VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Shree Varna M (1BM22CS263)

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 **Shree varna M (1BM22CS263) ,** 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.

Saritha A. N Assistant Professor Department of CSE, BMSCE	Dr. Jyothi S Nayak Professor & HOD Department of CSE, BMSCE

Index

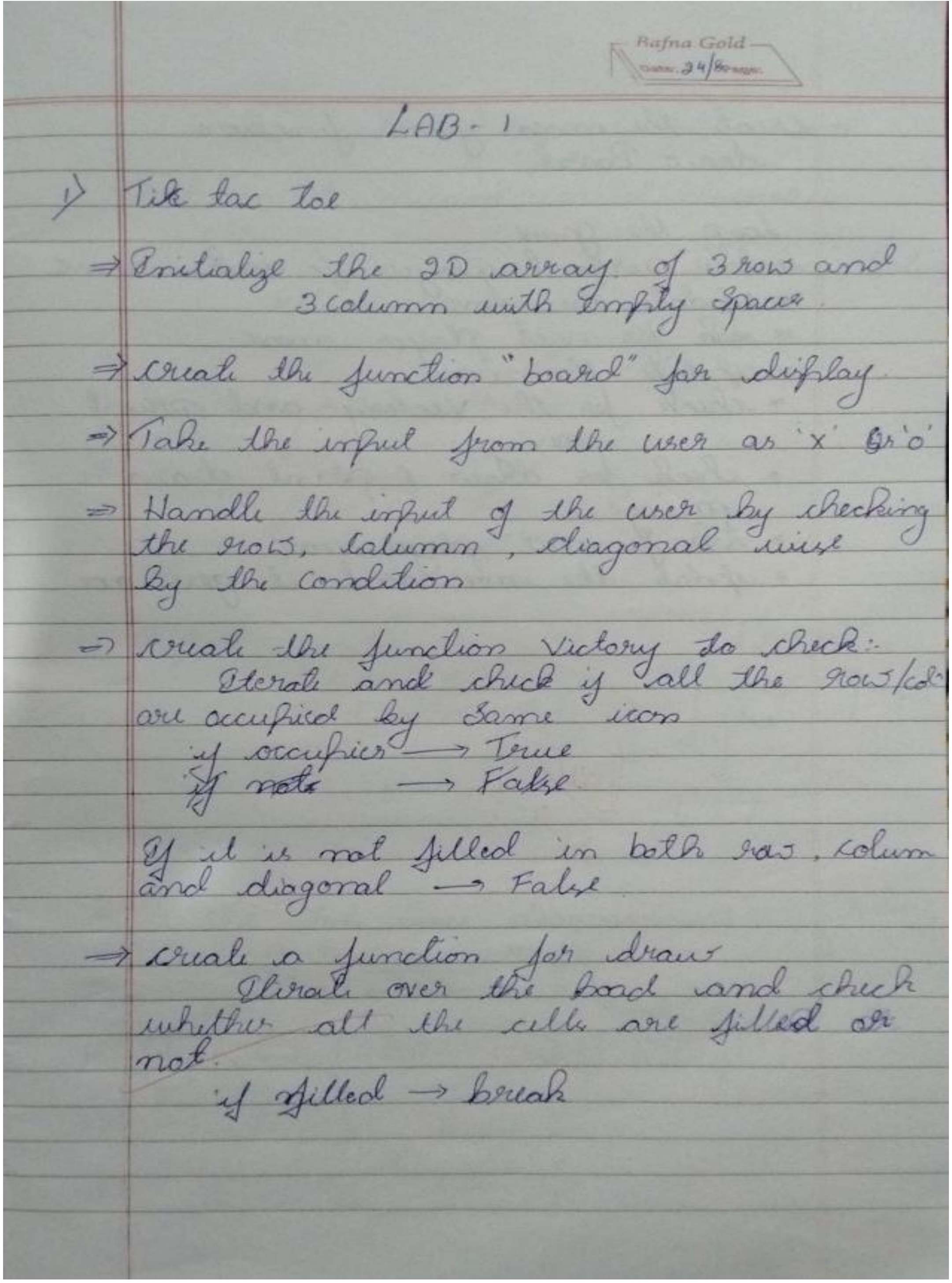
Github Link:
https://github.com/Shree-varna/AI

Program 1

Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Algorithm:



Date: 24/9

LAB - 1

1) Tik tac toe

⇒ Initialize the 2D array of 3 row and 3 column with empty spaces

⇒ create the function "board" for display

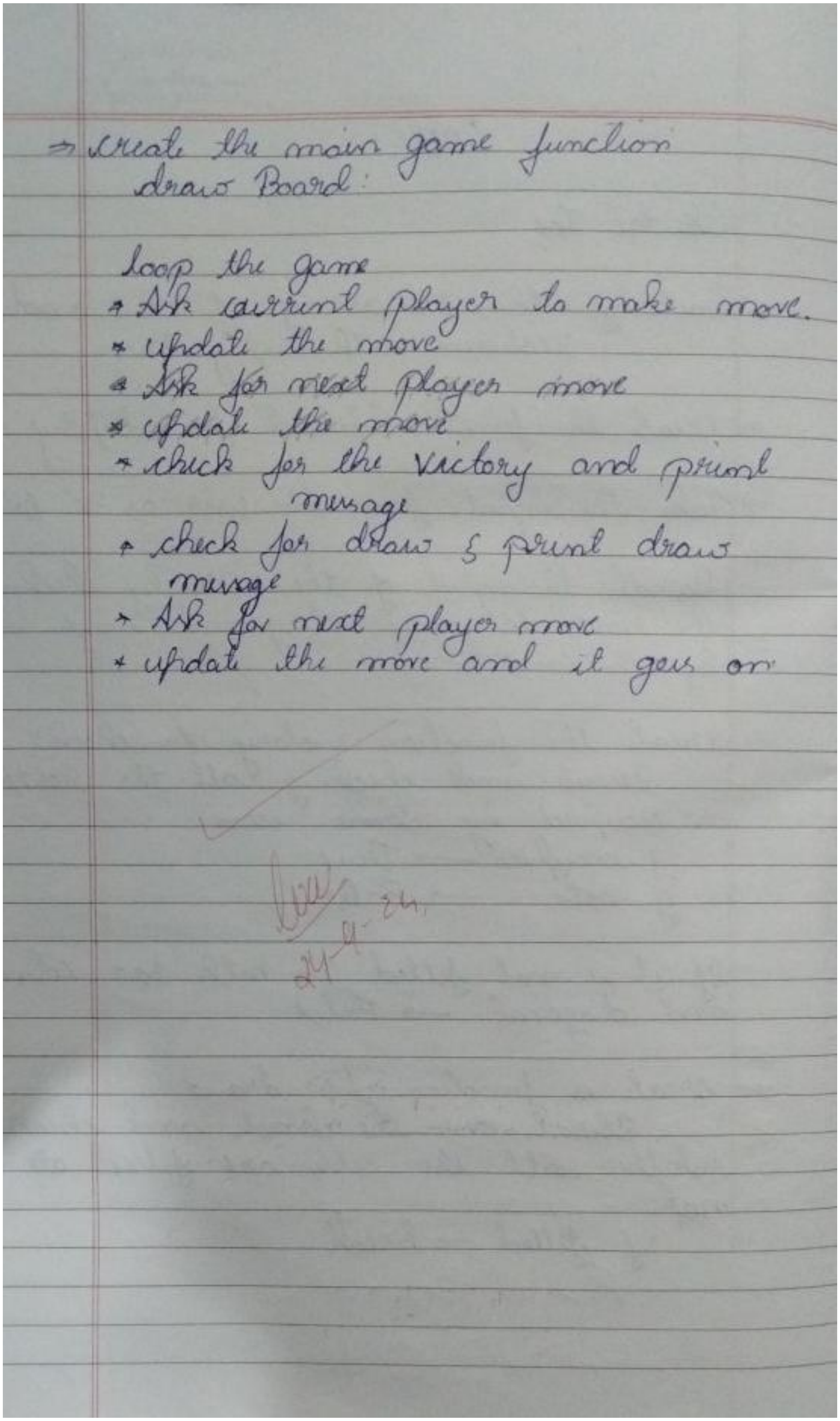
⇒ Take the input from the user as 'x' or 'o'

⇒ Handle the input of the user by checking the rows, column, diagonal wise by the condition

→ create the function victory to check:-
Iterate and check if all the rows/col are occupied by same icon
if occupies → True
if not → False

If it is not filled in both row, colum and diagonal → False

⇒ create a function for draw
Iterate over the board and check whether all the cells are filled or not.
if filled → break



⇒ create the main game function
draw Board:

loop the game

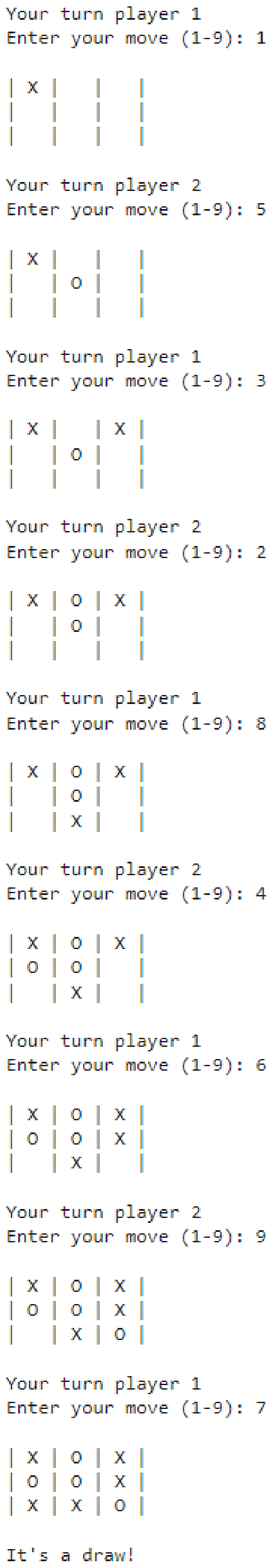
* Ask current player to make move.
* update the move
* Ask for next player move
* update the move
* check for the victory and print message
* check for draw & print draw message
* Ask for next player move
* update the move and it goes on

24-9-24

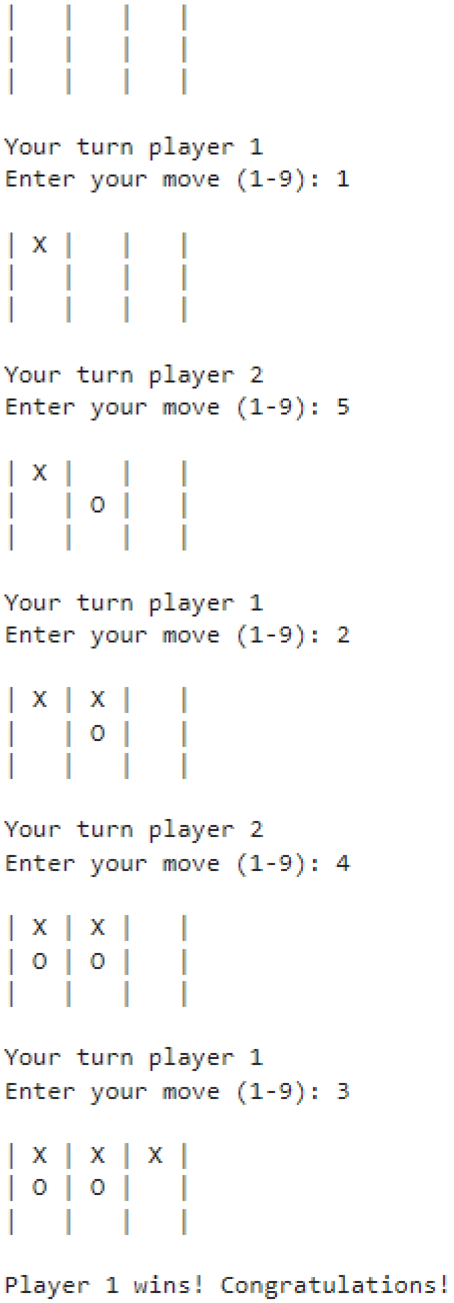
Code:

```
board = [' ' for _ in range(9)]

def draw_board():
    row1 = '| {} | {} | {} |'.format(board[0], board[1], board[2])
    row2 = '| {} | {} | {} |'.format(board[3], board[4], board[5])
    row3 = '| {} | {} | {} |'.format(board[6], board[7], board[8])

    print()
    print(row1)
    print(row2)
    print(row3)
    print()

def player_move(icon):
    if icon == 'X':
        number = 1
    elif icon == 'O':
        number = 2

    print("Your turn player {}".format(number))

    choice = int(input("Enter your move (1-9): "))
    if board[choice - 1] == ' ':
        board[choice - 1] = icon
    else:
        print()
        print("That space is taken!")

def is_victory(icon):
    if (board[0] == icon and board[1] == icon and board[2] == icon) or \
        (board[3] == icon and board[4] == icon and board[5] == icon) or \
        (board[6] == icon and board[7] == icon and board[8] == icon) or \
        (board[0] == icon and board[3] == icon and board[6] == icon) or \
        (board[1] == icon and board[4] == icon and board[7] == icon) or \
        (board[2] == icon and board[5] == icon and board[8] == icon) or \
        (board[0] == icon and board[4] == icon and board[8] == icon) or \
        (board[2] == icon and board[4] == icon and board[6] == icon):
        return True
    else:
        return False

def is_draw():
    if ' ' not in board:
        return True
    else:
        return False

def play_game():
    draw_board()
    while True:
        player_move('X')
        draw_board()
        if is_victory('X'):
            print("Player 1 wins! Congratulations!")
            break
        elif is_draw():
            print("It's a draw!")
            break
        player_move('O')
        draw_board()
        if is_victory('O'):
            print("Player 2 wins! Congratulations!")
            break
        elif is_draw():
            print("It's a draw!")
            break

play_game()
```

OUTPUT :



```
Your turn player 1
Enter your move (1-9): 1

| X |   |   |
|   |   |   |
|   |   |   |

Your turn player 2
Enter your move (1-9): 5

| X |   |   |
|   | O |   |
|   |   |   |

Your turn player 1
Enter your move (1-9): 3

| X |   | X |
|   | O |   |
|   |   |   |

Your turn player 2
Enter your move (1-9): 2

| X | O | X |
|   | O |   |
|   |   |   |

Your turn player 1
Enter your move (1-9): 8

| X | O | X |
|   | O |   |
|   | X |   |

Your turn player 2
Enter your move (1-9): 4

| X | O | X |
| O | O |   |
|   | X |   |

Your turn player 1
Enter your move (1-9): 6

| X | O | X |
| O | O | X |
|   | X |   |

Your turn player 2
Enter your move (1-9): 9

| X | O | X |
| O | O | X |
|   | X | O |

Your turn player 1
Enter your move (1-9): 7

| X | O | X |
| O | O | X |
| X | X | O |

It's a draw!
```



```
|   |   |   |
|   |   |   |
|   |   |   |

Your turn player 1
Enter your move (1-9): 1

| X |   |   |
|   |   |   |
|   |   |   |

Your turn player 2
Enter your move (1-9): 5

| X |   |   |
|   | O |   |
|   |   |   |

Your turn player 1
Enter your move (1-9): 2

| X | X |   |
|   | O |   |
|   |   |   |

Your turn player 2
Enter your move (1-9): 4

| X | X |   |
| O | O |   |
|   |   |   |

Your turn player 1
Enter your move (1-9): 3

| X | X | X |
| O | O |   |
|   |   |   |

Player 1 wins! Congratulations!
```

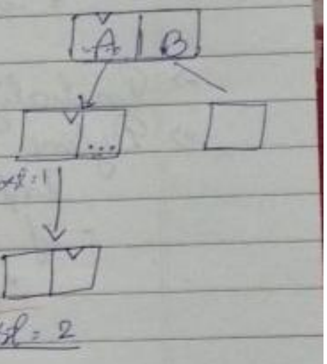
Lab-2

Implement Vaccum world cleaner.

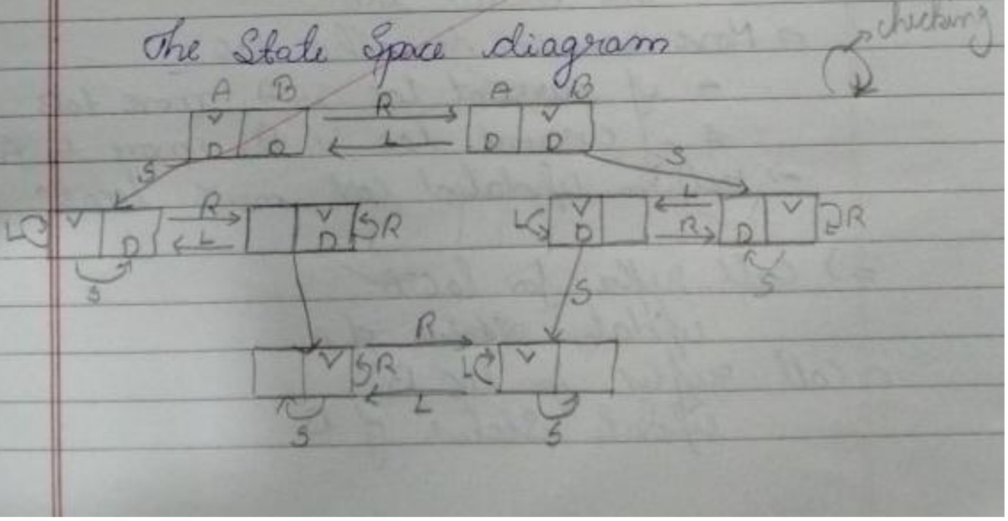
Function Reflex-Vaccum-Agent ([Location, Status])
return an action

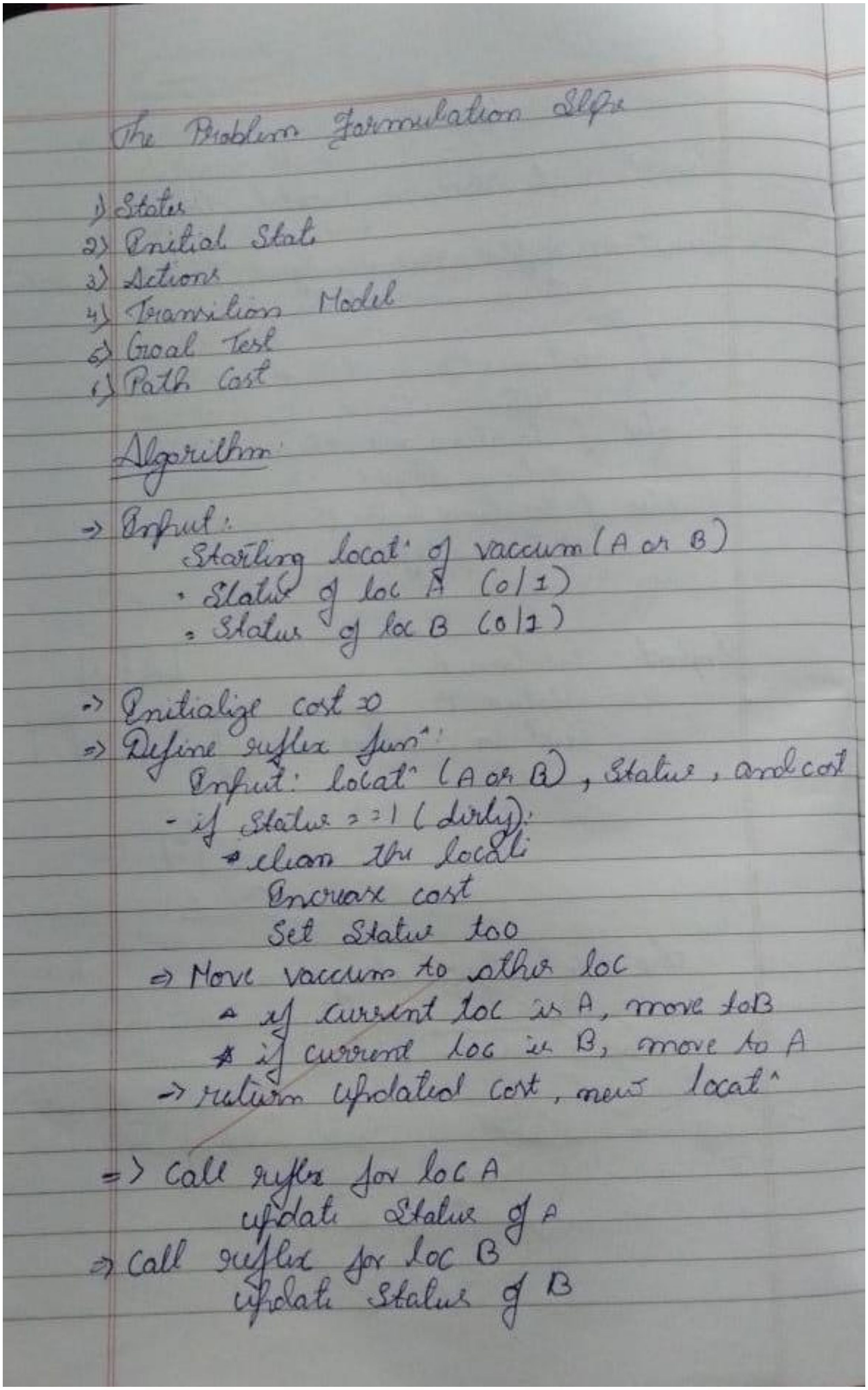
```
if Status = Dirty Then
    return Suck
else if location = A then
    return Right
else if location = B then
    return Left
```

Input: Location A
Status 0
Dust in another = 1



The State Space diagram



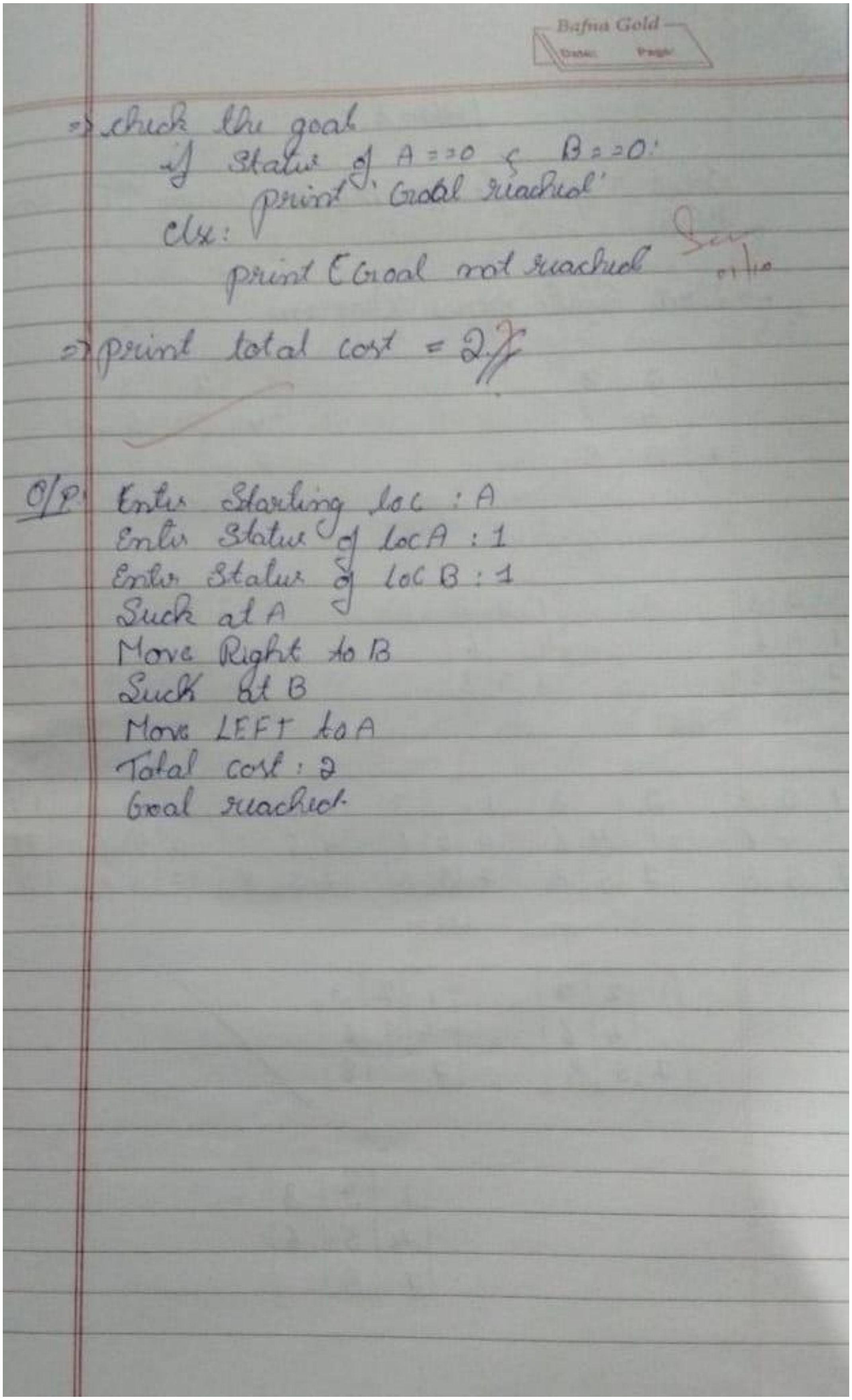


The Problem Formulation Steps

1) States
2) Initial State
3) Actions
4) Transition Model
5) Goal Test
6) Path Cost

Algorithm:

```
-> Input:
      Starting locat' of vaccum (A or B)
      . Status of loc A (0/1)
      . Status of loc B (0/1)

-> Initialize cost =0
=> Define reflex fun^:
      Input: locat^ (A or B), Status, and cost
      - if Status ==1 (dirty):
          * clean the locati
            Increase cost
            Set Status to 0
      => Move vaccum to other loc
          * if current loc is A, move to B
          * if current loc is B, move to A
      -> return updated cost, new locat^

=> Call reflex for loc A
        update Status of A
=> Call reflex for loc B
        update Status of B
```



```
=> check the goal
     if Status of A ==0 & B==0:
          print 'Goal reached'
     else:
          print (Goal not reached

=> print total cost = 2
```

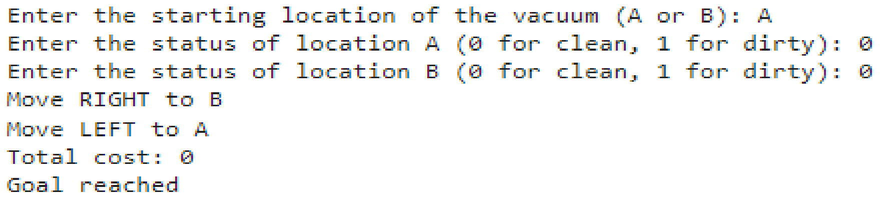
O/P:

```
Enter Starting loc : A
Enter Status of loc A : 1
Enter Status of loc B : 1
Suck at A
Move Right to B
Suck at B
Move LEFT to A
Total cost : 2
Goal reached.
```

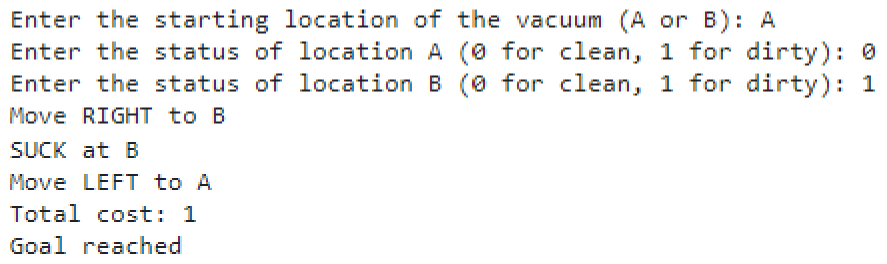
Code :

```
def reflex(loc, status, cost):
    s = status  # Track the current status of the location
    if status == 1:  # If the location is dirty
        cost += 1
        print(f"SUCK at {loc}")
        s = 0  # The location is now clean
    if loc == "A":
        print("Move RIGHT to B")
        loc = "B"  # Move to B
    elif loc == "B":
        print("Move LEFT to A")
        loc = "A"  # Move to A
    return cost, loc, s  # Return updated cost, location, and status

def goal(a_status, b_status):
    if a_status == 0 and b_status == 0:
        print("Goal reached")
    else:
        print("Goal not reached")

loc = input("Enter the starting location of the vacuum (A or B): ")
cost = 0
a_status = int(input("Enter the status of location A (0 for clean, 1 for dirty): "))
b_status = int(input("Enter the status of location B (0 for clean, 1 for dirty): "))

cost, loc, a_status = reflex("A", a_status, cost)

cost, loc, b_status = reflex("B", b_status, cost)

print(f"Total cost: {cost}")

goal(a_status, b_status)
```

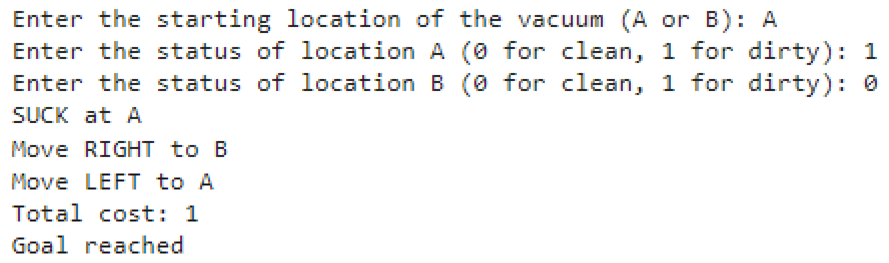
OUTPUT :



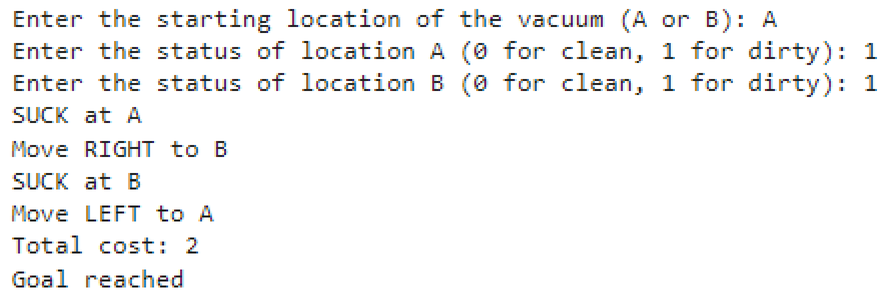
```
Enter the starting location of the vacuum (A or B): A
Enter the status of location A (0 for clean, 1 for dirty): 0
Enter the status of location B (0 for clean, 1 for dirty): 0
Move RIGHT to B
Move LEFT to A
Total cost: 0
Goal reached
```



```
Enter the starting location of the vacuum (A or B): A
Enter the status of location A (0 for clean, 1 for dirty): 0
Enter the status of location B (0 for clean, 1 for dirty): 1
Move RIGHT to B
SUCK at B
Move LEFT to A
Total cost: 1
Goal reached
```



```
Enter the starting location of the vacuum (A or B): A
Enter the status of location A (0 for clean, 1 for dirty): 1
Enter the status of location B (0 for clean, 1 for dirty): 0
SUCK at A
Move RIGHT to B
Move LEFT to A
Total cost: 1
Goal reached
```



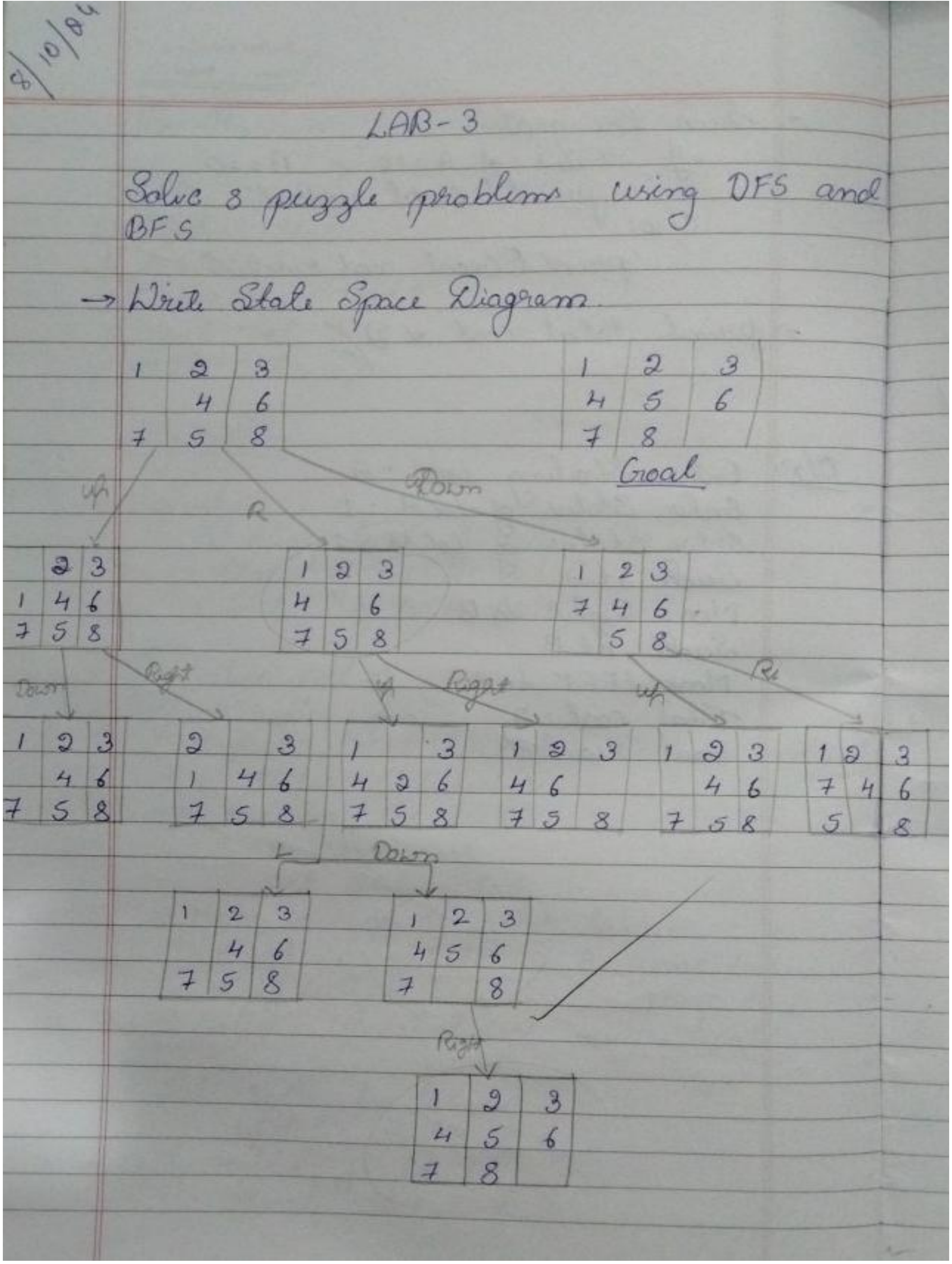
```
Enter the starting location of the vacuum (A or B): A
Enter the status of location A (0 for clean, 1 for dirty): 1
Enter the status of location B (0 for clean, 1 for dirty): 1
SUCK at A
Move RIGHT to B
SUCK at B
Move LEFT to A
Total cost: 2
Goal reached
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)

Implement Iterative deepening search

Algorithm:



8/10/24

LAB-3

Solve 8 puzzle problems using DFS and BFS

→ Write State Space Diagram.

1	2	3
	4	6
7	5	8

1	2	3
4	5	6
7	8	

Goal

	2	3
1	4	6
7	5	8

1	2	3
4		6
7	5	8

1	2	3
7	4	6
	5	8

1	2	3
	4	6
7	5	8

2		3
1	4	6
7	5	8

1		3
4	2	6
7	5	8

1	2	3
4	6	
7	5	8

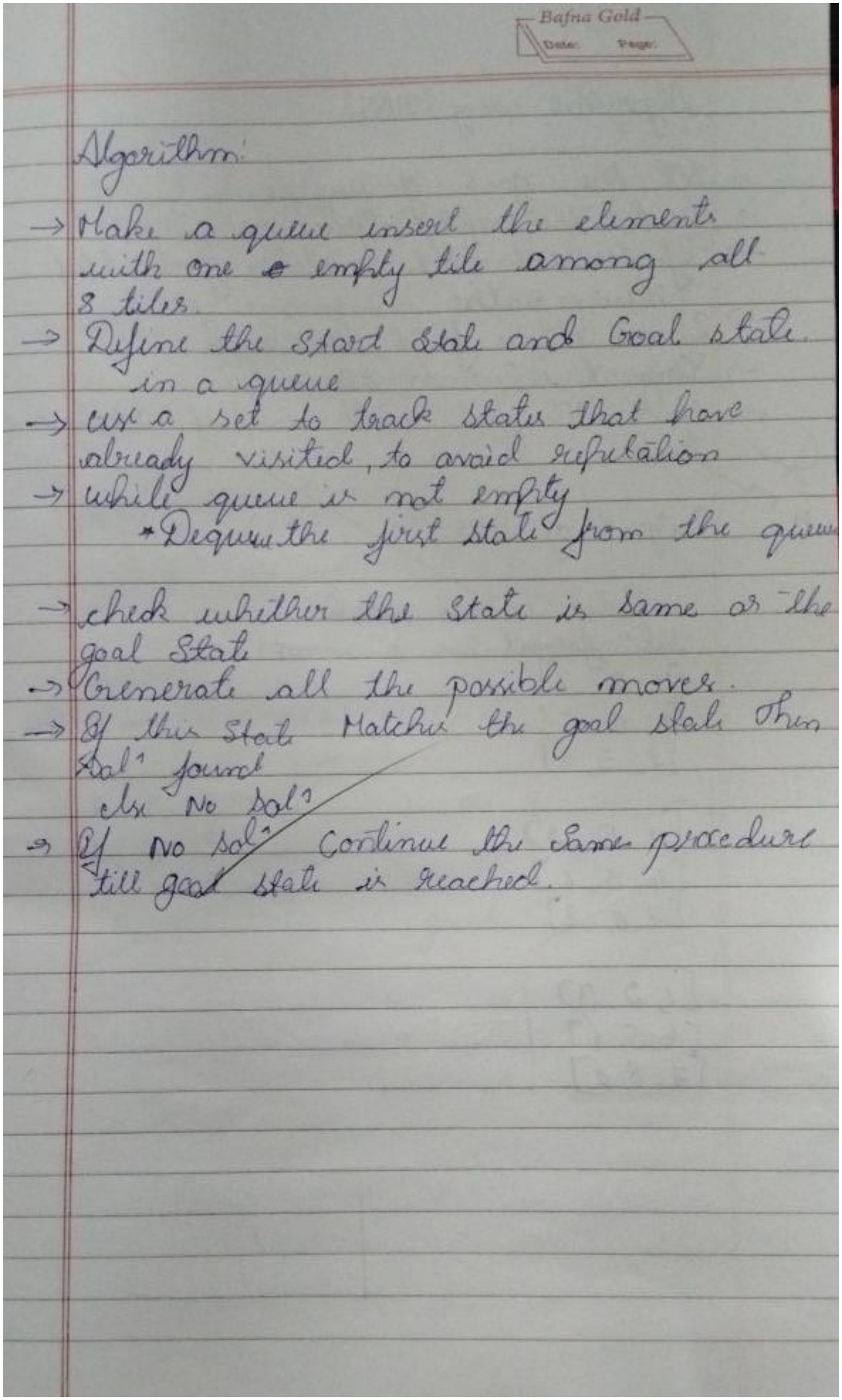
1	2	3
	4	6
7	5	8

1	2	3
7	4	6
5		8

1	2	3
	4	6
7	5	8

1	2	3
4	5	6
7		8

1	2	3
4	5	6
7	8	



Algorithm:

→ Make a queue insert the elements with one empty tile among all 8 tiles.

→ Define the Start State and Goal state in a queue

→ use a set to track states that have already visited, to avoid repetation

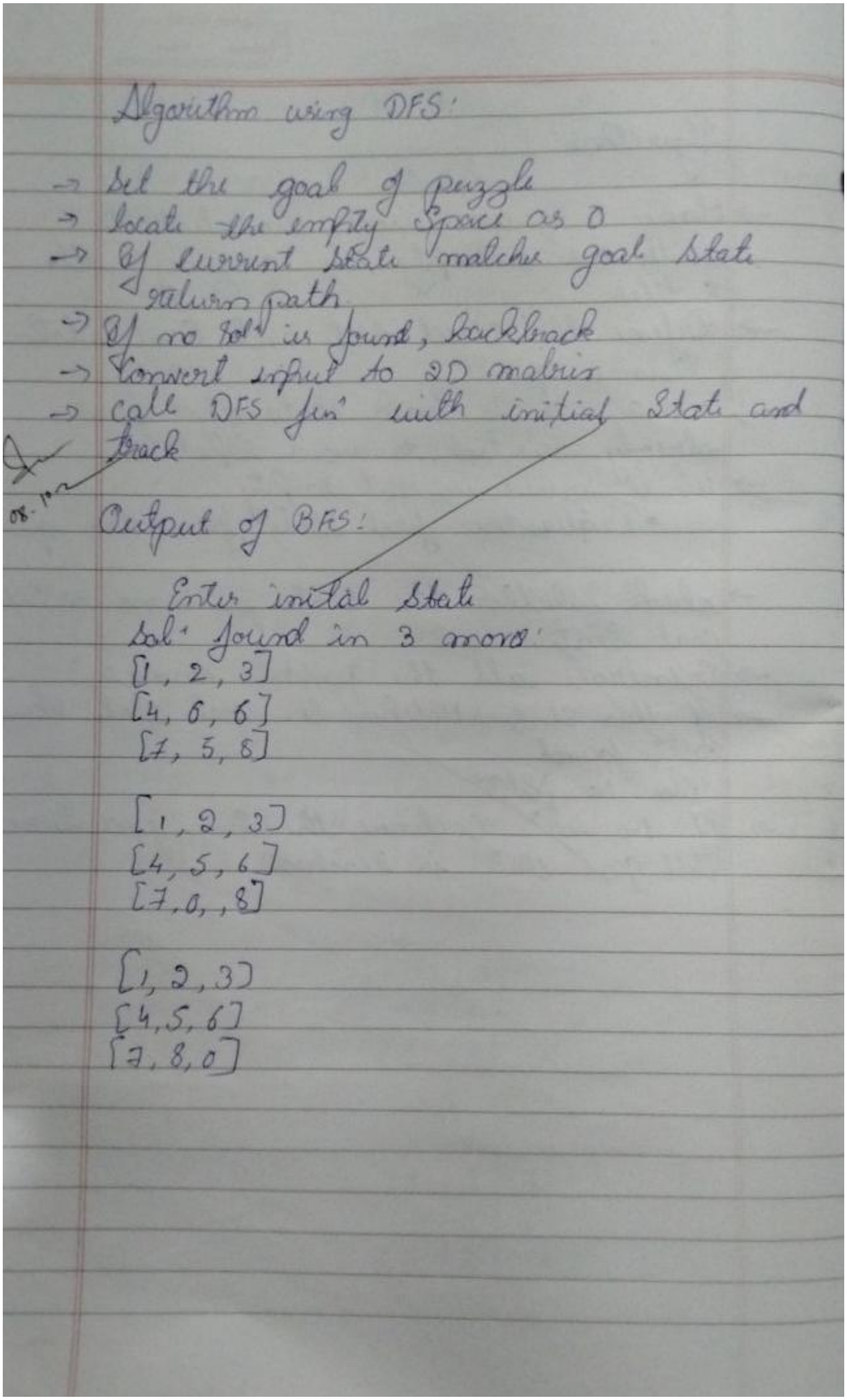
→ while queue is not empty
  *Dequeue the first state from the queue

→ check whether the state is same as the goal state

→ Generate all the possible moves.

→ If this state Matches the goal state then Sol^n found
  else No Sol^n

→ If No sol^n continue the same procedure till goal state is reached.



Algorithm using DFS:

→ Set the goal of puzzle
→ locate the empty space as 0
→ If current state matches goal state
  return path
→ If no sol^n is found, backtrack
→ Convert input to 2D matrix
→ call DFS fun^n with initial state and track

08.10

Output of BFS:

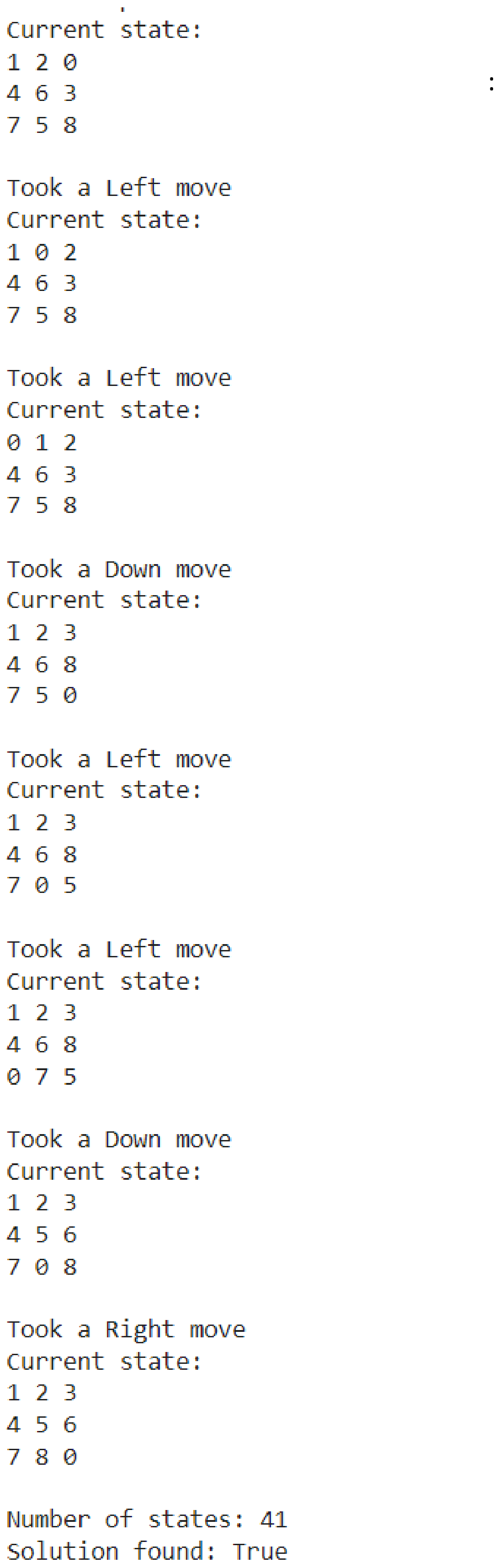
```
    Enter initial state
sol: found in 3 moves:
[1, 2, 3]
[4, 6, 6]
[7, 5, 8]

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

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

CODE :

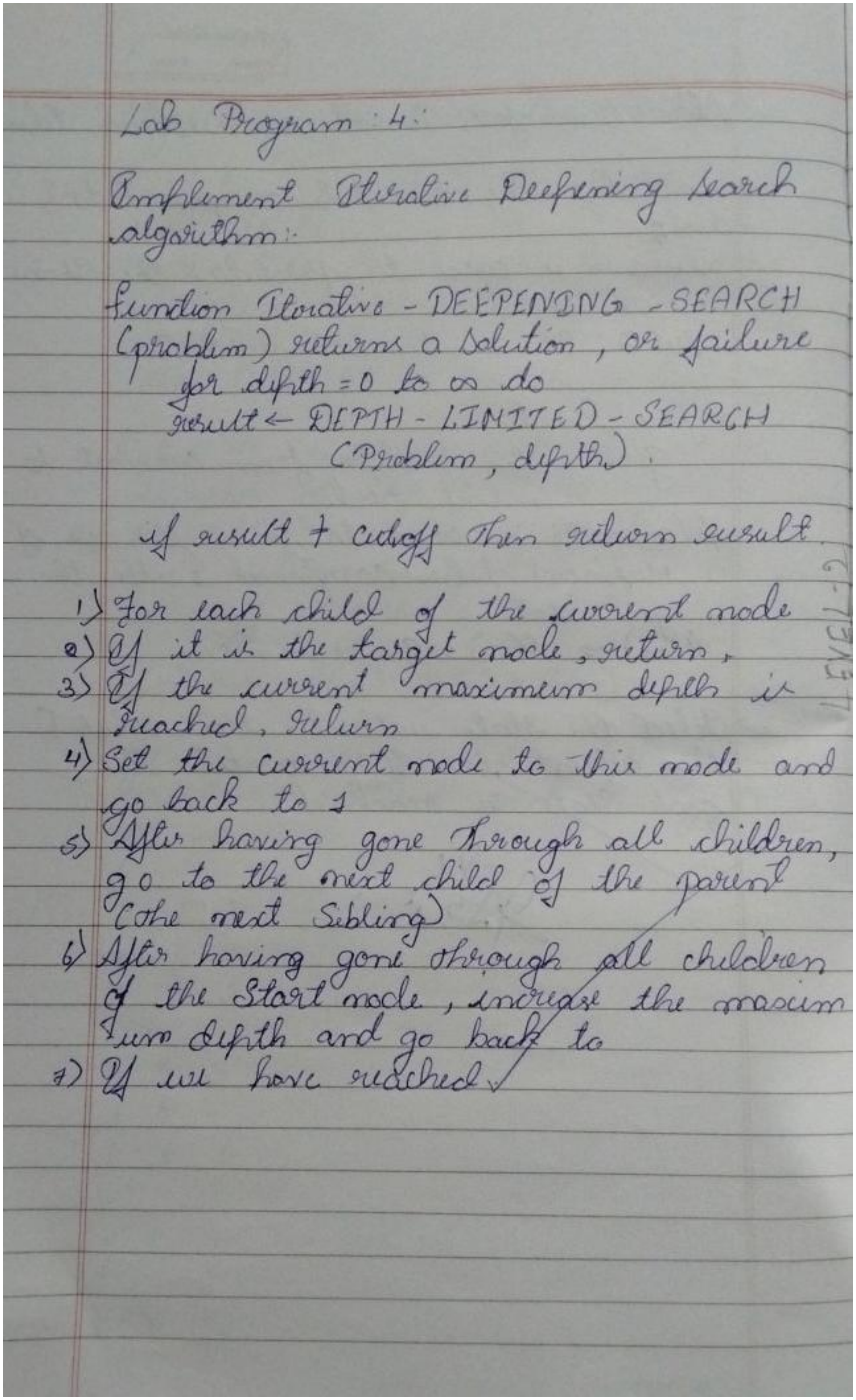
```
count = 0
def print_state(in_array):
global count
count += 1
for row in in_array:
print(' '.join(str(num) for num in row))
print()
def helper(goal, in_array, row, col, vis):
# Marking current position as visited
vis[row][col] = 1
drow = [-1, 0, 1, 0] # Dir for row: up, right, down, left
dcol = [0, 1, 0, -1] # Dir for column
dchange = ['Up', 'Right', 'Down', 'Left']
# Print current state
print("Current state:")
print_state(in_array)
# Check if the current state is the goal state
if in_array == goal:
print(f"Number of states: {count}")
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 0
4 6 3
7 5 8

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

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

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

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

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

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

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

Number of states: 41
Solution found: True
```

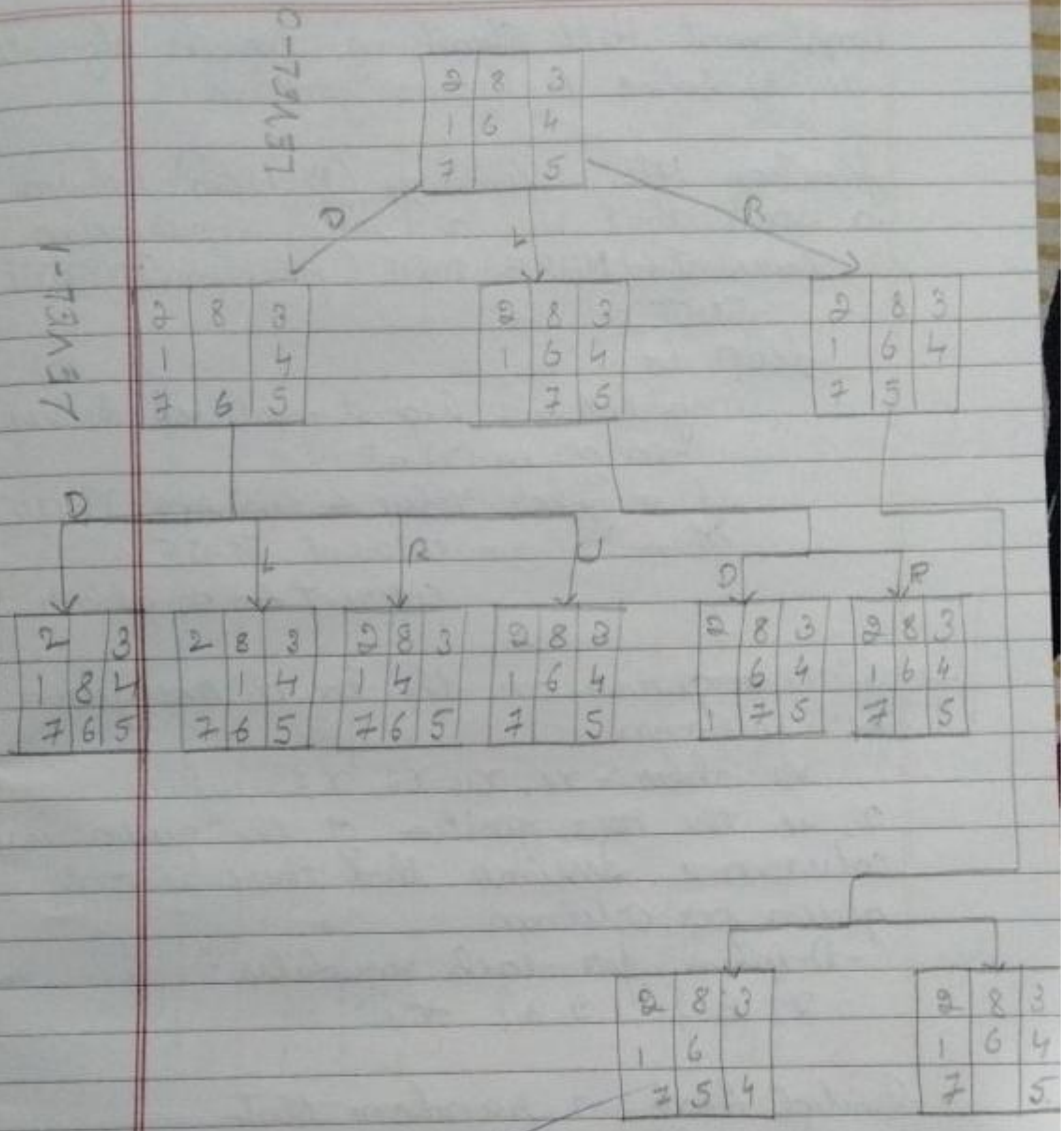


Lab Program : 4:

Implement Iterative Deepening Search algorithm:

```
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.
```

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 and 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
7) If we have reached.


LEVEL-0
2 8 3
1 6 4
7 5
D
L
R
LEVEL-1
2 8 3
1 4
7 6 5
2 8 3
1 6 4
7 5
2 8 3
1 6 4
7 5
D
L
R
U
LEVEL-2
2 3
1 8 4
7 6 5
2 8 3
1 4
7 6 5
2 8 3
1 4
7 6 5
2 8 3
1 6 4
7 5
D
R
2 8 3
6 4
1 7 5
2 8 3
1 6 4
7 5
2 8 3
1 6
7 5 4
2 8 3
1 6 4
7 5

Level = 2

CODE :

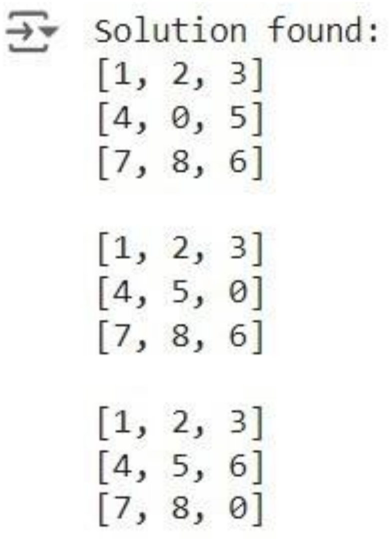
```
#iterative-deepening from collections import deque

class PuzzleState:

def __init__(self, board, zero_pos, moves=0, previous=None): self.board = board
self.zero_pos = zero_pos # Position of the zero tile self.moves = moves       #
Number of moves taken to reach this state self.previous = previous# For tracking
the path

def is_goal(self, goal_state):

return self.board == goal_state

def get_possible_moves(self): moves = [] x, y = self.zero_pos directions = [(-1, 0),
(1, 0), (0, -1), (0, 1)] # Up, Down, Left, Right for dx, dy in directions:

new_x, new_y = x + dx, y + dy if 0 <= new_x < 3 and 0 <= new_y < 3:

new_board = [row[:] for row in self.board] # Swap the zero tile with the adjacent
tile

new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],

new_board[x][y] moves.append((new_board, (new_x, new_y))) return moves


def ids(initial_state, goal_state, max_depth):

for depth in range(max_depth):

visited = set() result = dls(initial_state, goal_state, depth, visited) if result: return
result return None

def dls(state, goal_state, depth, visited):

if state.is_goal(goal_state):

return state

if depth == 0: return None

visited.add(tuple(map(tuple, state.board))) # Mark this state as visited for
new_board, new_zero_pos in state.get_possible_moves():

new_state = PuzzleState(new_board, new_zero_pos, state.moves + 1, state) if
tuple(map(tuple, new_board)) not in visited:

result = dls(new_state, goal_state, depth - 1, visited) if result:

return result                                  # Unmark this state
```

```
visited.remove(tuple(map(tuple,
state.board))) return None

def print_solution(solution):
path = [] while solution:
path.append(solution.board) solution = solution.previous
for board in reversed(path):
for row in board:
print(row)
print()
# Define the initial state and goal state initial_state = PuzzleState( board=[[1, 2, 3],
[4, 0, 5],
[7, 8, 6]], zero_pos=(1, 1)
)


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

```
# Perform Iterative Deepening Search max_depth = 20 # You can adjust this value solution =
ids(initial_state, goal_state, max_depth)
```

```
if solution:
print("Solution found:") print_solution(solution)
else:
print("No solution found.")
```

OUTPUT:

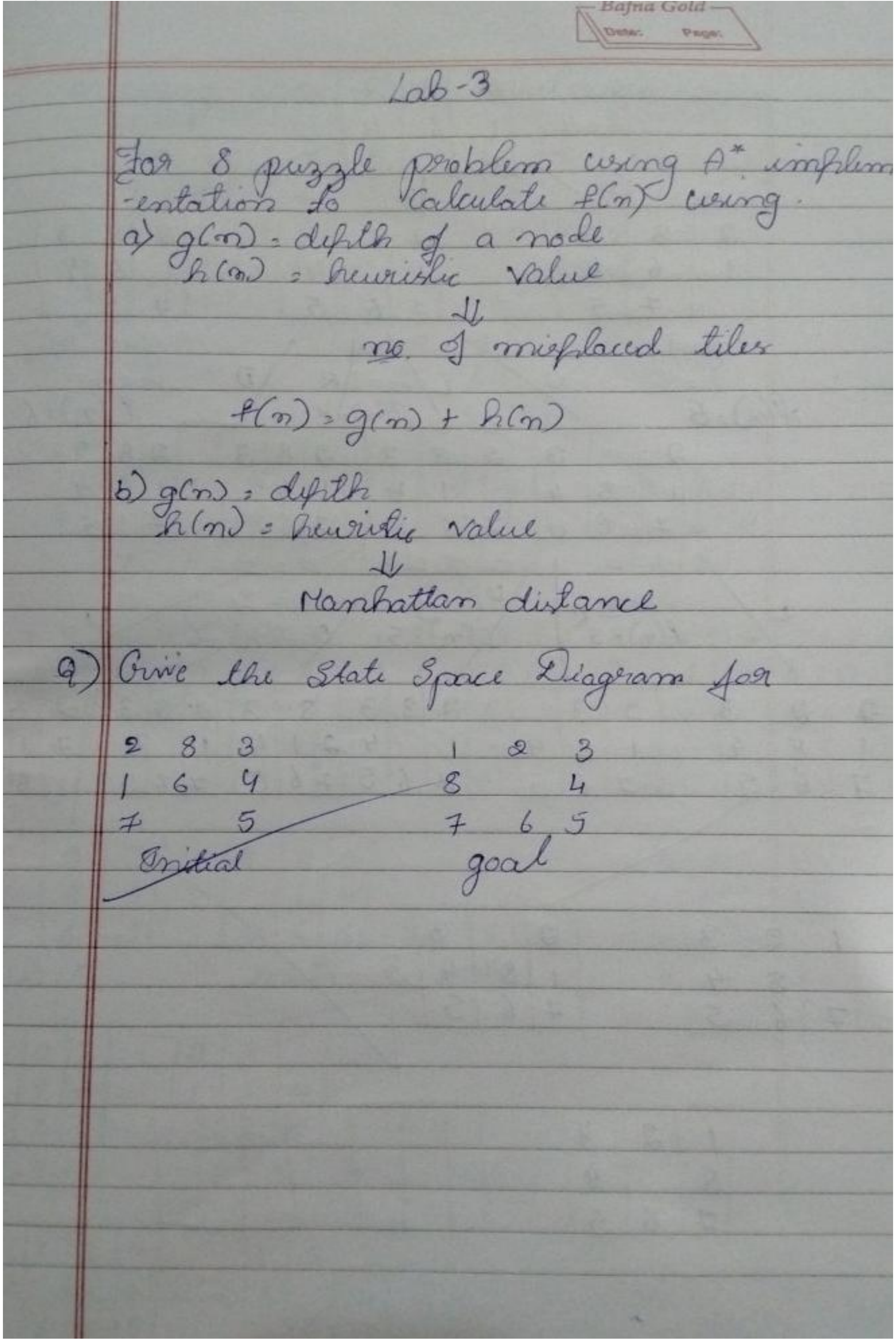


```
Solution found:
[1, 2, 3]
[4, 0, 5]
[7, 8, 6]

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

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

Program 3

Implement A* search algorithm

Algorithm:



Lab-3

For 8 puzzle problem using A^* implem-entation to calculate $f(n)$ using.

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

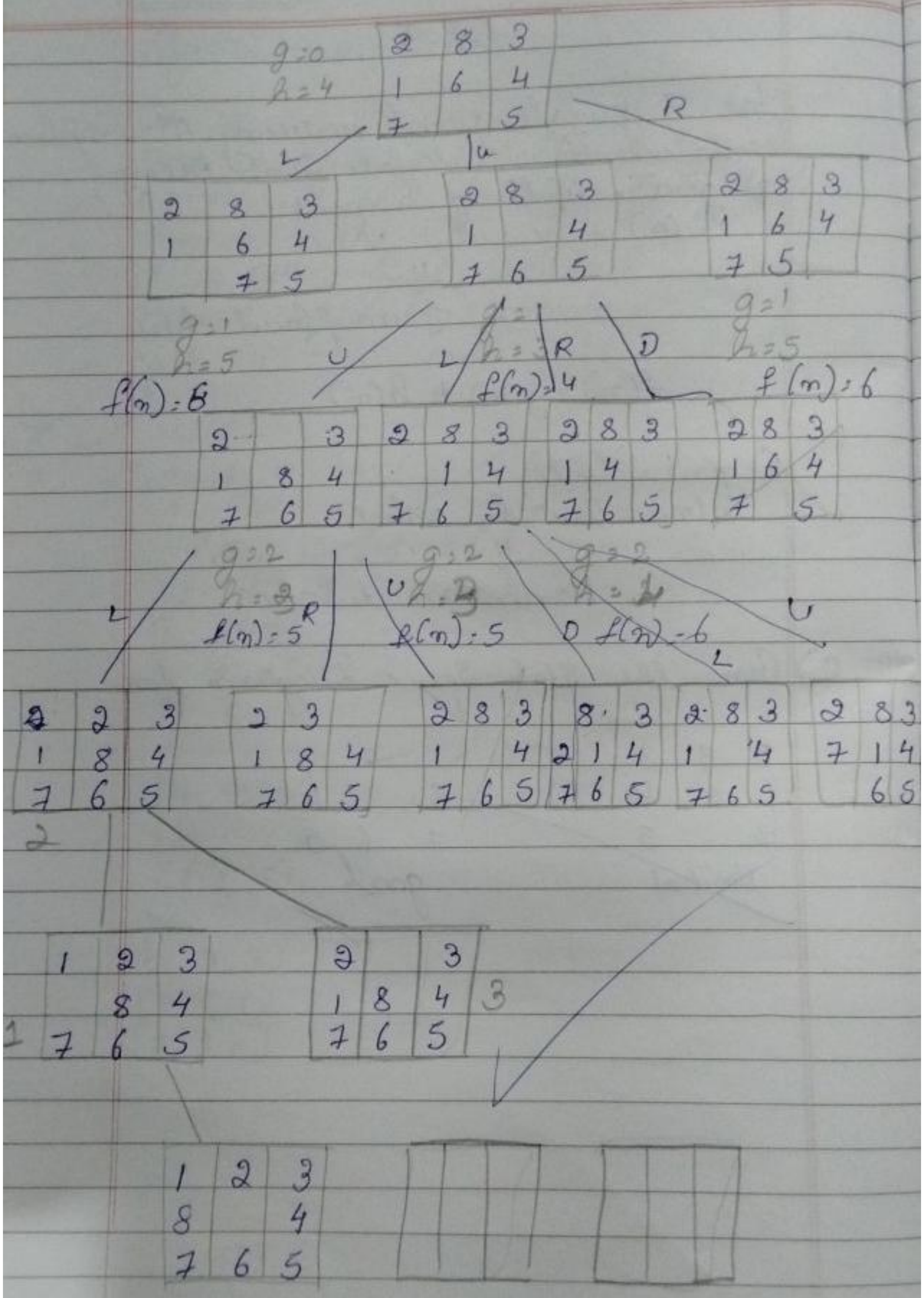
$$f(n) = g(n) + h(n)$$

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

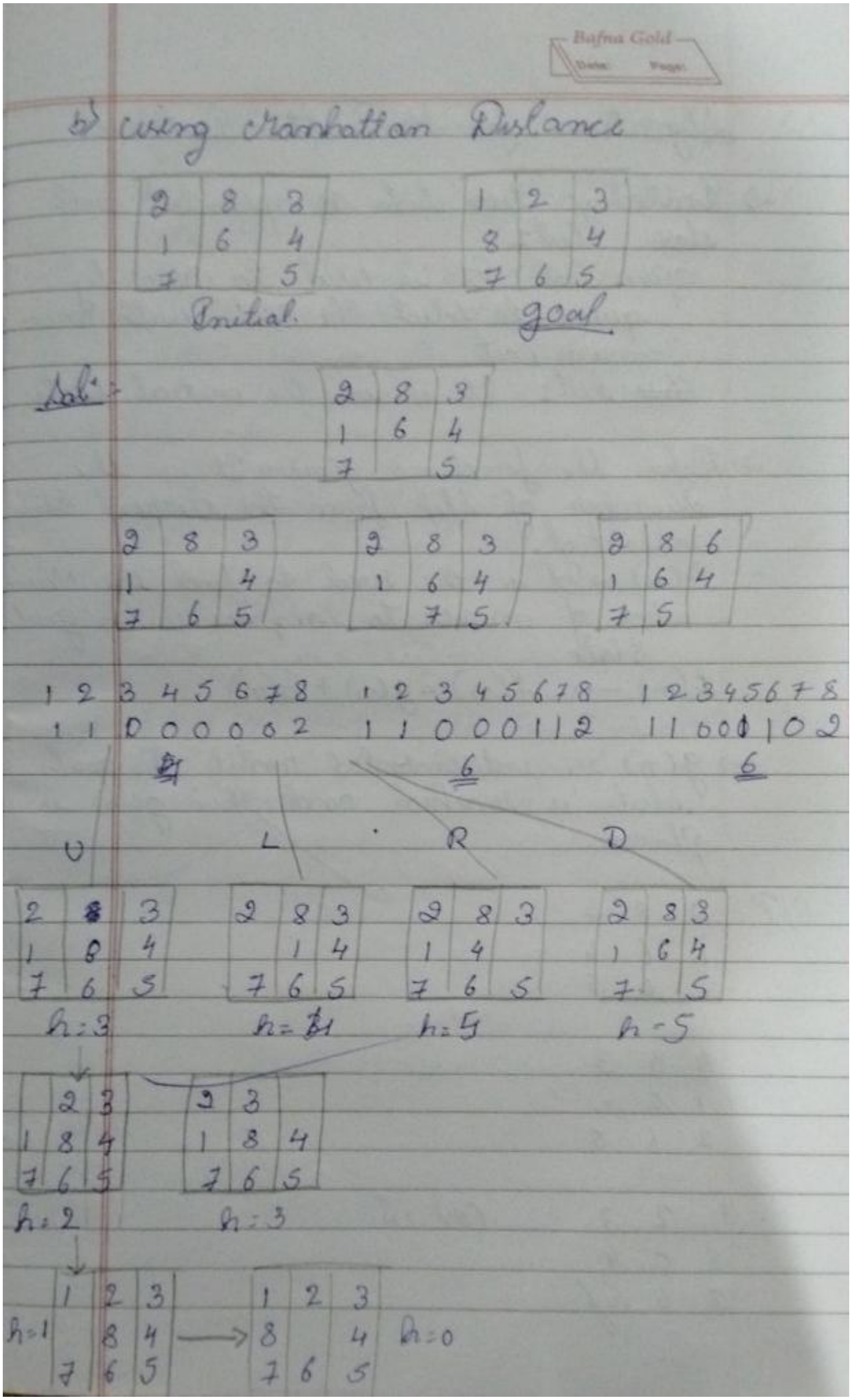
Q) Give the State Space Diagram for

2	8	3		1	2	3
1	6	4		8		4
7		5		7	6	5

Initial   goal



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



b) using Manhattan Distance

2	8	3
1	6	4
7		5

Initial.

1	2	3
8		4
7	6	5

goal

Sol:-

2	8	3
1	6	4
7		5

2	8	3
1		4
7	6	5

1	2	3	4	5	6	7	8
1	1	0	0	0	0	0	2

4

2	8	3
1	6	4
	7	5

1	2	3	4	5	6	7	8
1	1	0	0	0	1	1	2

6

2	8	6
1	6	4
7	5	

1	2	3	4	5	6	7	8
1	1	0	0	1	1	0	2

6

U

2	~~8~~	3
1	8	4
7	6	5

h:3

L

2	8	3
	1	4
7	6	5

h= 4

R

2	8	3
1	4	
7	6	5

h= 5

D

2	8	3
1	6	4
7		5

h =5

	2	3
1	8	4
7	6	5

h: 2

2	3	
1	8	4
7	6	5

h: 3

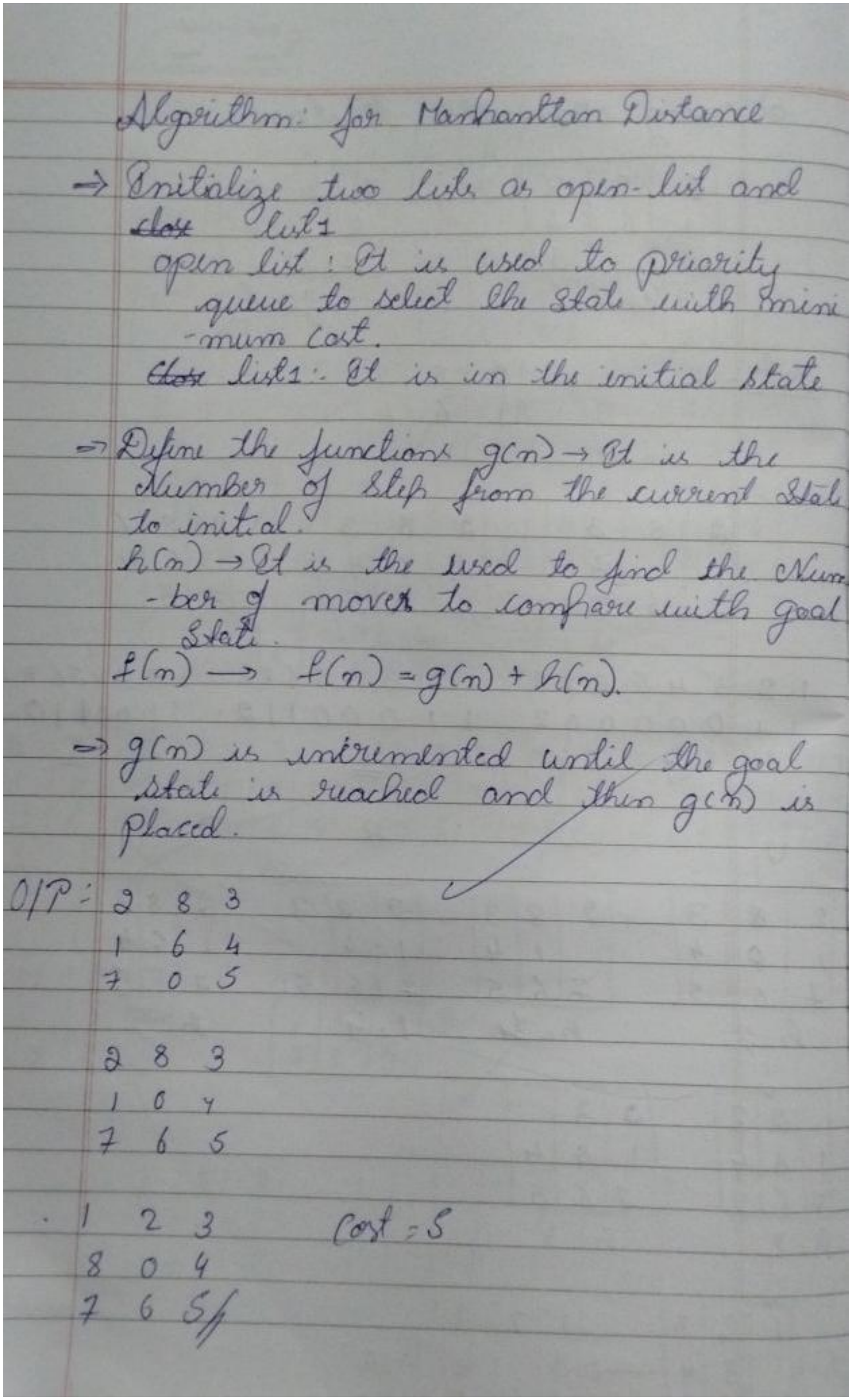
h=1

1	2	3
	8	4
7	6	5

→

1	2	3
8		4
7	6	5

h=0



Algorithm: for Manhanttan Distance

→ Initialize two lists as open-list and ~~close~~ list1

open list : It is used to priority queue to select the state with minimum cost.

~~Close~~ list1 : It is in the initial state

→ Define the functions g(n) → It is the Number of step from the current State to initial.

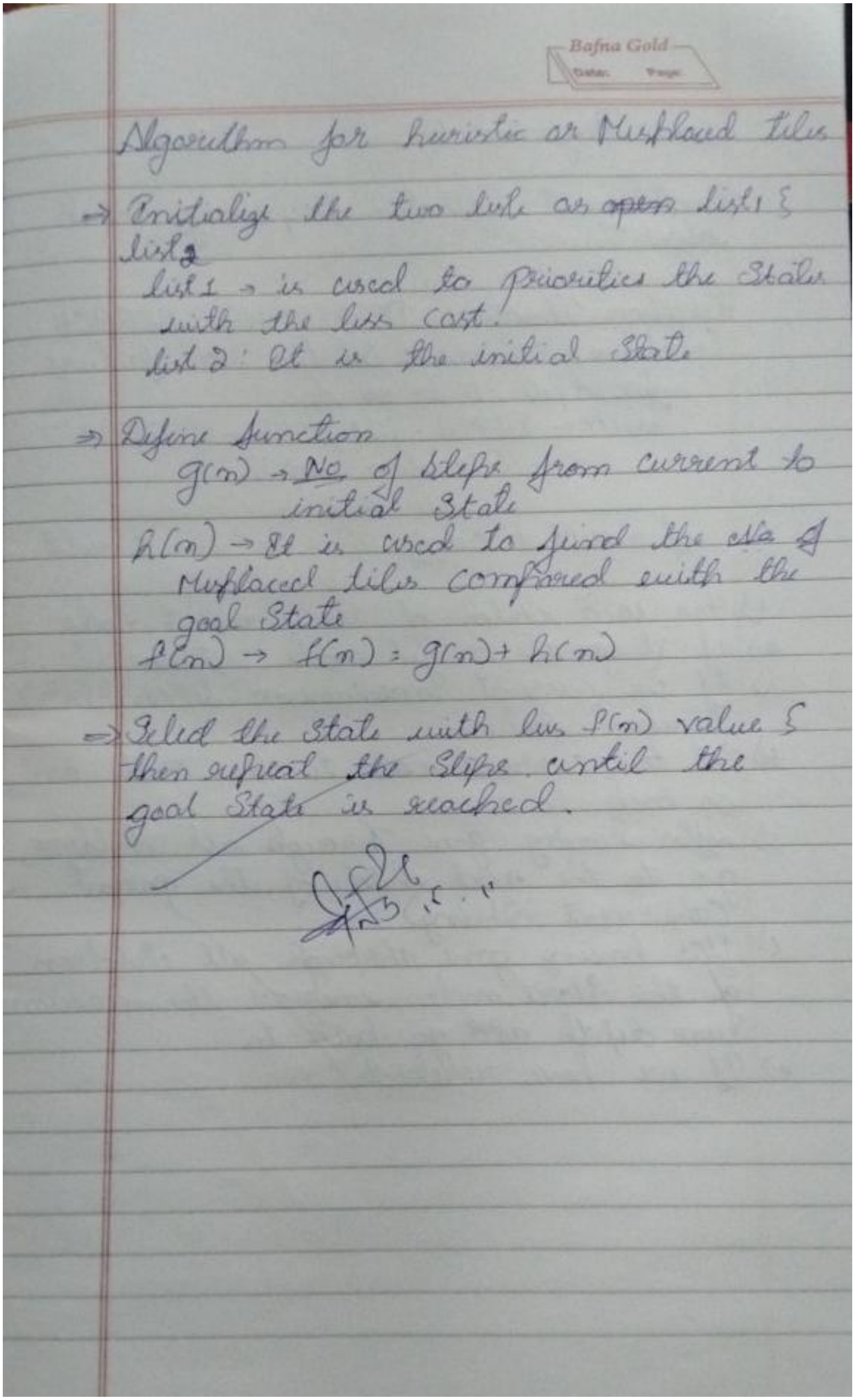
h(n) → It is the used to find the Number of moves to compare with goal State.

f(n) → f(n) = g(n) + h(n).

→ g(n) is incremented until the goal state is reached and then g(n) is placed.

O/P :

```
2 8 3
1 6 4
7 0 5

2 8 3
1 0 4
7 6 5

1 2 3      cost = 5
8 0 4
7 6 5
```



Algorithm for heuristic or Misplaced tiles

→ Initialize the two lists as open list1 & list2

list1 → is used to prioritize the States with the less cost.

list 2: It is the initial State

⇒ Define function

$g(n)$ → No of steps from current to initial State

$h(n)$ → It is used to find the no of Misplaced tiles compared with the goal State

$f(n) \rightarrow f(n) = g(n) + h(n)$

⇒ Select the State with less $f(n)$ value & then repeat the steps until the goal State is reached.

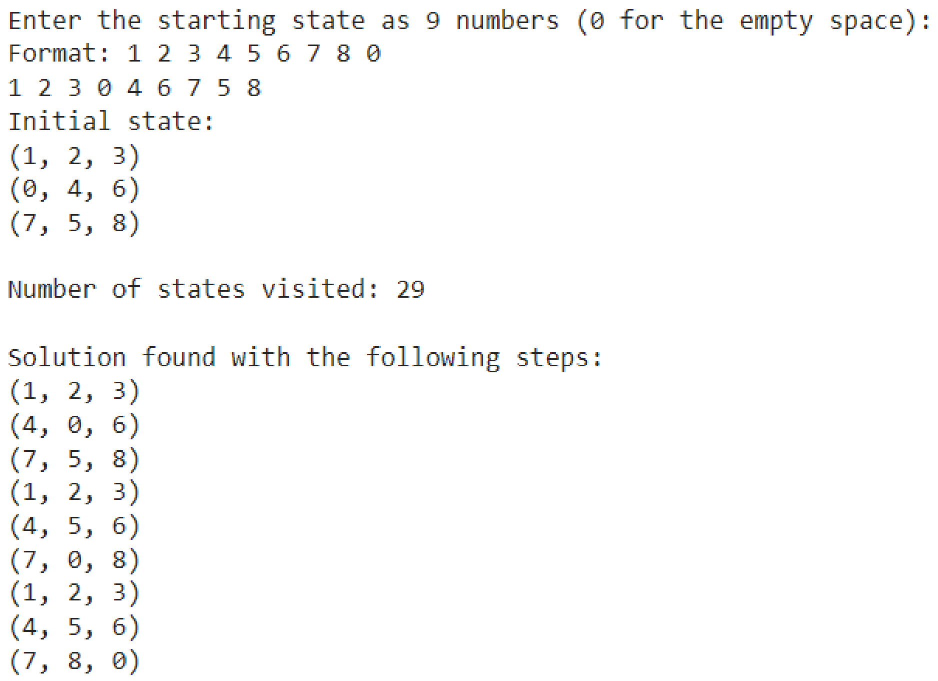
CODE :

```
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:
queue.append((neighbor, path + [neighbor]))
visited.add(neighbor)
visited_count += 1 # Increment visited countreturn None, visited_count # Return count if no solution found
def input_start_state():
while True:
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")
try:
numbers = list(map(int, input_state.split()))
if len(numbers) != 9 or set(numbers) != set(range(9)):
raise ValueError
return tuple(numbers)
except ValueError:
print("Invalid input. Please enter numbers from 0 to 8 with no duplicates.")
def print_matrix(state):
for i in range(0, 9, 3):
print(state[i:i+3])
if __name__ == "__main__": # Corrected main check
initial_state = input_start_state()
print("Initial state:")
print_matrix(initial_state)
print()
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)

else:

print("\nNo solution found.")
```

OUTPUT:

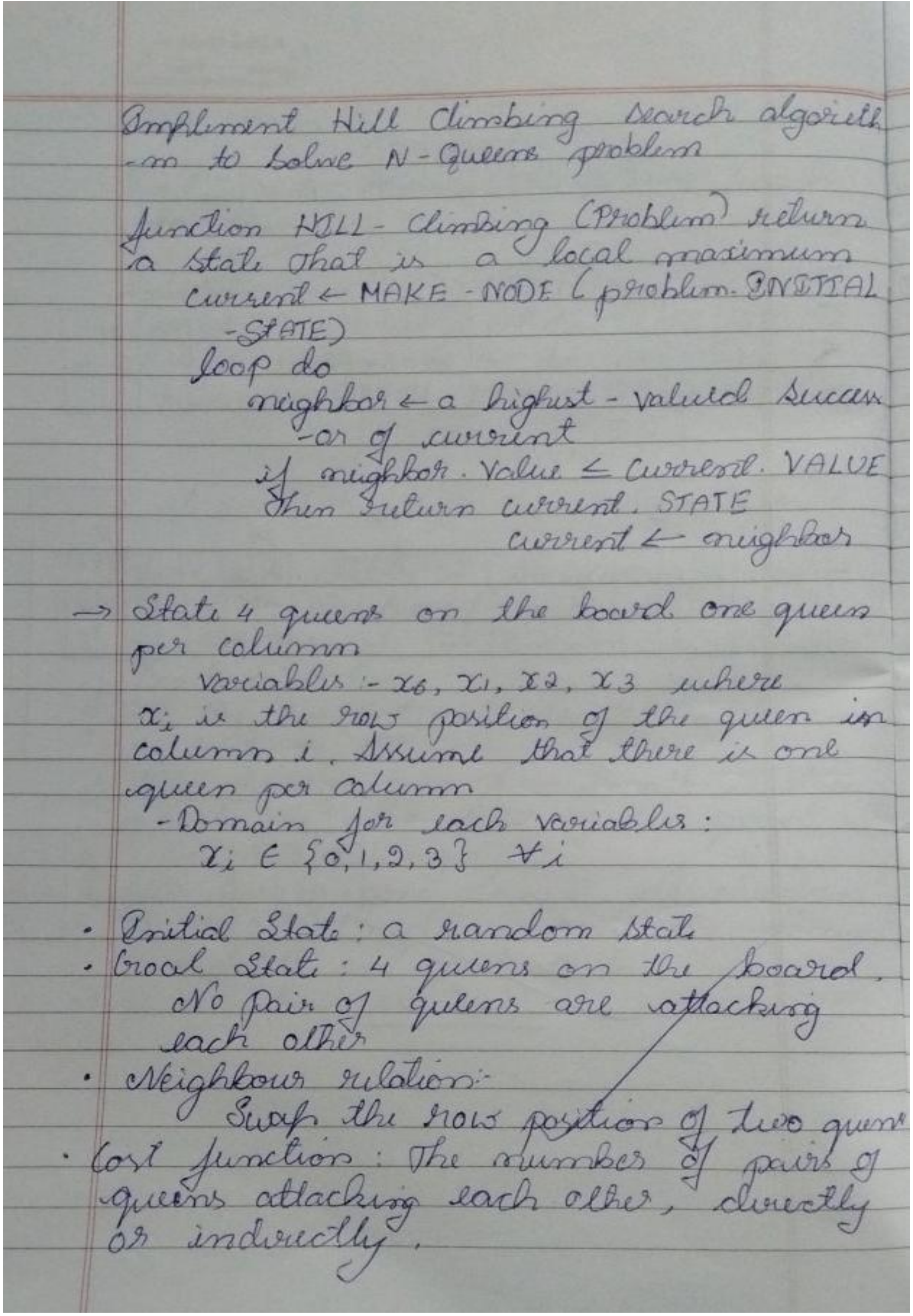


```
Enter the starting state as 9 numbers (0 for the empty space):
Format: 1 2 3 4 5 6 7 8 0
1 2 3 0 4 6 7 5 8
Initial state:
(1, 2, 3)
(0, 4, 6)
(7, 5, 8)

Number of states visited: 29

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

Program 4

Implement Hill Climbing search algorithm to
solve N-Queens problem

Algorithm:



Implement Hill Climbing search algorithm to solve N-Queens problem

```
function HILL-Climbing (Problem) return
a state that is a local maximum
   current ← MAKE-NODE (problem. INITIAL
     -STATE)
   loop do
      neighbor ← a highest-valued successor
        of current
      if neighbor. Value ≤ current. VALUE
      then return current. STATE
                  current ← neighbor
```

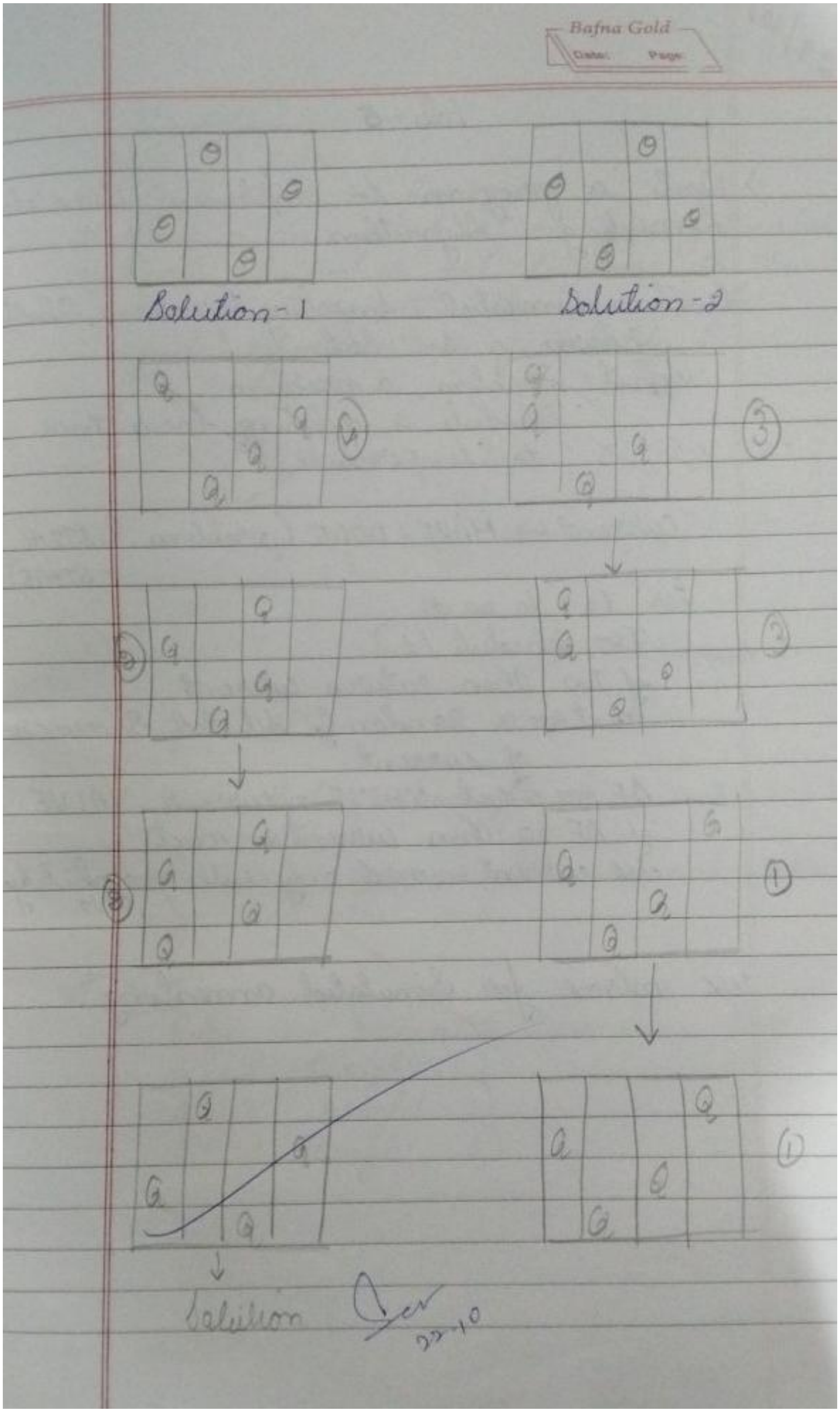
→ State 4 queens on the board one queens per column

Variables :- x_0, x_1, x_2, x_3 where x_i is the row position of the queen in column i. Assume that there is one queen per column

-Domain for each variables:

$x_i \in \{0,1,2,3\} \ \forall i$

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



Bafna Gold
Date: Page:

	Q		
			Q
Q			
		Q	

Solution-1

		Q	
Q			
			Q
	Q		

Solution-2

4

Q			
			Q
		Q	
	Q		

3

Q			
Q			
		Q	
	Q		

2

		Q	
Q			
		Q	
	Q		

2

Q			
Q			
		Q	
	Q		

3

		Q	
Q			
		Q	
Q			

1

			Q
Q			
		Q	
	Q		

	Q		
			Q
Q			
		Q	

Solution

22-10

1

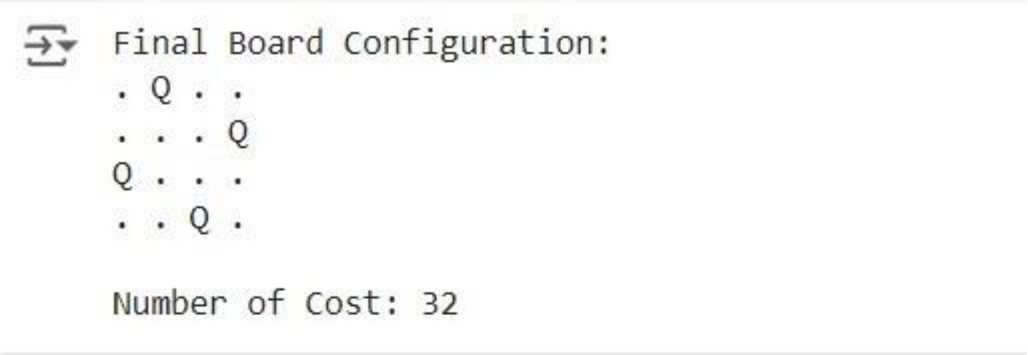
			Q
Q			
		Q	
	Q		

CODE :

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

def hill_climbing(n):
cost=0 while True:
# 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: current_board = neighbor_board
current_conflicts = neighbor_conflicts cost+=1
found_better = True break
if found_better:
break


# If no better neighbor 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

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()
```

```
# Example Usage
n = 4
solution, conflicts, cost = hill_climbing(n) print("Final Board Configuration:")
print_board(solution) print("Number of Cost:", cost)
```

OUTPUT:

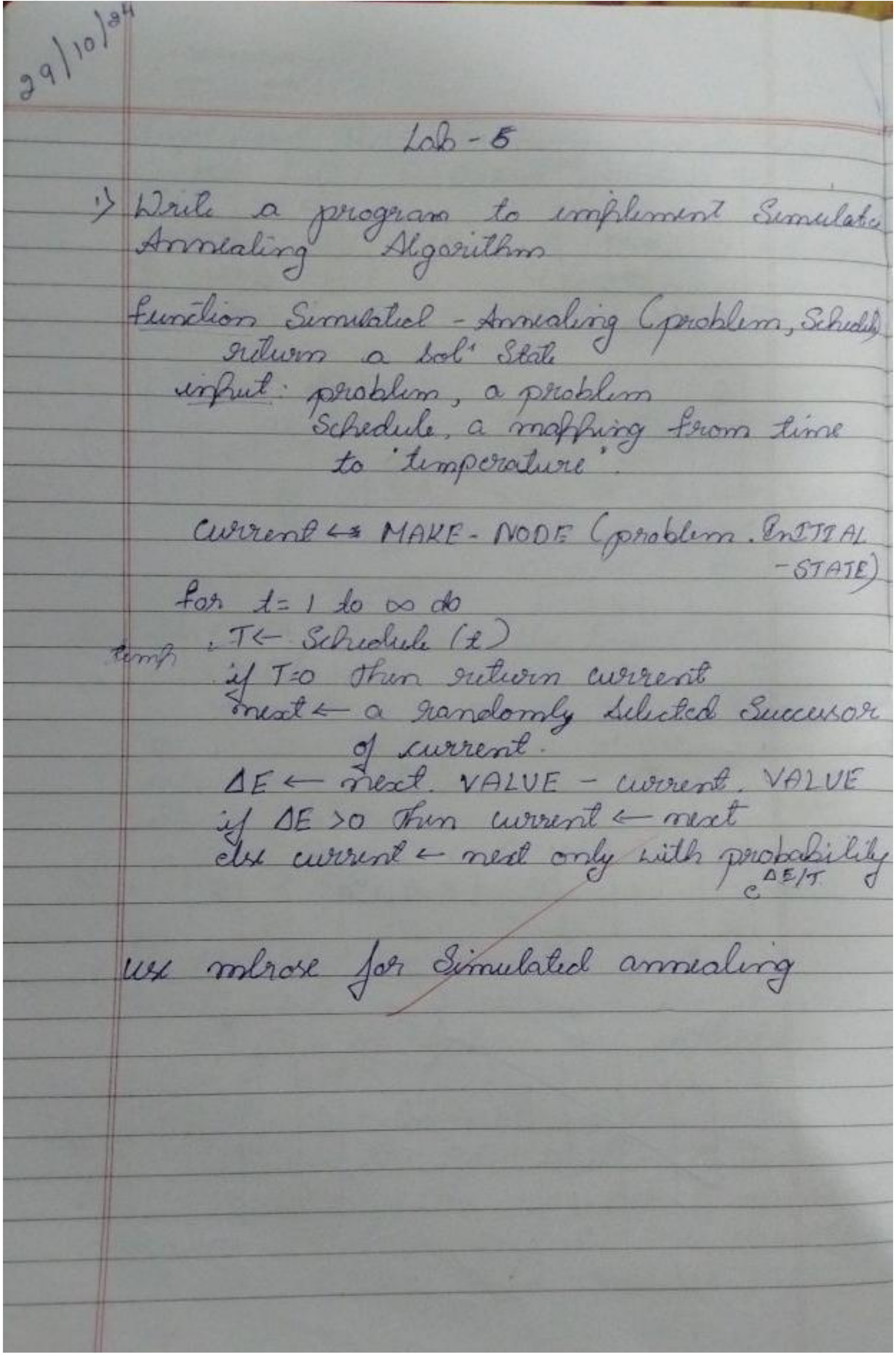


```
Final Board Configuration:
. Q . .
. . . Q
Q . . .
. . Q .

Number of Cost: 32
```

Program 5

Simulated Annealing to Solve 8-Queens

Algorithm:



29/10/24

Lab - 5

i) Write a program to implement Simulate Annealing Algorithm

```
function Simulated - Annealing (problem, Schedule)
    return a sol' State
  input: problem, a problem
         Schedule, a mapping from time
           to 'temperature'.

  current ← MAKE-NODE (problem. INITIAL
                              -STATE)
  for t = 1 to ∞ do
temp  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 $e^{\Delta E/T}$
```

use mlrose for Simulated annealing

· Algorithm

1) Start at a random point x
2) Choose a new point x_j on a nieghbourhood
3) Decide whether or not to move to the new point x_j. The decision will be made based on the probability fun' $P(x, x_j, T)$

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

4) Reduce T

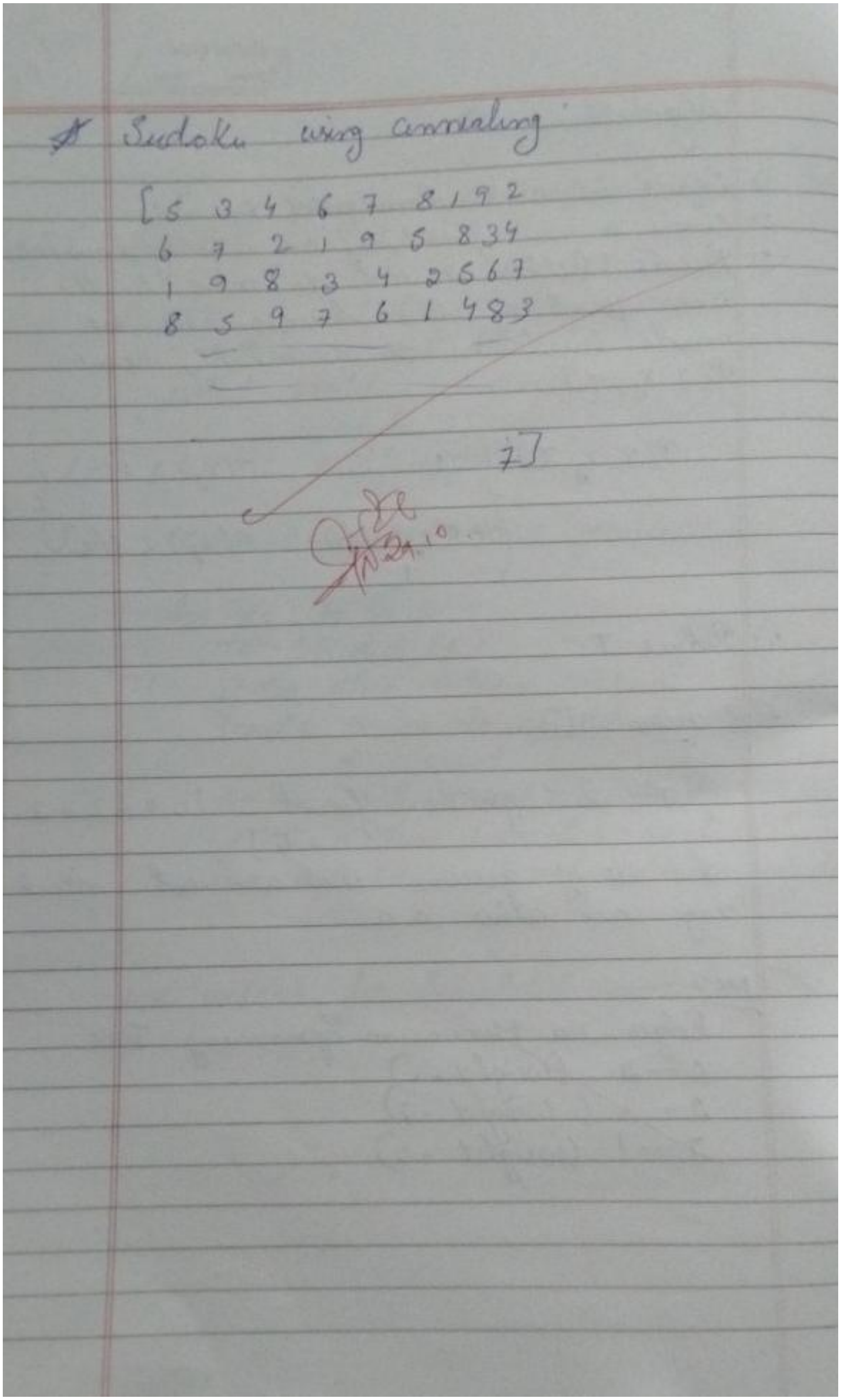
8 queen O/P

The best position found is [0 8 5 2 6 3 7 4]

The No of queens that are not attacking each other is 0

MST

Edges in Minimum Spanning Tree
0 -- 2 (weight = 1)
2 -- 3 (weight = 3)
2 -- 1 (weight = 2)



Sudoku using annealing

```
[5 3 4 6 7 8 1 9 2
 6 7 2 1 9 5 8 3 4
 1 9 8 3 4 2 5 6 7
 8 5 9 7 6 1 4 8 3

                  7]
```

CODE :

Simulated annealing algorithm for 8 puzzle problem

CODE :

```
import numpy as np
from scipy.optimize import dual_annealing

def queens_max(position):
    # This function calculates the number of pairs of queens that are
    # not attacking each other
    position = np.round(position).astype(int)
    n = len(position)
    queen_not_attacking = 0

    for i in range(n - 1):
        no_attack_on_j = 0
        for j in range(i + 1, n):
            if position[i] != position[j] and abs(position[i] - position[j]) != (j - i):
                no_attack_on_j += 1
        if no_attack_on_j == n - 1 - i:
            queen_not_attacking += 1
    if queen_not_attacking == n - 1:
        queen_not_attacking += 1
    return -queen_not_attacking  # Negative because we want to maximize this value

# Bounds for each queen's position (0 to 7 for an 8x8 chessboard)
bounds = [(0, 7) for _ in range(8)]

# Use dual_annealing for simulated annealing optimization
result = dual_annealing(queens_max, bounds)

# Display the results
best_position = np.round(result.x).astype(int)
best_objective = -result.fun

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: [2 6 1 7 4 0 3 5]
The number of queens that are not attacking each other is: 8
```

2)SUDOKU PROBLEM

CODE :

```
import numpy as np
import random
import math

def is_valid(puzzle, row, col, num):
    if num in puzzle[row] or num in puzzle[:, col]:
        return False
    box_x, box_y = row // 3 * 3, col // 3 * 3
    if num in puzzle[box_x:box_x + 3, box_y:box_y + 3]:
        return False
    return True

def initial_fill(puzzle):
    filled = puzzle.copy()
    for row in range(9):
        for col in range(9):
            if filled[row][col] == 0:
                possible_values = [num for num in range(1, 10) if is_valid(filled, row, col, num)]
                if possible_values:
                    filled[row][col] = random.choice(possible_values)
    return filled

def objective(puzzle):
    conflicts = 0
    for row in range(9):
        conflicts += 9 - len(set(puzzle[row]))
    for col in range(9):
        conflicts += 9 - len(set(puzzle[:, col]))
    for box_x in range(0, 9, 3):
        for box_y in range(0, 9, 3):
            box = puzzle[box_x:box_x+3, box_y:box_y+3].flatten()
            conflicts += 9 - len(set(box))
    return conflicts

def simulated_annealing(puzzle, max_iter=100000000, start_temp=1.0, end_temp=0.01,
alpha=0.99):
    current_state = initial_fill(puzzle)
    current_score = objective(current_state)
    temp = start_temp

    for iteration in range(max_iter):
        if current_score == 0:
            break
        row, col = random.randint(0, 8), random.randint(0, 8)
        while puzzle[row][col] != 0:
            row, col = random.randint(0, 8), random.randint(0, 8)

        new_state = current_state.copy()
        new_value = random.randint(1, 9)
        new_state[row][col] = new_value if is_valid(new_state, row, col, new_value) else
current_state[row][col]
        new_score = objective(new_state)
        delta_score = new_score - current_score

        if delta_score < 0 or random.uniform(0, 1) < math.exp(-delta_score / temp):
            current_state, current_score = new_state, new_score

        temp *= alpha

    return current_state

# Example usage:
puzzle = np.array([
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
    [6, 0, 0, 1, 9, 5, 0, 0, 0],
    [0, 9, 8, 0, 0, 0, 0, 6, 0],
    [8, 0, 0, 0, 6, 0, 0, 0, 3],
    [4, 0, 0, 8, 0, 3, 0, 0, 1],
    [7, 0, 0, 0, 2, 0, 0, 0, 6],
    [0, 6, 0, 0, 0, 0, 2, 8, 0],
    [0, 0, 0, 4, 1, 9, 0, 0, 5],
    [0, 0, 0, 0, 8, 0, 0, 7, 9]
])

solved_puzzle = simulated_annealing(puzzle)
print("Solved Sudoku:\n", solved_puzzle)
```

OUTPUT :

```
Solved Sudoku:
[[5 3 1 2 7 6 8 4 0]
 [6 7 4 1 9 5 3 2 0]
 [2 9 8 3 0 0 5 6 7]
 [8 5 9 7 6 1 4 0 3]
 [4 2 6 8 5 3 9 0 1]
 [7 1 3 9 2 4 0 5 6]
 [9 6 5 0 0 7 2 8 4]
 [0 8 2 4 1 9 6 3 5]
 [1 4 0 5 8 2 0 7 9]]
```

3)MST (Minimum Spanning Tree)

CODE :

```
import random
import math
from collections import defaultdict

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

    def add_edge(self, u, v, weight):
        self.edges[u].append((v, weight))
        self.edges[v].append((u, weight))

    def get_edges(self):
        return [(u, v, weight) for u in self.edges for v, weight in self.edges[u] if u < v]

def random_spanning_tree(graph):
    nodes = list(graph.edges.keys())
    random.shuffle(nodes)
    tree_edges = set()
    selected = {nodes[0]}

    while len(selected) < len(nodes):
        u = random.choice(list(selected))
        candidates = [(v, weight) for v, weight in graph.edges[u] if v not in selected]
        if candidates:
            v, weight = random.choice(candidates)
            tree_edges.add((u, v, weight))
            selected.add(v)

    return tree_edges

def energy(tree):
    return sum(weight for u, v, weight in tree)

def generate_neighbor(tree, graph):
    tree_list = list(tree)
    if len(tree_list) < 2:
        return tree
    u, v, weight = random.choice(tree_list)
    new_tree = tree - {(u, v, weight)}

    candidates = [(x, w) for x, w in graph.edges[u] if (x, u, w) not in tree and (u, x, w) not in tree]
    if not candidates:
        return tree

    new_v, new_weight = random.choice(candidates)
    new_tree.add((u, new_v, new_weight))

    return new_tree

def simulated_annealing(graph):
    T = 1.0
    final_temperature = 0.001
    cooling_factor = 0.95
    current_solution = random_spanning_tree(graph)
    best_solution = current_solution

    while T > final_temperature:
        for _ in range(100):
            neighbor = generate_neighbor(current_solution, graph)
            current_energy = energy(current_solution)
            neighbor_energy = energy(neighbor)

            if neighbor_energy < current_energy:
                current_solution = neighbor
            else:
                acceptance_probability = math.exp((current_energy - neighbor_energy) / T)
                if random.random() < acceptance_probability:
                    current_solution = neighbor

            if energy(current_solution) < energy(best_solution):
                best_solution = current_solution

        T *= cooling_factor

    return best_solution

if __name__ == "__main__":
    random.seed(42)
    graph = Graph()
    edges = [(0, 1, 4), (0, 2, 1), (1, 2, 2), (1, 3, 5), (2, 3, 3)]
    for u, v, weight in edges:
        graph.add_edge(u, v, weight)

    mst = simulated_annealing(graph)
    print("Edges in the Minimum Spanning Tree:")
    for u, v, weight in mst:
        print(f"{u} -- {v} (weight: {weight})")
    print("Total weight:", energy(mst))
```

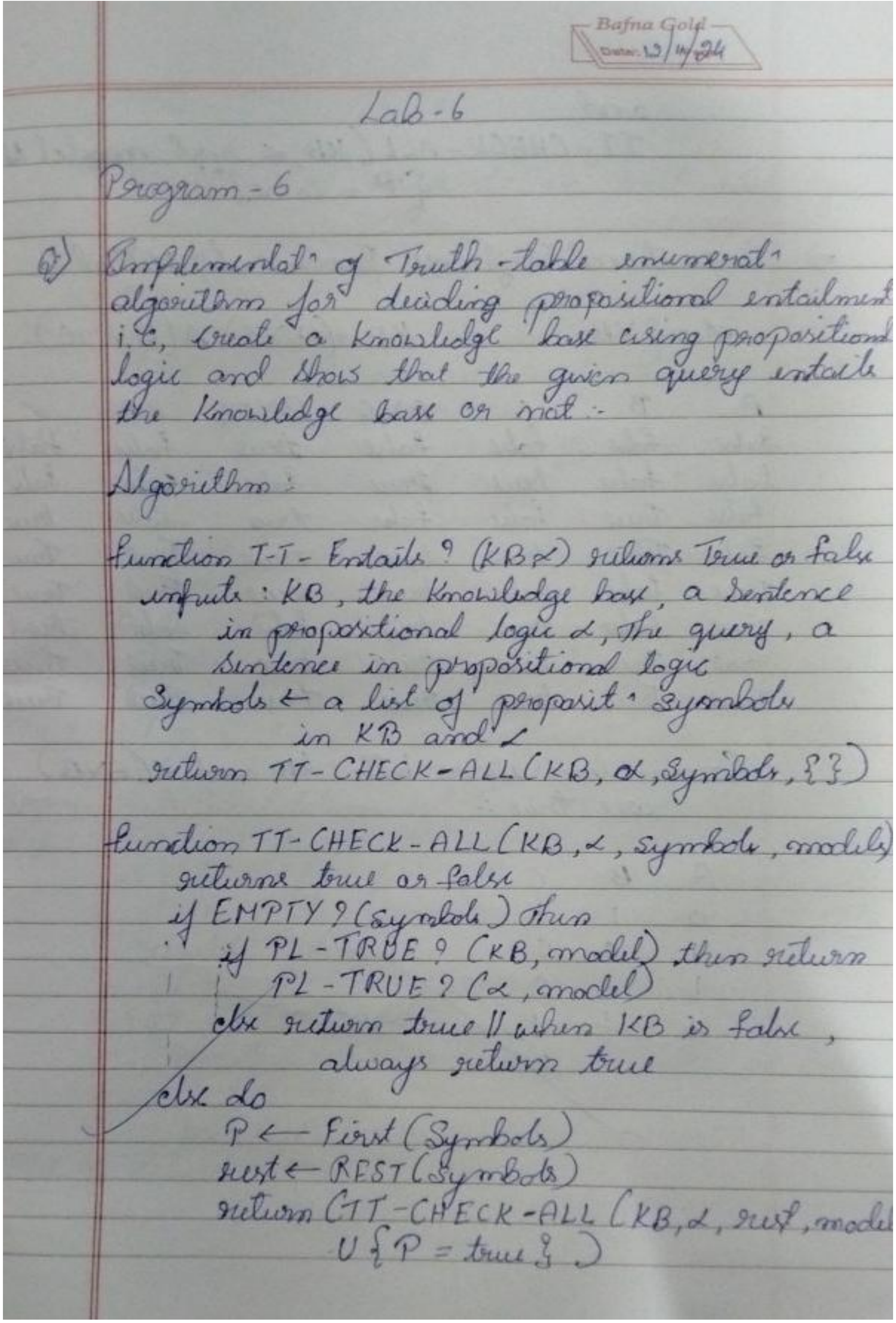
OUTPUT :

```
Edges in the Minimum Spanning Tree:
0 -- 2 (weight: 1)
2 -- 3 (weight: 3)
2 -- 1 (weight: 2)
Total weight: 6
```

Program 6

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

Algorithm:



Bafna Gold

Date: 13/11/24

Lab-6

Program-6

Q) Implementat^n of Truth-table enumerat^n algorithm for deciding propositional entailment i.e, create a knowledge base using propositional logic and shows that the given query entails the Knowledge base or not:-

Algorithm:

```
function T-T-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 proposit^n symbols
            in KB and α
  return TT-CHECK-ALL(KB, α, Symbols, {})

function TT-CHECK-ALL(KB, α, symbols, models)
    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 $\cup$ $\{P = \text{false}\}$))

⇒ Propositional Inference Enumerati^ Method :-

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

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

Combinations where both KB and α(A∨B) are true :-

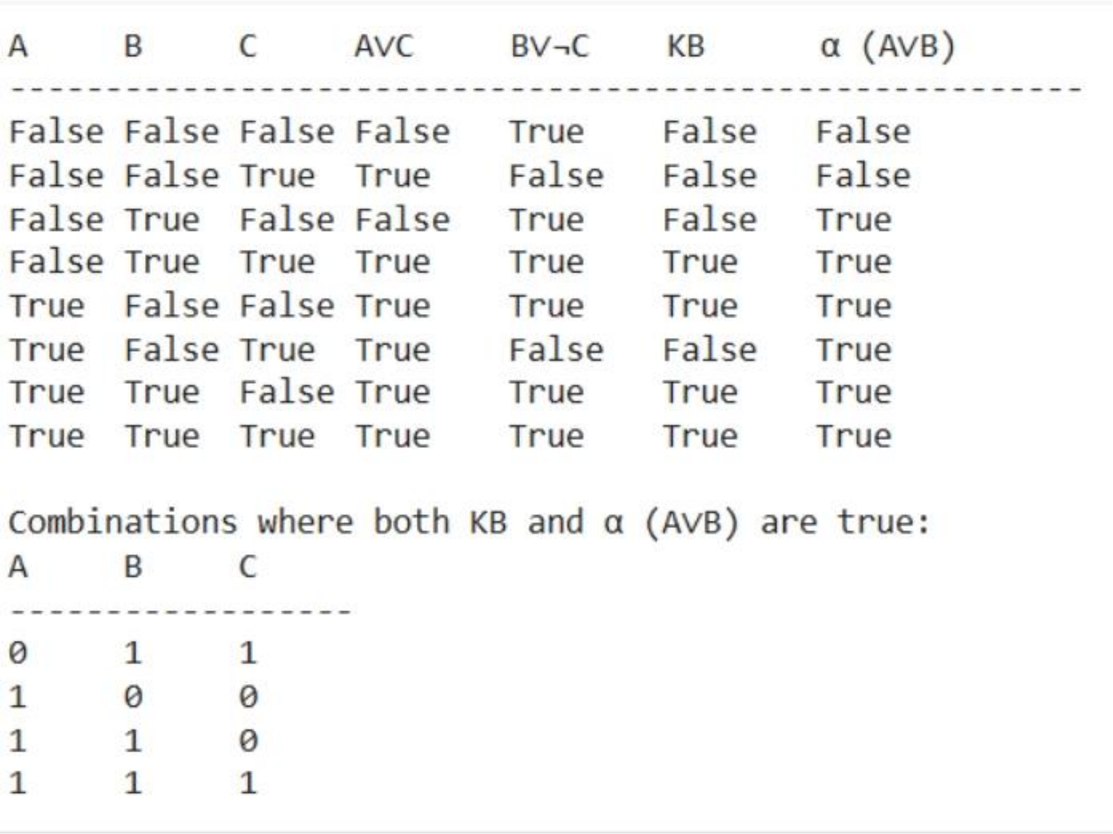
A	B	C
0	1	1
1	0	0
1	1	0
1	1	1

CODE :

```
import itertools
# Define symbols in the KB and query
symbols = ['A', 'B', 'C']
# Define the Knowledge Base (KB) as separate components
A_or_C = lambda A, B, C: A or C
B_or_not_C = lambda A, B, C: B or not C
# Combine the components to define KB
KB = lambda A, B, C: A_or_C(A, B, C) and B_or_not_C(A, B, C)
# Define the Query (alpha)
query = lambda A, B, C: A or B
# Function to print the truth tables
def print_truth_tables(symbols, A_or_C, B_or_not_C, KB, query):
    # Full truth table
    print(f"{'A':<6}{'B':<6}{'C':<6}{'A∨C':<8}{'B∨¬C':<8}{'KB':<8}{'α (A∨B)':<8}")
    print("-" * 56)
    # List to store combinations where both KB and α are true
    both_true = []
    # Generate all possible truth assignments for symbols
    for values in itertools.product([False, True], repeat=len(symbols)):
        # Create a dictionary for the current truth assignment
        assignment = dict(zip(symbols, values))
        # Evaluate each part of the table based on the current assignment
        A_val = assignment['A']
        B_val = assignment['B']
        C_val = assignment['C']
        A_or_C_val = A_or_C(A_val, B_val, C_val)
        B_or_not_C_val = B_or_not_C(A_val, B_val, C_val)
        KB_val = KB(A_val, B_val, C_val)
        query_val = query(A_val, B_val, C_val)
        # Print each row of the truth table
        print(f"{str(A_val):<6}{str(B_val):<6}{str(C_val):<6}"
              f"{str(A_or_C_val):<8}{str(B_or_not_C_val):<8}"
              f"{str(KB_val):<8}{str(query_val):<8}")
        # Store combinations where both KB and α are true
        if KB_val and query_val:
            both_true.append(assignment)
    # Table for combinations where both KB and α are true
    print("\nCombinations where both KB and α (A∨B) are true:")
    print(f"{'A':<6}{'B':<6}{'C':<6}")
    print("-" * 18)
    for assignment in both_true:
        print(f"{assignment['A']:<6}{assignment['B']:<6}{assignment['C']:<6}")
```

```
# Run the function to print the truth tables
print_truth_tables(symbols, A_or_C, B_or_not_C, KB, query)
```

OUTPUT:

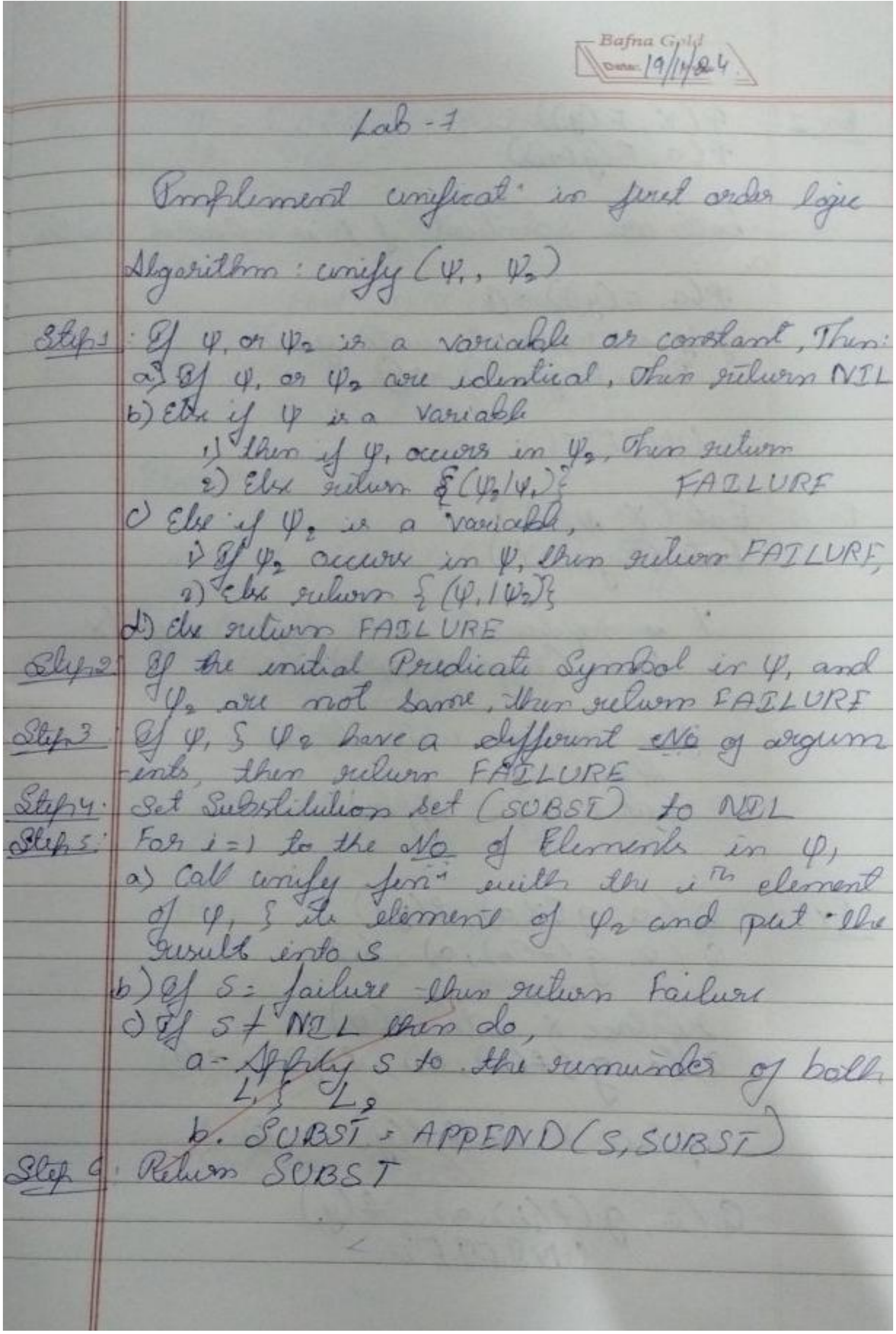


```
A     B     C     A∨C     B∨¬C    KB      α (A∨B)
--------------------------------------------------------
False False False False   True    False   False
False False True  True    False   False   False
False True  False False   True    False   True
False True  True  True    True    True    True
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

Combinations where both KB and α (A∨B) are true:
A     B     C
------------------
0     1     1
1     0     0
1     1     0
1     1     1
```

Program 7

Implement unification in first order logic

Algorithm:



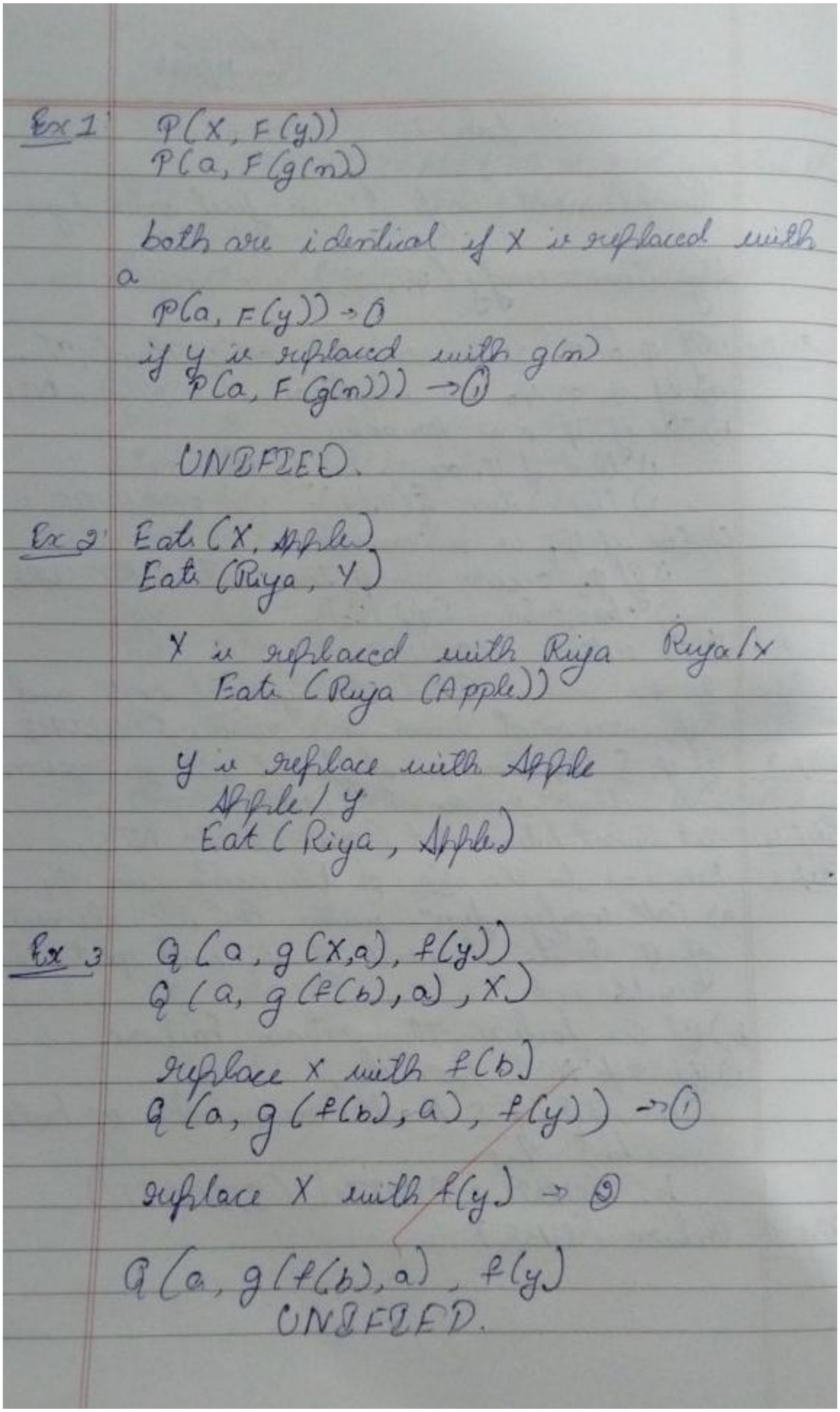
Bafna Gold

Date: 19/11/24

Lab - 7

Implement unificat^n in first order logic

```
Algorithm : unify ($\Psi_1$, $\Psi_2$)

Step 1: If $\Psi_1$ or $\Psi_2$ is a variable or constant, Then:
    a) If $\Psi_1$ or $\Psi_2$ are identical, then return NIL
    b) Else if $\Psi$ is a variable
        1) then if $\Psi_1$ occurs in $\Psi_2$, then return FAILURE
        2) Else return {($\Psi_2$/$\Psi_1$)}
    c) Else if $\Psi_2$ is a variable,
        1) If $\Psi_2$ occurs in $\Psi_1$ then return FAILURE,
        2) Else return {($\Psi_1$/$\Psi_2$)}
    d) Else return FAILURE
Step 2: If the initial Predicate Symbol in $\Psi_1$ and
        $\Psi_2$ are not same, then return FAILURE
Step 3: If $\Psi_1$ & $\Psi_2$ have a different No of arguments, then return FAILURE
Step 4: Set Substitution set (SUBST) to NIL
Step 5: For i=1 to the No of Elements in $\Psi_1$
    a) Call unify fn with the ith element of $\Psi_1$ & ith element of $\Psi_2$ and put the result into S
    b) If S = failure then return failure
    c) If S ≠ NIL then do,
        a. Apply S to the remainder of both
           $L_1$ & $L_2$
        b. SUBST = APPEND(S, SUBST)
Step 6: Return SUBST
```



Ex 1 P(X, F(y))
P(a, F(g(n)))

both are identical if X is replaced with a

P(a, F(y)) → ①
if y is replaced with g(n)
P(a, F(g(n))) → ①

UNIFIED.

Ex 2 Eat (X, Apple)
Eat (Riya, Y)

X is replaced with Riya Riya/x
Eat (Riya (Apple))

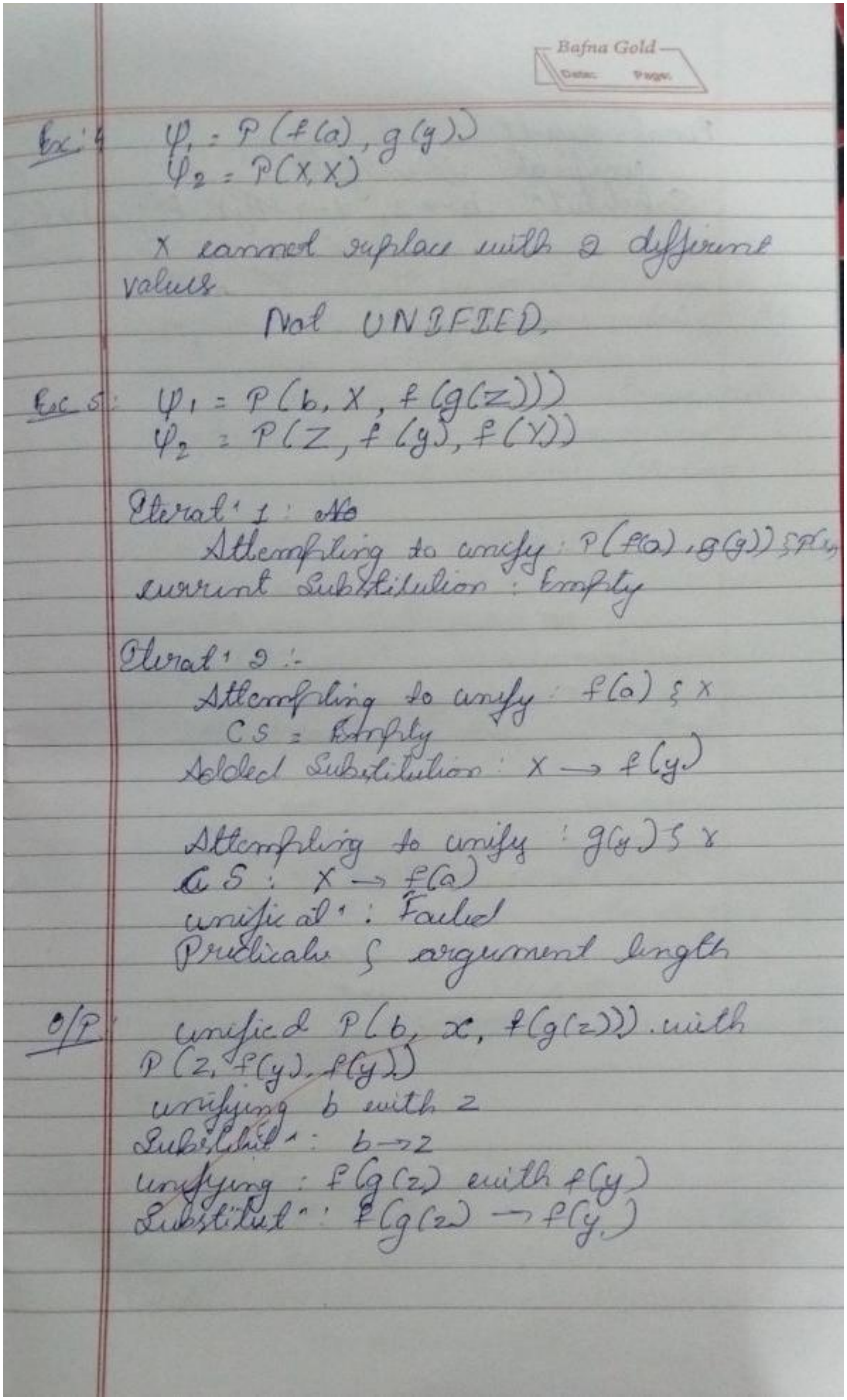
y is replace with Apple
Apple / y
Eat (Riya, Apple)

Ex 3 Q(a, g(X,a), f(y))
Q(a, g(f(b), a), X)

replace X with f(b)
Q(a, g(f(b), a), f(y)) → ①

replace X with f(y) → ②

Q(a, g(f(b), a), f(y)
UNIFIED.



Bafna Gold

Date: Page:

Ex: 4 $\psi_1 = P(f(a), g(y))$
$\psi_2 = P(X, X)$

X cannot replace with 2 different values.

Not UNIFIED.

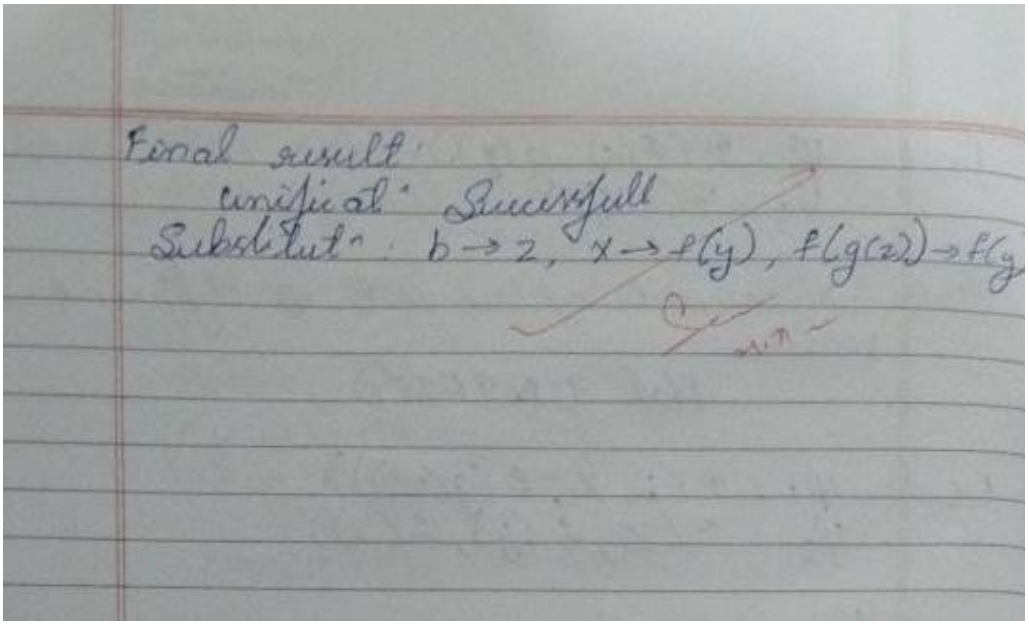
Ex 5: $\psi_1 = P(b, X, f(g(Z)))$
$\psi_2 = P(Z, f(y), f(Y))$

Iterat' 1 : ~~No~~
Attempting to unify : $P(f(a), g(y))$ & $P(x,$
current Substitution : Empty

Iterat' 2 :-
Attempting to unify : $f(a)$ & X
CS = Empty
Added Substitution : $X \rightarrow f(y)$

Attempting to unify : $g(y)$ & X
CS : $X \rightarrow f(a)$
unificat' : Failed
Predicate & argument length

O/P unified $P(b, x, f(g(z)))$ with
$P(z, f(y), f(y))$
unifying b with z
Substitut' : $b \rightarrow z$
unifying : $f(g(z))$ with $f(y)$
Substitut' : $f(g(z)) \rightarrow f(y)$



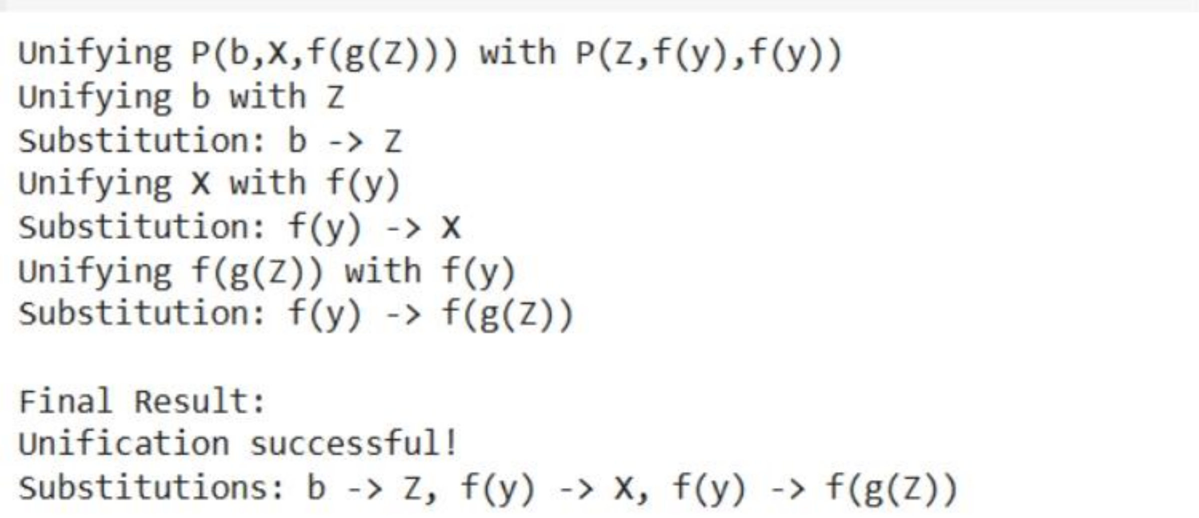
Final result:
unificat: Succesfull
Substitut^n: b → z, x → f(y), f(g(z)) → f(y

CODE :

```
def unify(expr1, expr2):
    print(f"Unifying {expr1} with {expr2}")
    if expr1 == expr2:
        print("Result: Identical terms, no substitution needed.")
        return []  # Return NIL if expressions are identical
    elif is_variable(expr1):
        return failure_if_occurs_check(expr1, expr2)
    elif is_variable(expr2):
        return failure_if_occurs_check(expr2, expr1)
    elif is_compound(expr1) and is_compound(expr2):
        if get_predicate(expr1) != get_predicate(expr2):
            print("Failure: Predicates do not match.")
            return "FAILURE"
        return unify_args(get_arguments(expr1), get_arguments(expr2))
    else:
        print("Failure: Incompatible terms.")
        return "FAILURE"
def unify_args(args1, args2):
    if len(args1) != len(args2):
        print("Failure: Arguments have different lengths.")
        return "FAILURE"
    subst = []
    for a1, a2 in zip(args1, args2):
        s = unify(a1, a2)
        if s == "FAILURE":
            print(f"Failure: Could not unify {a1} with {a2}.")
            return "FAILURE"
        if s:
            subst.extend(s)
            args1 = apply_substitution(s, args1)
            args2 = apply_substitution(s, args2)
    return subst
def is_variable(symbol):
    return isinstance(symbol, str) and symbol.islower()
def is_compound(expression):
    return isinstance(expression, str) and "(" in expression and ")" in expression
def get_predicate(expression):
    return expression.split("(")[0]
def get_arguments(expression):
    args_str = expression[expression.index("(") + 1 : expression.rindex(")")]
    return [arg.strip() for arg in args_str.split(",")]
def failure_if_occurs_check(variable, expression):
    if occurs_check(variable, expression):
        print(f"Failure: Occurs check failed for {variable} in {expression}.")
        return "FAILURE"
    print(f"Substitution: {variable} -> {expression}")
    return [(variable, expression)]
def occurs_check(variable, expression):
    if variable == expression:
        return True
    if is_compound(expression):
        return variable in get_arguments(expression)
    return False
def apply_substitution(subst, expression):
    if isinstance(expression, list):
        return [apply_substitution(subst, sub_expr) for sub_expr in expression]
    elif is_variable(expression):
        for var, value in subst:
            if expression == var:
                return value
    elif is_compound(expression):
        predicate = get_predicate(expression)
        arguments = get_arguments(expression)
        substituted_args = [apply_substitution(subst, arg) for arg in arguments]
        return f"{predicate}({', '.join(substituted_args)})"
    return expression
# Example usage:
expr1 = "P(b,X,f(g(Z)))"
expr2 = "P(Z,f(y),f(y))"
```

```
result = unify(expr1, expr2)
print("\nFinal Result:")
if result == "FAILURE":
    print("Unification failed!")
else:
    print("Unification successful!")
    print("Substitutions:", ', '.join(f"{var} -> {val}" for var, val in result))
```

OUTPUT :

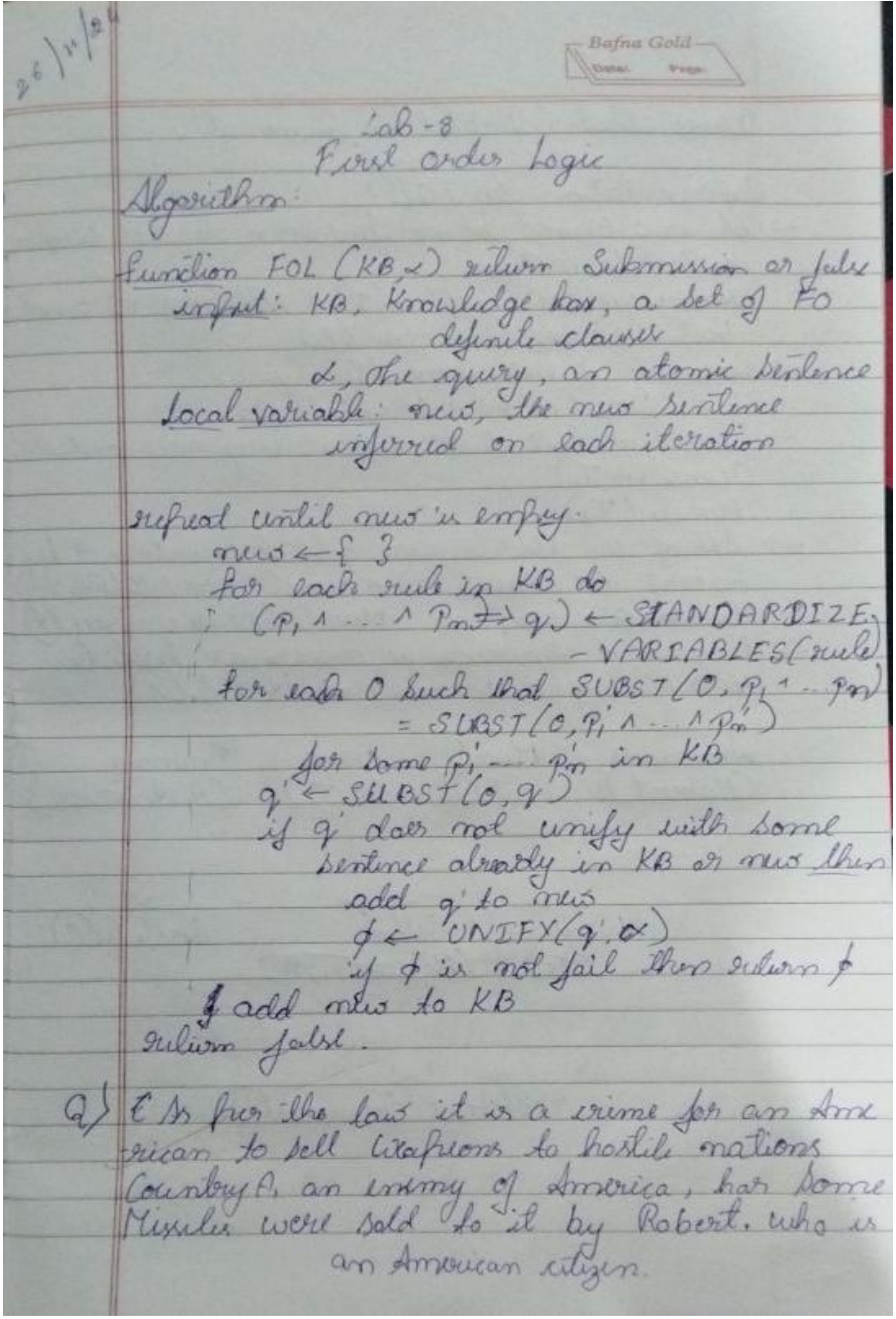


```
Unifying P(b,X,f(g(Z))) with P(Z,f(y),f(y))
Unifying b with Z
Substitution: b -> Z
Unifying X with f(y)
Substitution: f(y) -> X
Unifying f(g(Z)) with f(y)
Substitution: f(y) -> f(g(Z))

Final Result:
Unification successful!
Substitutions: b -> Z, f(y) -> X, f(y) -> f(g(Z))
```

Program 8

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

Algorithm:



26/11/24

Bafna Gold

Lab - 8

First order Logic

Algorithm:

```
function FOL (KB, α) return Submission or false
   input: KB, Knowledge base, a set of FO
                 definite clauses
          α, the query, an atomic sentence
   Local variable: new, the new sentence
          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
              add q' to new
              φ ← UNIFY(q', α)
              if φ is not fail then return φ
     add new to KB
return false.
```

Q) { As per the law it is a crime for an American to sell weapons to hostile nations Country A, an enemy of America, has some Missiles were sold to it by Robert, who is an American citizen.

Prove that "Robert is Criminal".

Representation In FOL:

→ It is a crime for an American to sell weapon to hostile nations
$American(p) \land Weapon(q) \land sells(p, q, r)$ and $Hostile(r) \Rightarrow Criminal(p)$

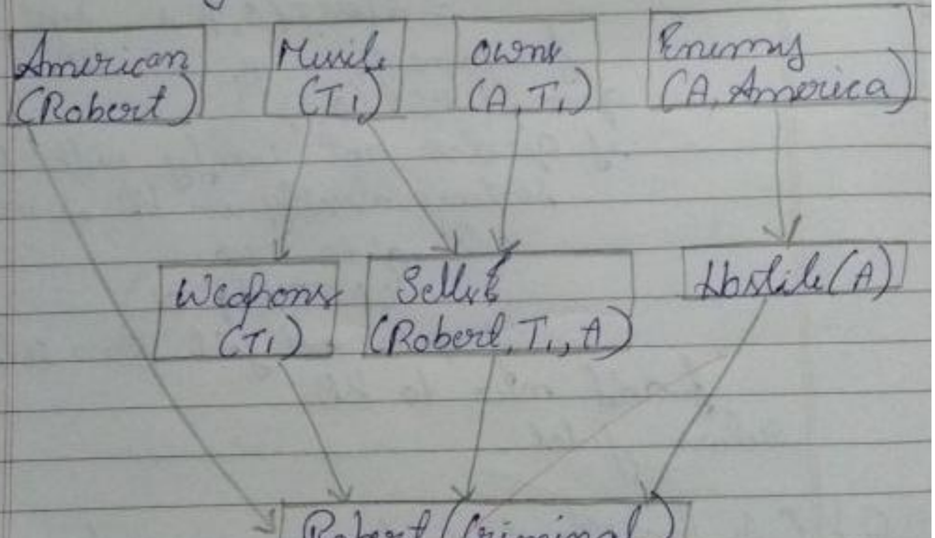
→ Country A has some Missiles.
$\exists x\ Owns(A, x) \land Missiles(x)$.

→ Existial Instantial: Introducing new constant T_1
$Owns(A, T_1)$
$Missile(T_1)$

→ All of the Missiles are sold to country A by Robert $\forall x\ Missile(x) \land owns(A, X) \Rightarrow Sells(Rob, X, A)$

→ Missiles are weapons $Missile(x) \Rightarrow weapon(x)$

⇒ Enemy of America is known as hostile
$\forall x\ Enemy(x, America) \Rightarrow Hostile(x)$.



26/11/24

CODE :

```
class ForwardReasoning:
    def __init__(self, rules, facts):
        """
        Initializes the ForwardReasoning system.
        Parameters:
            rules (list): List of rules as tuples (condition, result),
                          where 'condition' is a set of facts.
            facts (set): Set of initial known facts.
        """
        self.rules = rules  # List of rules (condition -> result)
        self.facts = set(facts)  # Known facts
    def infer(self, query):
        """
        Applies forward reasoning to infer new facts based on rules and initial facts.
        Parameters:
            query (str): The fact to verify if it can be inferred.
        Returns:
            bool: True if the query can be inferred, False otherwise.
        """
        applied_rules = Tru
        while applied_rules:
            applied_rules = False
            for condition, result in self.rules:
                if condition.issubset(self.facts) and result not in self.facts:
                    self.facts.add(result)
                    applied_rules = True
                    print(f"Applied rule: {condition} -> {result}")
                    # If the query is inferred, return True immediately
                    if query in self.facts:
                        return True
        return query in self.fact
# Define the Knowledge Base (KB) with rules as (condition, result)
rules = [
    ({"American(Robert)"}, "Sells(Robert, m1, CountryA)"),  # Based on Owns(CountryA, m) ^ Missile(m)
    ({"Sells(Robert, m1, CountryA)", "American(Robert)", "Hostile(CountryA)"}, "Criminal(Robert)"),  # Final inference
]
# Define initial facts
facts = {
    "American(Robert)",
    "Hostile(CountryA)",
    "Missile(m1)",
    "Owns(CountryA, m1)",
}
# Query
alpha = "Criminal(Robert)"
# Initialize and run forward reasoning
reasoner = ForwardReasoning(rules, facts)
result = reasoner.infer(alpha
print("\nFinal facts:")
print(reasoner.facts)
print(f"\nQuery '{alpha}' inferred: {result}
```

OUTPUT:

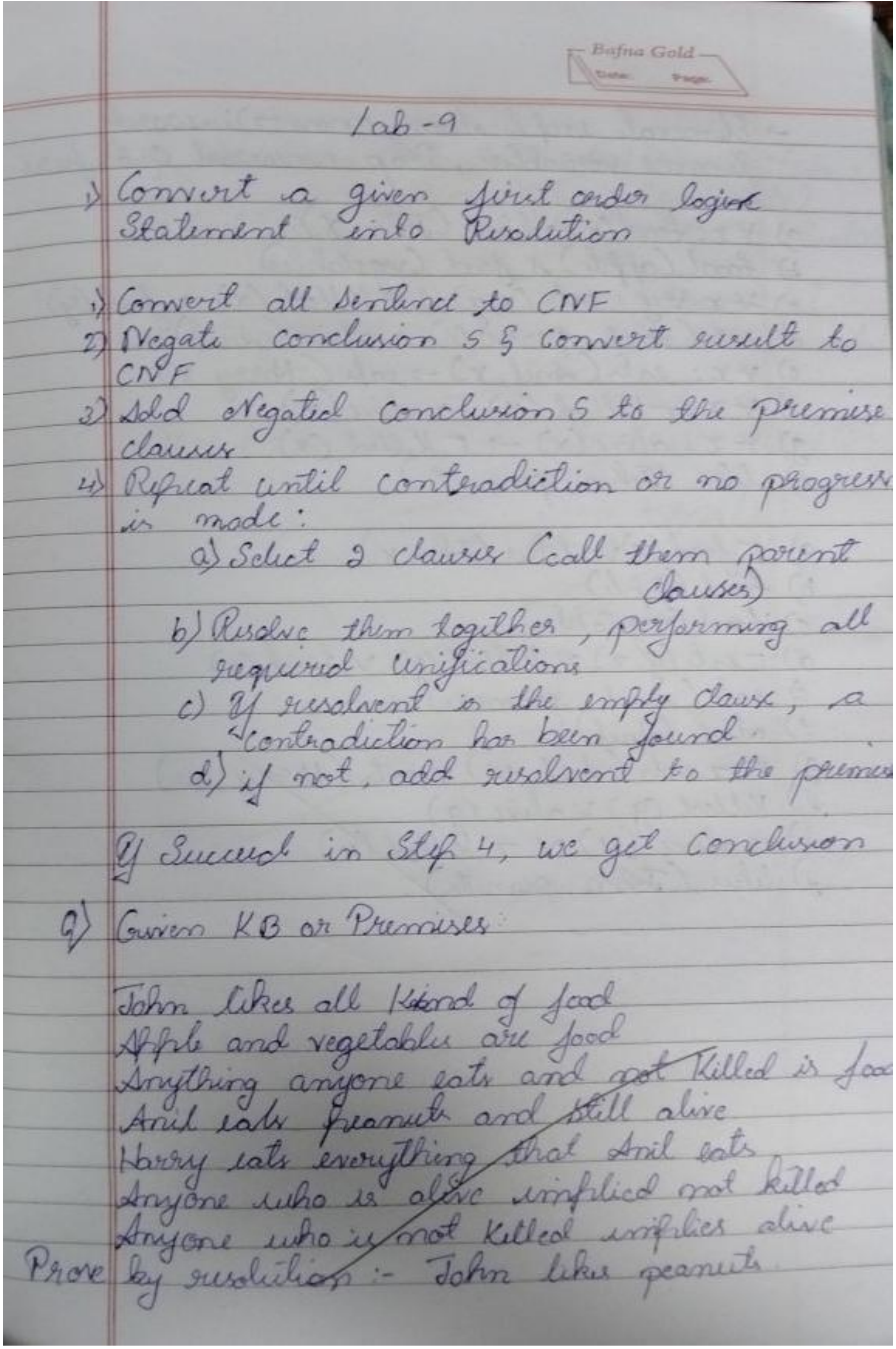
```
Applied rule: {'American(Robert)'} -> Sells(Robert, m1, CountryA)

Applied rule: {'Hostile(CountryA)', 'Sells(Robert, m1, CountryA)', 'American(Robert)'} -> Criminal(Robert)

Final facts:
```

```
{'Criminal(Robert)', 'Hostile(CountryA)', 'Sells(Robert, m1, CountryA)', 'American(Robert)',
'Owns(CountryA, m1)', 'Missile(m1)'}


Query 'Criminal(Robert)' inferred: True
```

Program 9

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

Algorithm:



Lab-9

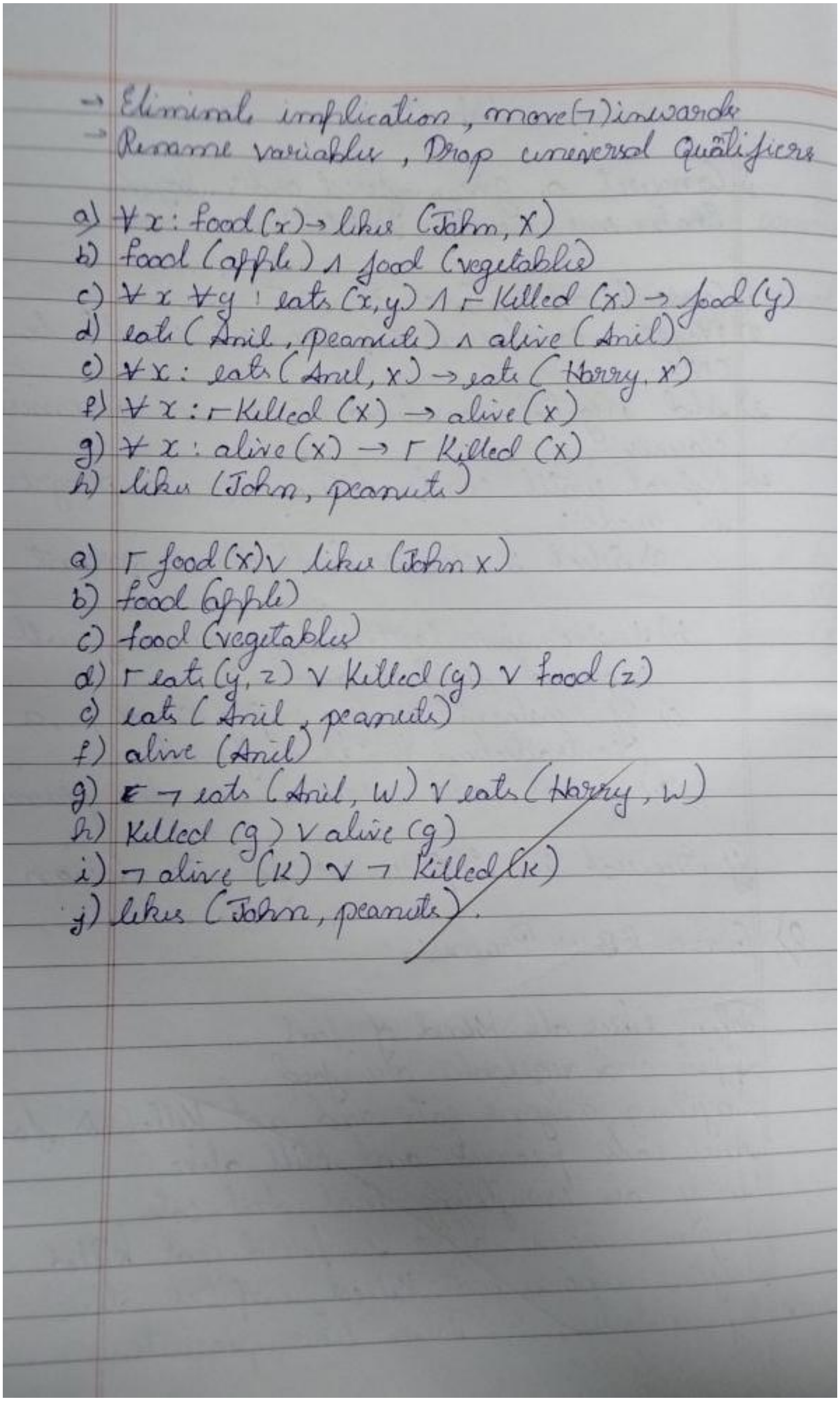
1) Convert a given first order logic Statement into Resolution

1) Convert all sentence to CNF
2) Negate conclusion S & convert result to CNF
3) Add Negated conclusion S to the premise 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 clause, a contradiction has been found.
   d) if not, add resolvent to the premis

If Succeed in Step 4, we get conclusion

Q) Given KB or Premises

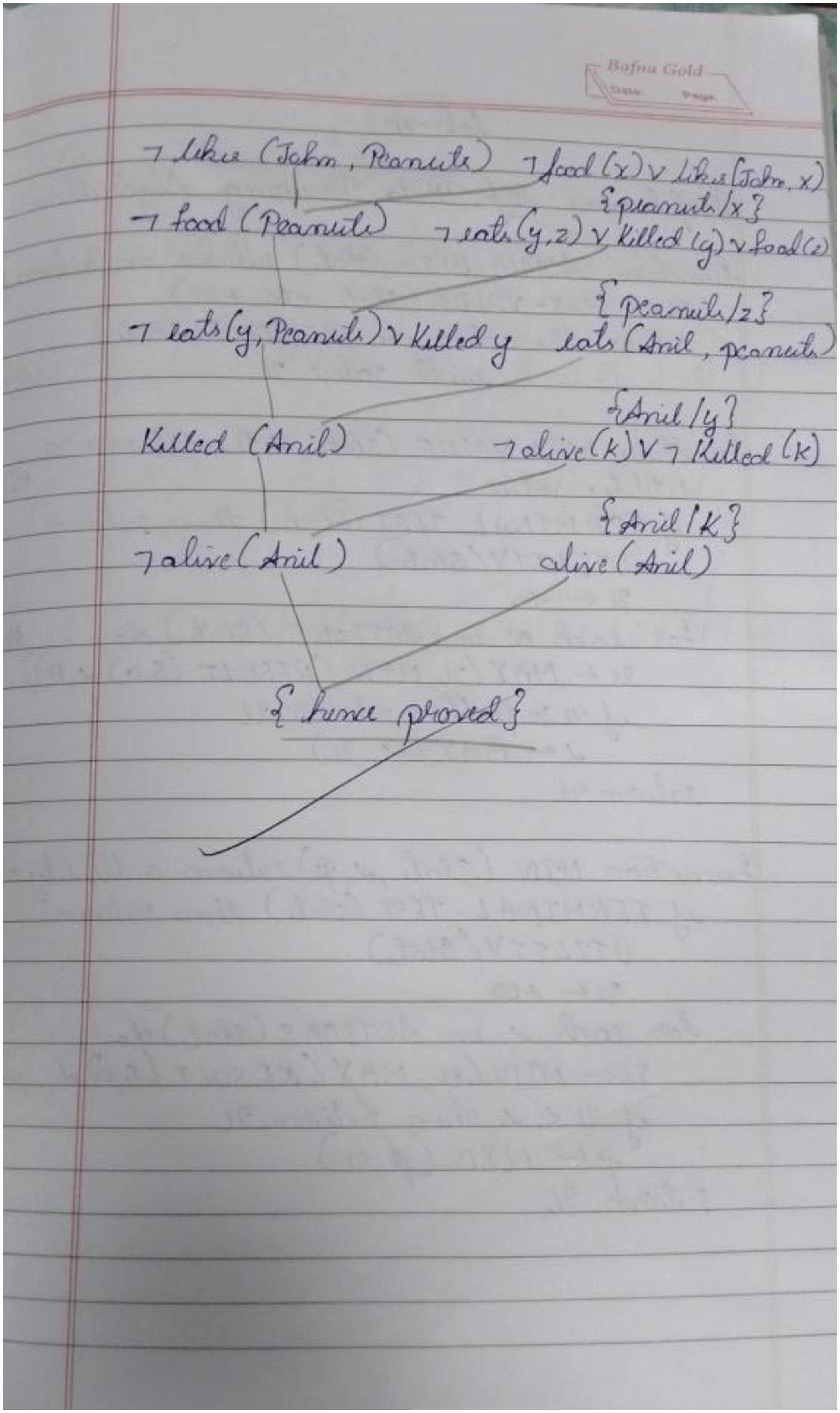
John likes all kind of food
Apple and vegetables are food
Anything anyone eats and not killed is foo
Anil eats peanuts and still alive
Harry eats everything that Anil eats
Anyone who is alive implies not killed
Anyone who is not killed implies alive
Prove by resolution :- John likes peanuts



→ Eliminal implication, move(¬)inwards
→ Rename variables, Drop universal Qualifiers

a) $\forall x : food(x) \rightarrow likes(John, x)$
b) $food(apple) \land food(vegetables)$
c) $\forall x \forall y : eats(x, y) \land \neg Killed(x) \rightarrow food(y)$
d) $eats(Anil, peanuts) \land alive(Anil)$
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) $likes(John, peanuts)$

a) $\neg food(x) \lor likes(John\ x)$
b) $food(apple)$
c) $food(vegetables)$
d) $\neg eats(y, z) \lor Killed(y) \lor food(z)$
e) $eats(Anil, peanuts)$
f) $alive(Anil)$
g) $\neg eats(Anil, w) \lor eats(Harry, w)$
h) $Killed(g) \lor alive(g)$
i) $\neg alive(k) \lor \neg Killed(k)$
j) $likes(John, peanuts)$.



¬likes (John, Peanuts)   ¬food (x) ∨ likes (John, x)

{peanuts/x}

¬food (Peanuts)   ¬eats (y, z) ∨ Killed (y) ∨ food (z)

{peanuts/z}

¬eats (y, Peanuts) ∨ Killed y   eats (Anil, peanuts)

{Anil/y}

Killed (Anil)   ¬alive (k) ∨ ¬Killed (k)

{Anil/K}

¬alive (Anil)   alive (Anil)

{hence proved}

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'}")
```

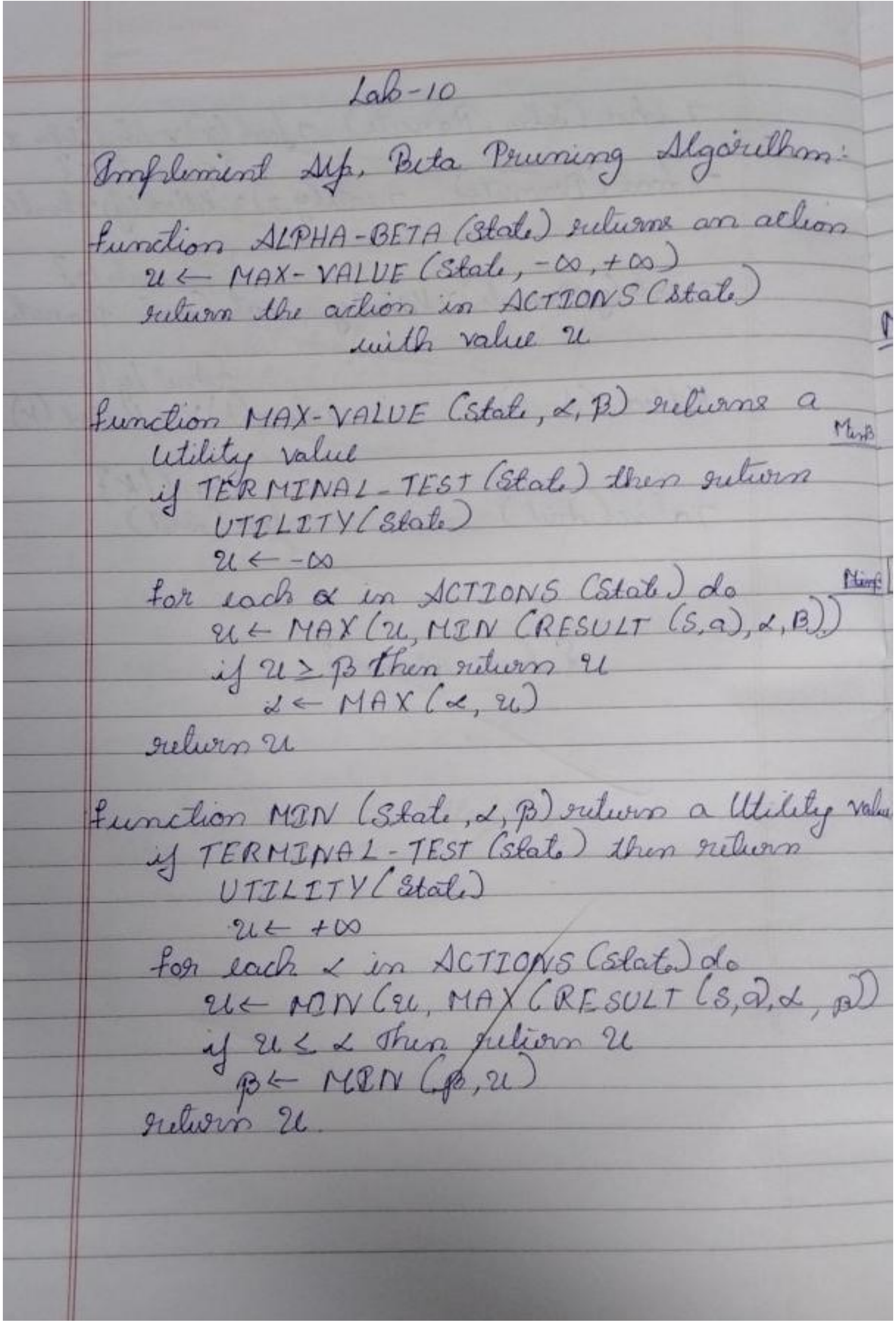
OUTPUT :

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

Program 10

Implement Alpha-Beta Pruning.

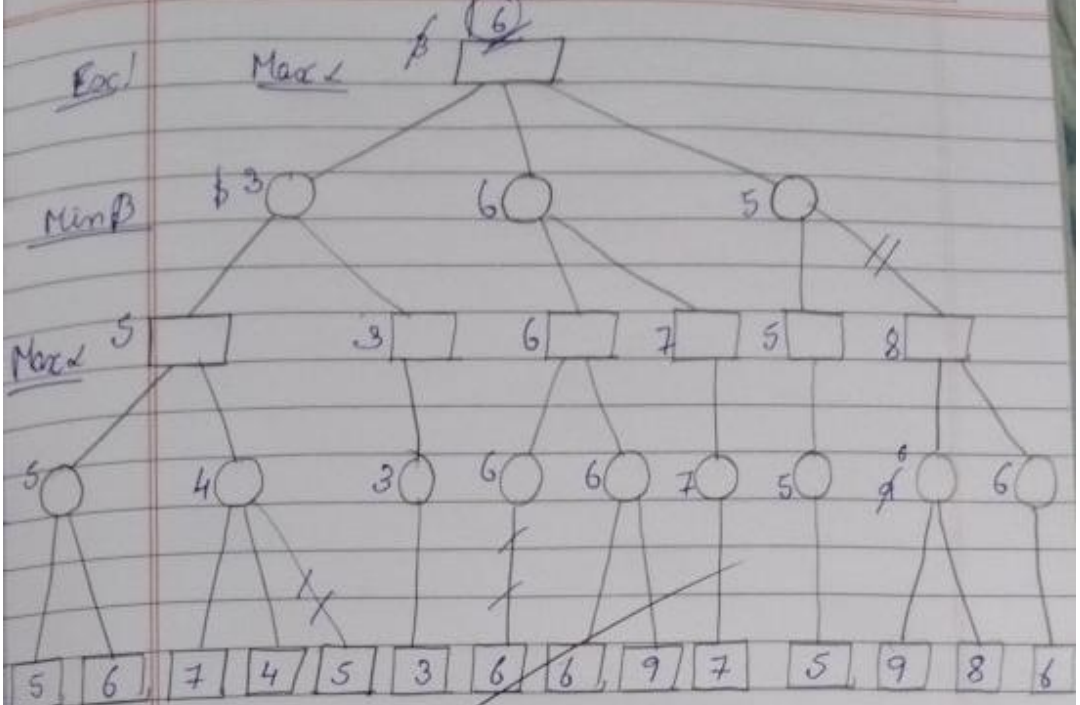
Algorithm:



Lab-10

Impliment Alp. Beta Pruning Algorithm:

```
function ALPHA-BETA (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
        UTILITY (State)
        v ← -∞
    for each α in ACTIONS (State) do
        v ← MAX (v, MIN (RESULT (S, a), α, β))
        if v ≥ β then return v
          α ← MAX (α, v)
    return v

function MIN (State, α, β) return a Utility value
    if TERMINAL-TEST (State) then return
        UTILITY (State)
        v ← +∞
    for each α in ACTIONS (State) do
        v ← MIN (v, MAX (RESULT (S, a), α, β))
        if v ≤ α then return v
          β ← MIN (β, v)
    return v.
```


Ex:
Max α
Min β
Max α
6
β
β 3
6
5
5
3
6
7
5
8
5
4
3
6
6
7
5
6
~~9~~
6
5
6
7
4
5
3
6
6
9
7
5
9
8
6

08.12

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
```