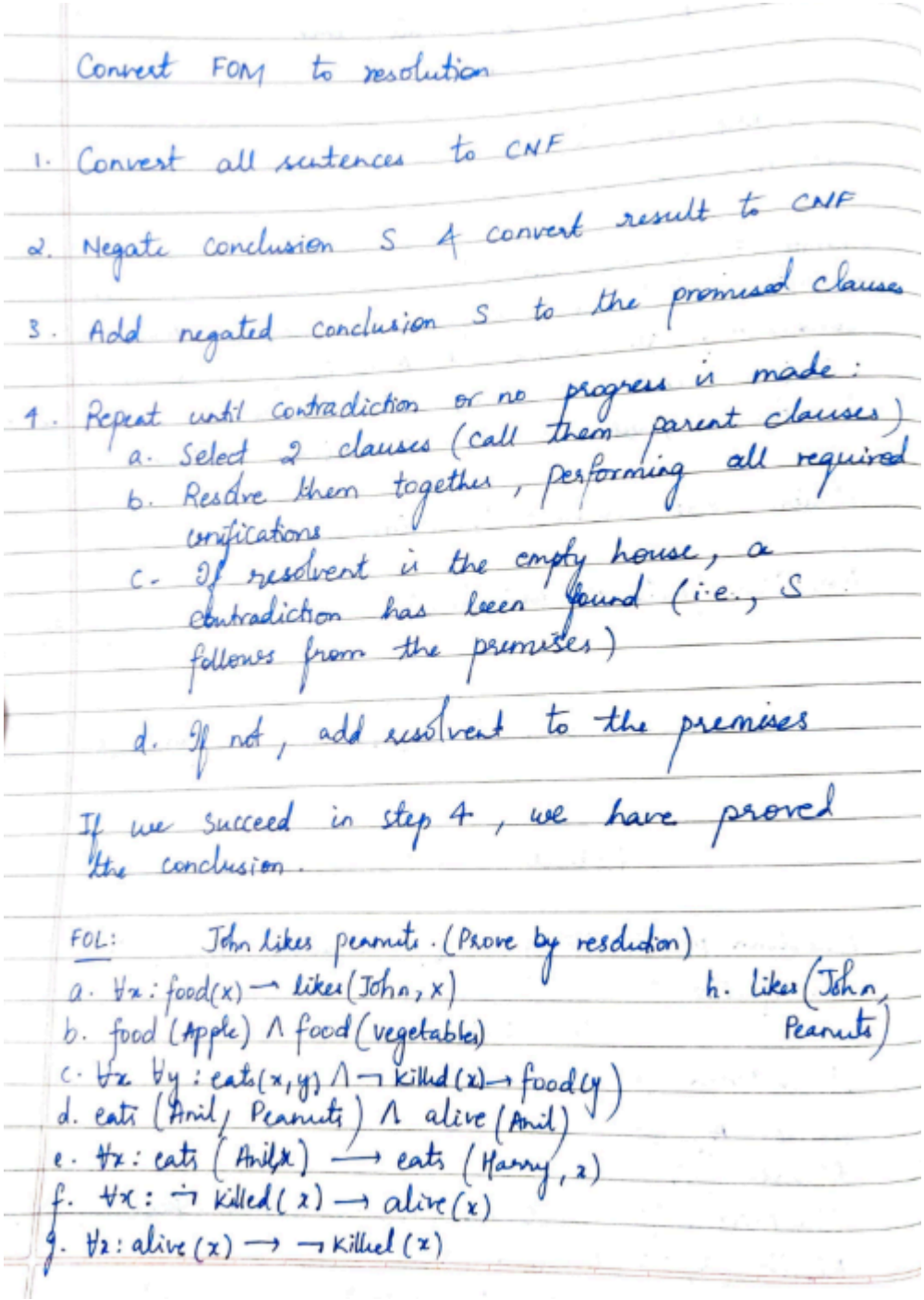
OUTPUT:

```
Robert is a criminal.
```

Program 9:

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

Algorithm:



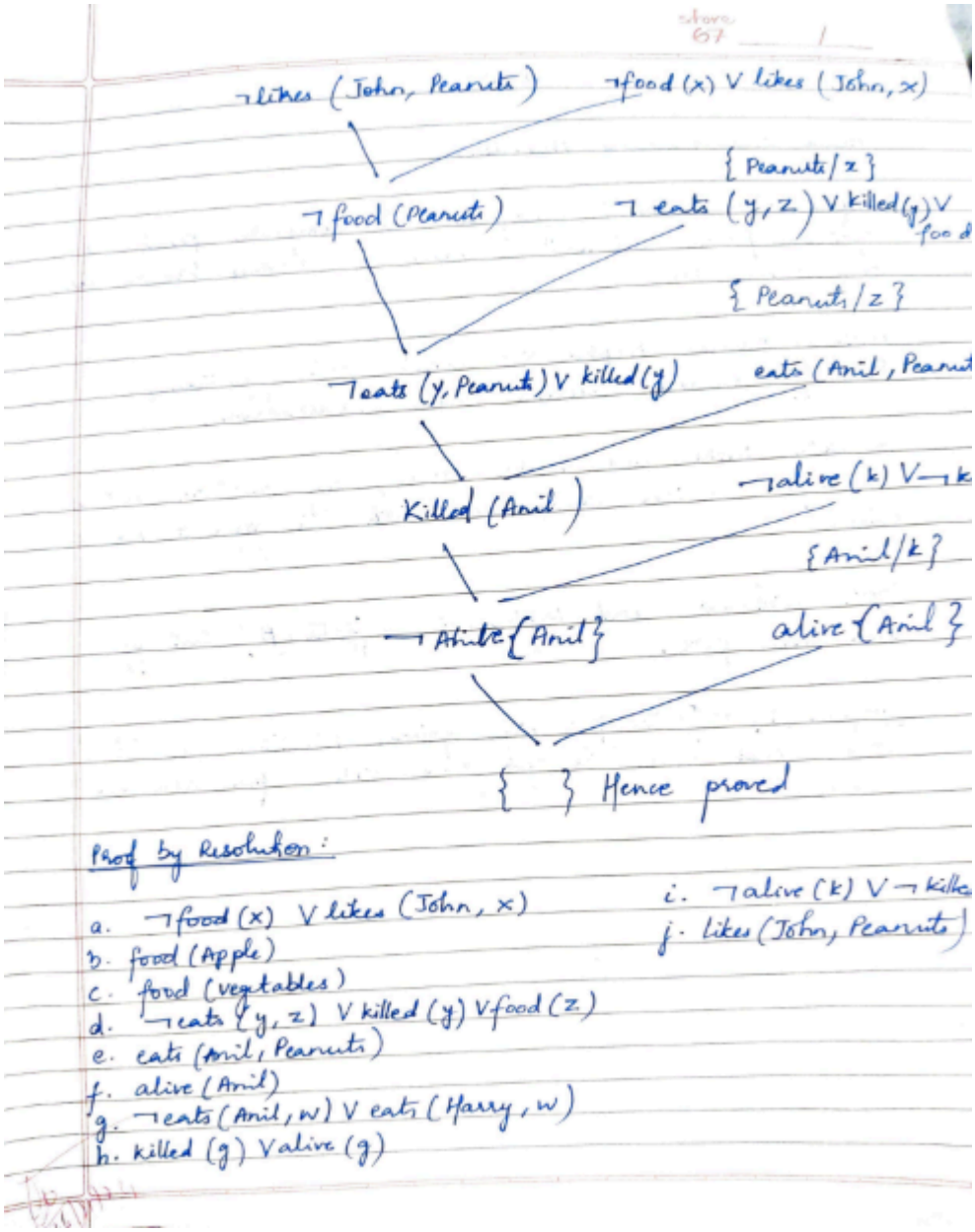
Convert FOM to resolution

1. Convert all sentences to CNF
2. Negate conclusion S & convert result to CNF
3. Add negated conclusion S to the premised clauses
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 house, a contradiction has been found (i.e., S follows from the premises)
   d. If not, add resolvent to the premises

If we succeed in step 4, we have proved the conclusion.

FOL: John likes peanuts. (Prove by resolution)

a. $\forall x : food(x) \rightarrow likes(John, x)$
b. $food(Apple) \wedge food(vegetables)$
c. $\forall x \forall y : eats(x, y) \wedge \neg killed(x) \rightarrow food(y)$
d. $eats(Anil, Peanuts) \wedge alive(Anil)$
e. $\forall x : eats(Anil, x) \rightarrow eats(Harry, x)$
f. $\forall x : \neg killed(x) \rightarrow alive(x)$
g. $\forall x : alive(x) \rightarrow \neg killed(x)$
h. $likes(John, Peanuts)$



Proof by Resolution:

a. ¬food (x) V likes (John, x)
b. food (Apple)
c. food (vegetables)
d. ¬eats (y, z) V killed (y) V food (z)
e. eats (Anil, Peanuts)
f. alive (Anil)
g. ¬eats (Anil, w) V eats (Harry, w)
h. killed (g) V alive (g)
i. ¬alive (k) V ¬kille
j. likes (John, Peanuts)

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'}")
```

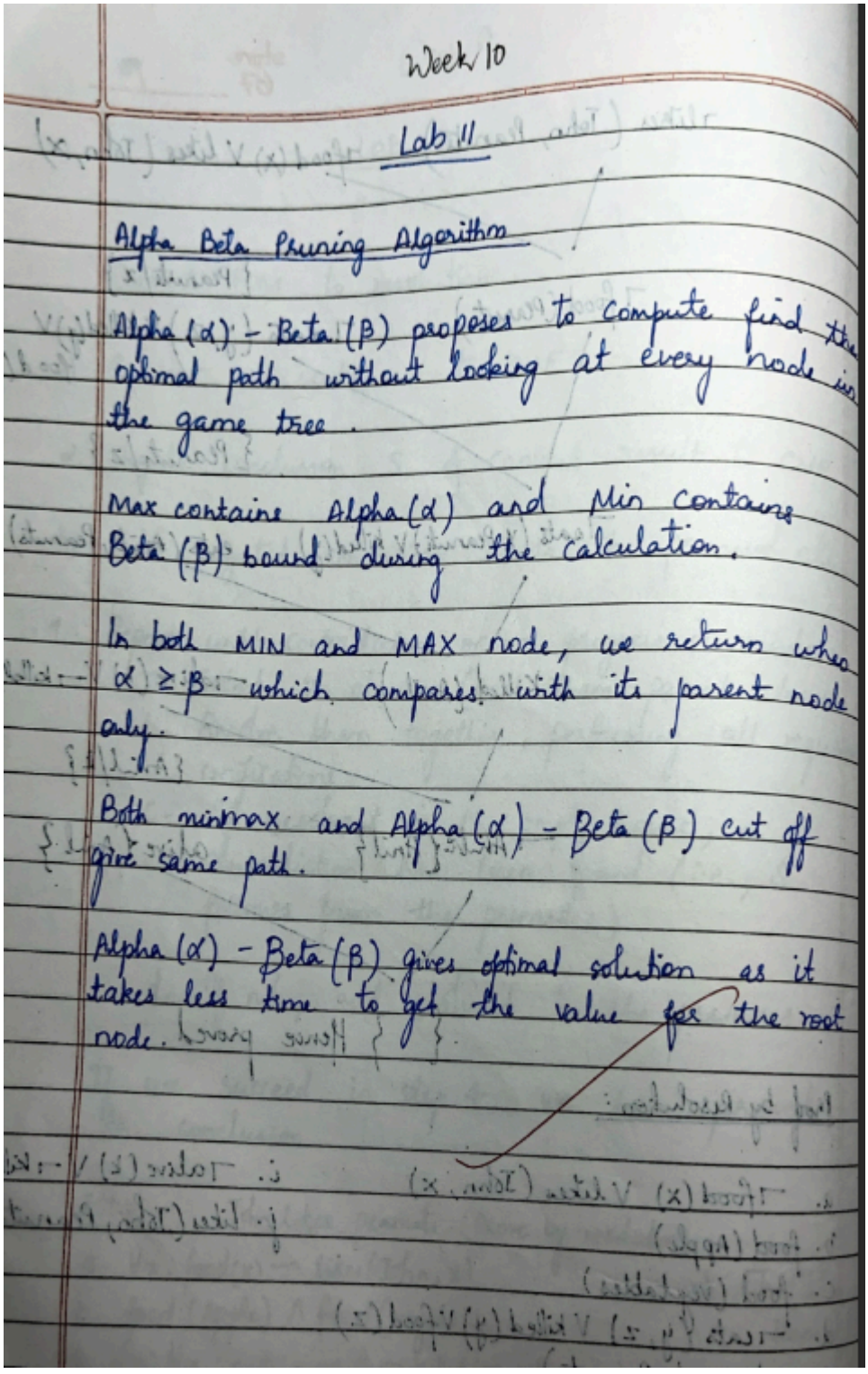
OUTPUT:

```
Does John like peanuts? Yes
```

Program 10:

Implement Alpha-Beta Pruning.

Algorithm:



Week 10

Lab 11

Alpha Beta Pruning Algorithm

Alpha (α) - Beta (β) proposes to compute find the optimal path without looking at every node in the game tree.

Max contains Alpha (α) and Min contains Beta (β) bound during the calculation.

In both MIN and MAX node, we return when α ≥ β which compares with its parent node only.

Both minmax and Alpha (α) - Beta (β) cut off give same path.

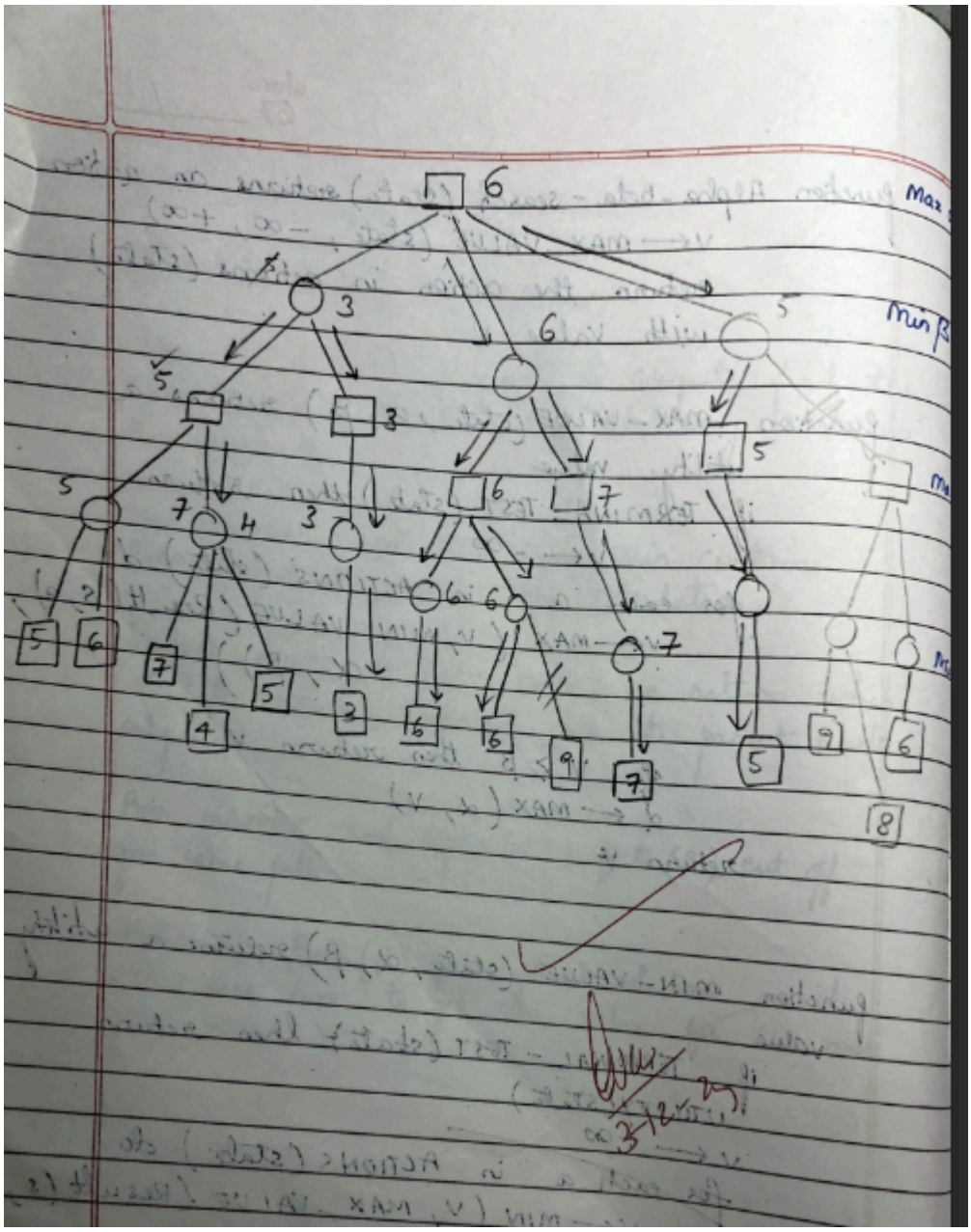
Alpha (α) - Beta (β) gives optimal solution as it takes less time to get the value for the root node.

store
67 /

function Alpha-beta-search (state) returns an action
    v ← MAX_VALUE (state, −∞, +∞)
    return the action in actions (state)
    with value v

function MAX-VALUE (state, α, β) returns a
utility value
    if TERMINAL_TEST (state) then return
        v ← −∞
    for each a in ACTIONS (state) do
        v ← MAX (v, MIN_VALUE (Result (s, a),
                α, β))
        if v ≥ β then return v
        α ← MAX (α, v)
    return v

function MIN-VALUE (state, α, β) returns a utility
value
    if TERMINAL-TEST (state) then return
    UTILITY (state)
    v ← +∞
    for each a in ACTIONS (state) do
        v ← MIN (v, MAX_VALUE (Result (s,
                α, β)
        if v ≤ α then return v
    return v


Max
Min
6
3
5
6
5
3
5
7
4
3
6
7
5
6
7
5
2
6
6
9
7
5
9
6
8

Code:

```
import math


def minimax(node, depth, is_maximizing):
    """
    Implement the Minimax algorithm to solve the decision tree.

    Parameters:
    node (dict): The current node in the decision tree, with the
following structure:
        {
            'value': int,
            'left': dict or None,
            'right': dict or None
        }
    depth (int): The current depth in the decision tree.
    is_maximizing (bool): Flag to indicate whether the current
player is the maximizing player.

    Returns:
    int: The utility value of the current node.
    """
    # Base case: Leaf node
    if node['left'] is None and node['right'] is None:
        return node['value']



    # Recursive case
    if is_maximizing:
        best_value = -math.inf
        if node['left']:
            best_value = max(best_value, minimax(node['left'],
depth + 1, False))
        if node['right']:
            best_value = max(best_value, minimax(node['right'],
depth + 1, False))
        return best_value
    else:
        best_value = math.inf
        if node['left']:
```

```
            best_value = min(best_value, minimax(node['left'],
depth + 1, True))
        if node['right']:
            best_value = min(best_value, minimax(node['right'],
depth + 1, True))
        return best_value


# Example usage
decision_tree = {
    'value': 5,
    'left': {
        'value': 6,
        'left': {
            'value': 7,
            'left': {
                'value': 4,
                'left': None,
                'right': None
            },
            'right': {
                'value': 5,
                'left': None,
                'right': None
            }
        },
        'right': {
            'value': 3,
            'left': {
                'value': 6,
                'left': None,
                'right': None
            },
            'right': {
                'value': 9,
                'left': None,
                'right': None
            }
        }
    },
    'right': {
```

```
        'value': 8,
        'left': {
            'value': 7,
            'left': {
                'value': 6,
                'left': None,
                'right': None
            },
            'right': {
                'value': 9,
                'left': None,
                'right': None
            }
        },
        'right': {
            'value': 8,
            'left': {
                'value': 6,
                'left': None,
                'right': None
            },
            'right': None
        }
    }
}


# Find the best move for the maximizing player
best_value = minimax(decision_tree, 0, True)
print(f"The best value for the maximizing player is: {best_value}")
```

OUTPUT:

```
The best value for the maximizing player is: 6
```