VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitte

d by

Shilpa K M(1BM23CS419)

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 **Shilpa K M(1BM23CS419),** 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.

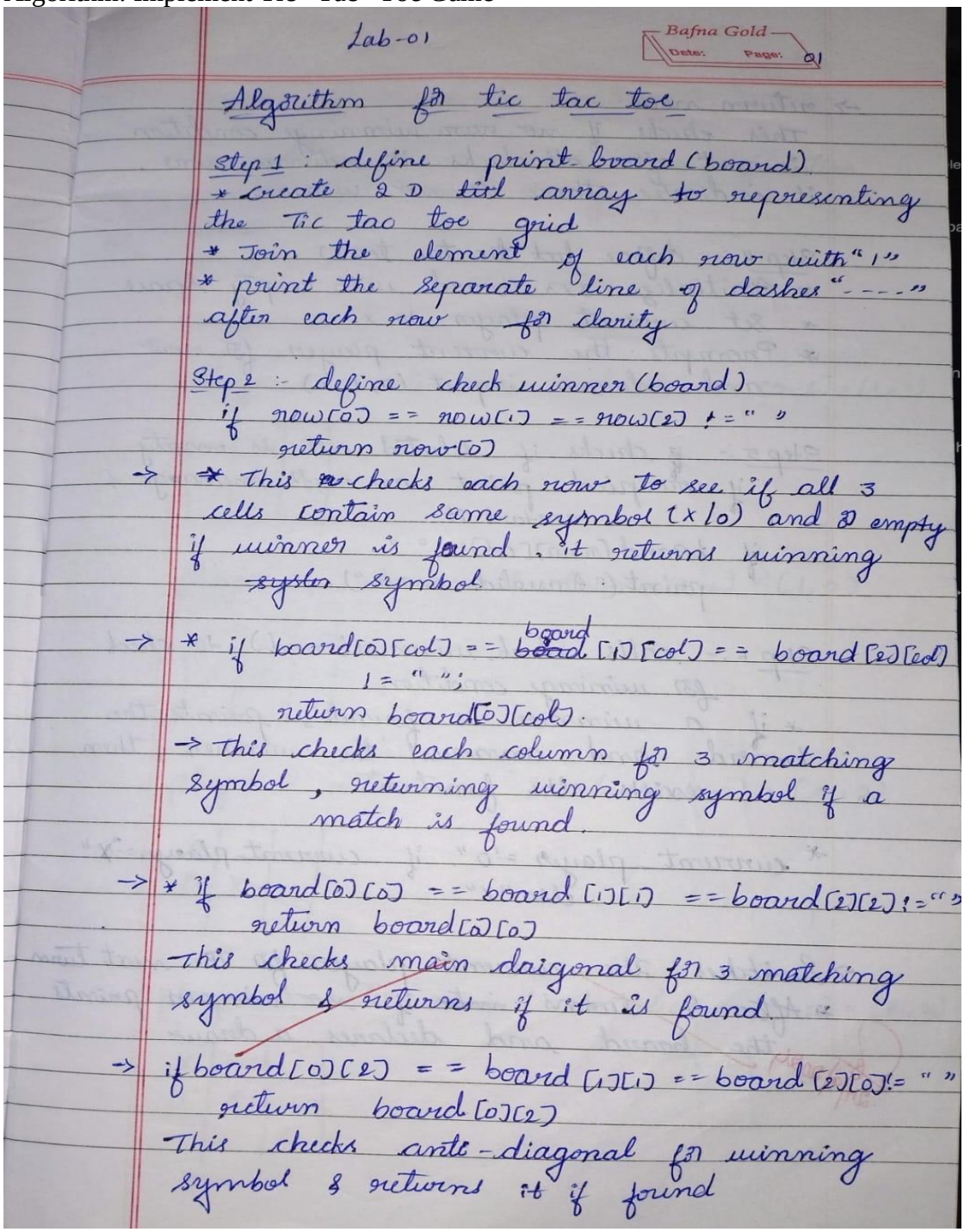
Sunayana S Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/SHILPA-45/AI.git

Program 1
Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm: Implement Tic –Tac –Toe Game



lab-01

Bafna Gold
Date: Page: 01

Algorithm for tic tac toe

Step 1 : define print board (board)
* create 2 D list array to representing the Tic tac toe grid
* Join the element of each row with "|"
* print the separate line of dashes "---" after each row for clarity

Step 2 :- define check winner (board)
if row[0] == row[1] == row[2] != " "
return row[0]
→ * This checks each row to see if all 3 cells contain same symbol (x/o) and 2 empty if winner is found, it returns winning symbol.

→ * if board[0][col] == board[1][col] == board[2][col] != " ";
return board[0][col].
→ this checks each column for 3 matching symbol, returning winning symbol if a match is found.

→ * if board[0][0] == board[1][1] == board[2][2] != " "
return board[0][0]
-this checks main daigonal for 3 matching symbol & returns if it is found.

→ if board[0][2] == board[1][1] == board[2][0] != " "
return board[0][2]
This checks anti-diagonal for winning symbol & returns it if found

→ return none
This checks if no winning condition is met after all checks it returns none. it indicates there is no winner.

Step 4 :- define tic tac toe ()
* Initialize 3×3 board with empty Space
* Set current player to x
* Prompts the current players for row and column input (0, 1, 2)

Step 5 :- checks if selected cell is empty if occupied print an error message & restart it
if board [row] [col] != " "
print ("Invalid move")

Step 6 :- calls check_winner(board) to check for wining condition
* if a winner is found, prints the board and annouces the winner, then exists the function

* current_player = "o" if current_player = "x" else "x"

→ Switches the current player for the next turn
* After 9 turns if no winner prints the board and declares a draw

24/9/2024

Code:

Tic-Tac-Toe

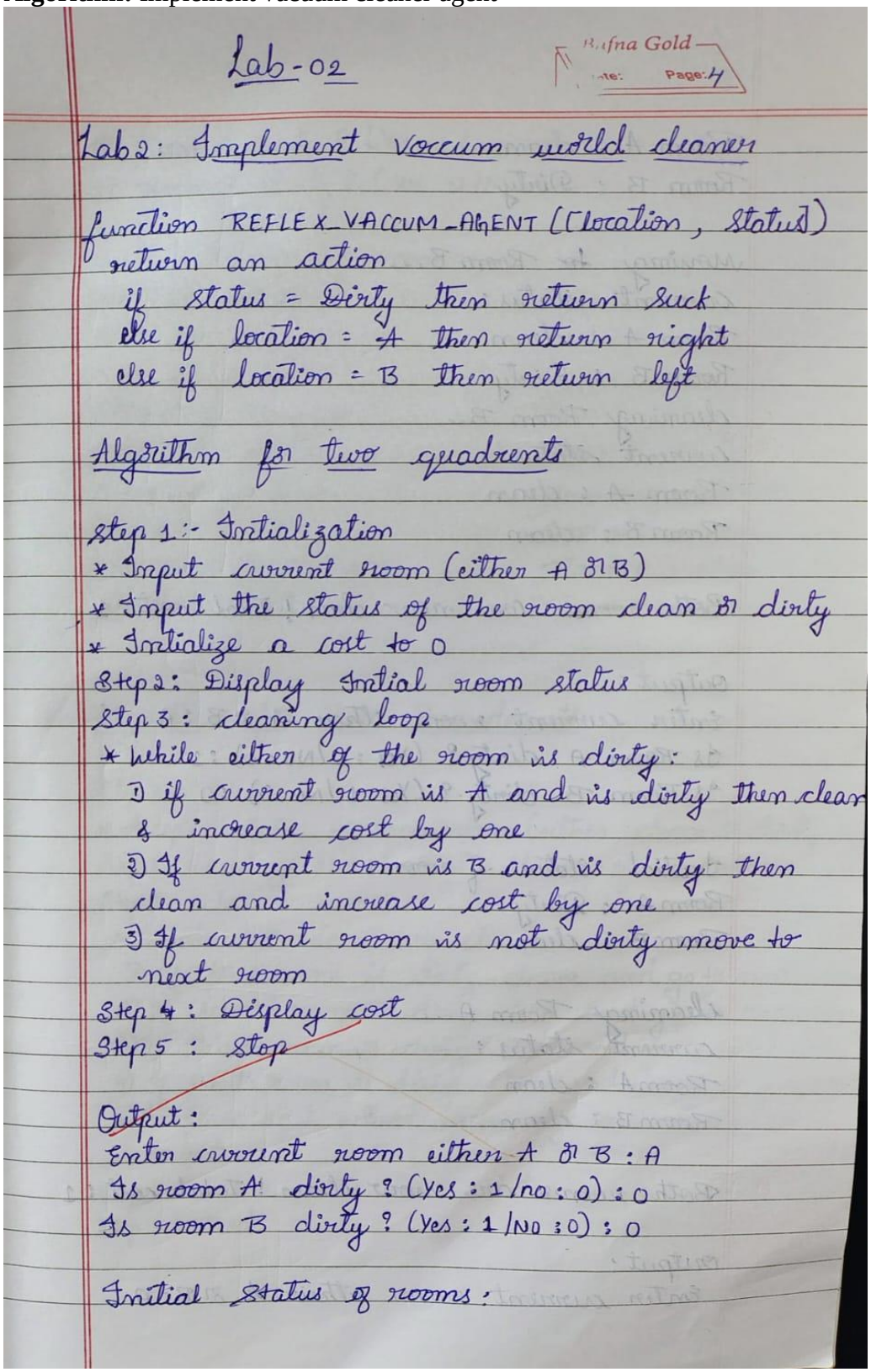
```
# Set up the game board as a 2D list
board = [["-", "-", "-"],
["-", "-", "-"],
["-", "-", "-"]]
# Define a function to print the game board
def print_board():
for row in board:
print(" | ".join(row))
# Define a function to handle a player's turn
def take_turn(player):
print(player + "'s turn.")
position = input("Choose a position from 1-9: ")
while position not in ["1", "2", "3", "4", "5", "6", "7", "8", "9"]:
position = input("Invalid input. Choose a position from 1-9: ")
position = int(position) - 1
row, col = divmod(position, 3)
while board[row][col] != "-":
position = int(input("Position already taken. Choose a different position: ")) - 1
row, col = divmod(position, 3)
board[row][col] = player
print_board()
# Define a function to check if the game is over
def check_game_over():
# Check for a win
for i in range(3):
if board[i][0] == board[i][1] == board[i][2] != "-":
return "win"
```

```
if board[0][i] == board[1][i] == board[2][i] != "-":
return "win"
if board[0][0] == board[1][1] == board[2][2] != "-":
return "win"
if board[0][2] == board[1][1] == board[2][0] != "-":
return "win"
# Check for a tie
elif all(cell != "-" for row in board for cell in row):
return "tie"
# Game is not over
else:
return "play"
# Define the main game loop
def play_game():
print_board()
current_player = "X"
game_over = False
while not game_over:
take_turn(current_player)
game_result = check_game_over()
if game_result == "win":
print(current_player + " wins!")
game_over = True
elif game_result == "tie":
print("It's a tie!")
game_over = True
else:
# Switch to the other player
current_player = "O" if current_player == "X" else "X"
# Start the game
play_game()
```

Output:

```
- | - | -
- | - | -
- | - | -
X's turn.
Choose a position from 1-9: 9
- | - | -
- | - | -
- | - | X
O's turn.
Choose a position from 1-9: 3
- | - | O
- | - | -
- | - | X
X's turn.
Choose a position from 1-9: 4
- | - | O
X | - | -
- | - | X
O's turn.
Choose a position from 1-9: 5
- | - | O
X | O | -
- | - | X
X's turn.
Choose a position from 1-9: 2
- | X | O
X | O | -
- | - | X
O's turn.
Choose a position from 1-9: 2
Position already taken. Choose a different position: 4
Position already taken. Choose a different position: 8
- | X | O
X | O | -
- | O | X
X's turn.
Choose a position from 1-9: 1
X | X | O
X | O | -
- | O | X
O's turn.
Choose a position from 1-9: 6
X | X | O
X | O | O
- | O | X
X's turn.
```

Algorithm: Implement vacuum cleaner agent



Lab-02

Rufna Gold
Page: 4

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

Algorithm for two quadrents

Step 1:- Initialization
* Input current room (either A or B)
* Input the status of the room clean or dirty
* Initialize a cost to 0
Step 2: Display initial room status
Step 3: cleaning loop
* while either of the room is dirty:
1) if current room is A and is dirty then clean & increase cost by one
2) If current room is B and is dirty then clean and increase cost by one
3) If current room is not dirty move to next room
Step 4: Display cost
Step 5: Stop

Output:
Enter current room either A or B: A
Is room A dirty? (Yes: 1/no: 0): 0
Is room B dirty? (Yes: 1/NO: 0): 0

Initial Status of rooms:

Room A : clean
Room B : Dirty

Moving to Room B...
current status:
Room A : clean
Room B : Dirty
cleaning Room B...
current status
Room A : clean
Room B : clean

Both rooms are now clean! Total cost : 1

Output:
Enter current room either A or B : A
Is Room A dirty? (Yes : 1 / No : 0) : 1
Is Room B dirty? (Yes : 1 / No : 0) : 0

Initial status of rooms:
Room A : Dirty
Room B : clean

cleaning Room A...
current status:
Room A : clean
Room B : clean

Both rooms are now clean! Total cost : 1

output:
Enter current room eitheir A or B : A

Is Room A dirty? (Yes: 1 / no: 0): 1
Is Room B dirty? (Yes: N / NO: 0): 1

Initial status of rooms:
Room A: Dirty
Room B: Dirty

Cleaning Room A...
current status:
Room A: clean
Room B: Dirty
Moving to Room B...
current status:
Room A: clean
Room B: Dirty
cleaning Room B...
current status:
Room A: clean
Room B: clean

Algorithm for four Quandrants

Step 1: Start
* Accept status of each room (either clean or dirty)
Step 2: Display the initial status of all rooms
Step 3: cleans rooms
* while all rooms are clean
1) If first room is dirty clean and go to next room
2) If second room is dirty clean and go to next room
3) If third room is dirty clean and go to next room
4) If fourth room is dirty clean and go to next room
5) Increase cost when cleaning and decreases count
Step 4: Show the last status
Step 5: End

Code:

Vaccum Cleaner

```
#For two quadrants
def vacuum_cleaner_simulation():

 current_room = input("Enter current room either A or B: ").upper()
 room_A = int(input("Is Room A dirty? (yes:1/no:0): "))
 room_B = int(input("Is Room B dirty? (yes:1/no:0): "))
 cost = 0
 def display_rooms():
 print(f"Room A: {'Clean' if room_A == 0 else 'Dirty'}")
 print(f"Room B: {'Clean' if room_B == 0 else 'Dirty'}")

 print("\nInitial status of rooms:")
 display_rooms()
 print()


 while room_A == 1 or room_B == 1:
 if current_room == 'A' and room_A == 1:
 print("Cleaning Room A...")
 room_A = 0
 cost += 1
 elif current_room == 'B' and room_B == 1:
 print("Cleaning Room B...")
 room_B = 0
 cost += 1
 else:
 current_room = 'B' if current_room == 'A' else 'A'
 print(f"Moving to Room {current_room}...")
 print("Current status:")
 display_rooms()

 print(f"\nBoth rooms are now clean! Total cost: {cost}")
vacuum_cleaner_simulation()


#For four quadrants
def vacuum_cleaner_simulation():
 current_room = input("Enter current room (A, B, C, or D): ").upper()
 room_A = int(input("Is Room A dirty? (yes:1/no:0): "))
 room_B = int(input("Is Room B dirty? (yes:1/no:0): "))
 room_C = int(input("Is Room C dirty? (yes:1/no:0): "))
 room_D = int(input("Is Room D dirty? (yes:1/no:0): "))
 cost = 0
```

```
count=2
def display_rooms():
print(f"Room A: {'Clean' if room_A == 0 else 'Dirty'}")
print(f"Room B: {'Clean' if room_B == 0 else 'Dirty'}")
print(f"Room C: {'Clean' if room_C == 0 else 'Dirty'}")
print(f"Room D: {'Clean' if room_D == 0 else 'Dirty'}")

print("\nInitial status of rooms:")
display_rooms()
print()


while room_A == 1 or room_B == 1 or room_C == 1 or room_D == 1:
if count==0:
print("Vacuum is recharging")
count=2
else:
if current_room == 'A' and room_A == 1:
print("Cleaning Room A...")
room_A = 0
cost += 1
count-=1
elif current_room == 'B' and room_B == 1:
print("Cleaning Room B...")
room_B = 0
cost += 1
count-=1
elif current_room == 'C' and room_C == 1:
print("Cleaning Room C...")
room_C = 0
cost += 1
count-=1
elif current_room == 'D' and room_D == 1:
print("Cleaning Room D...")
room_D = 0
cost += 1
count-=1
else:
if current_room == 'A':
current_room = 'B'
elif current_room == 'B':
current_room = 'C'
elif current_room == 'C':
current_room = 'D'
else:
current_room = 'A'
print(f"Moving to Room {current_room}...")
```

```
print("\nCurrent status:")
display_rooms()

print(f"\nAll rooms are now clean! Total cost: {cost}")
vacuum_cleaner_simulation()
```

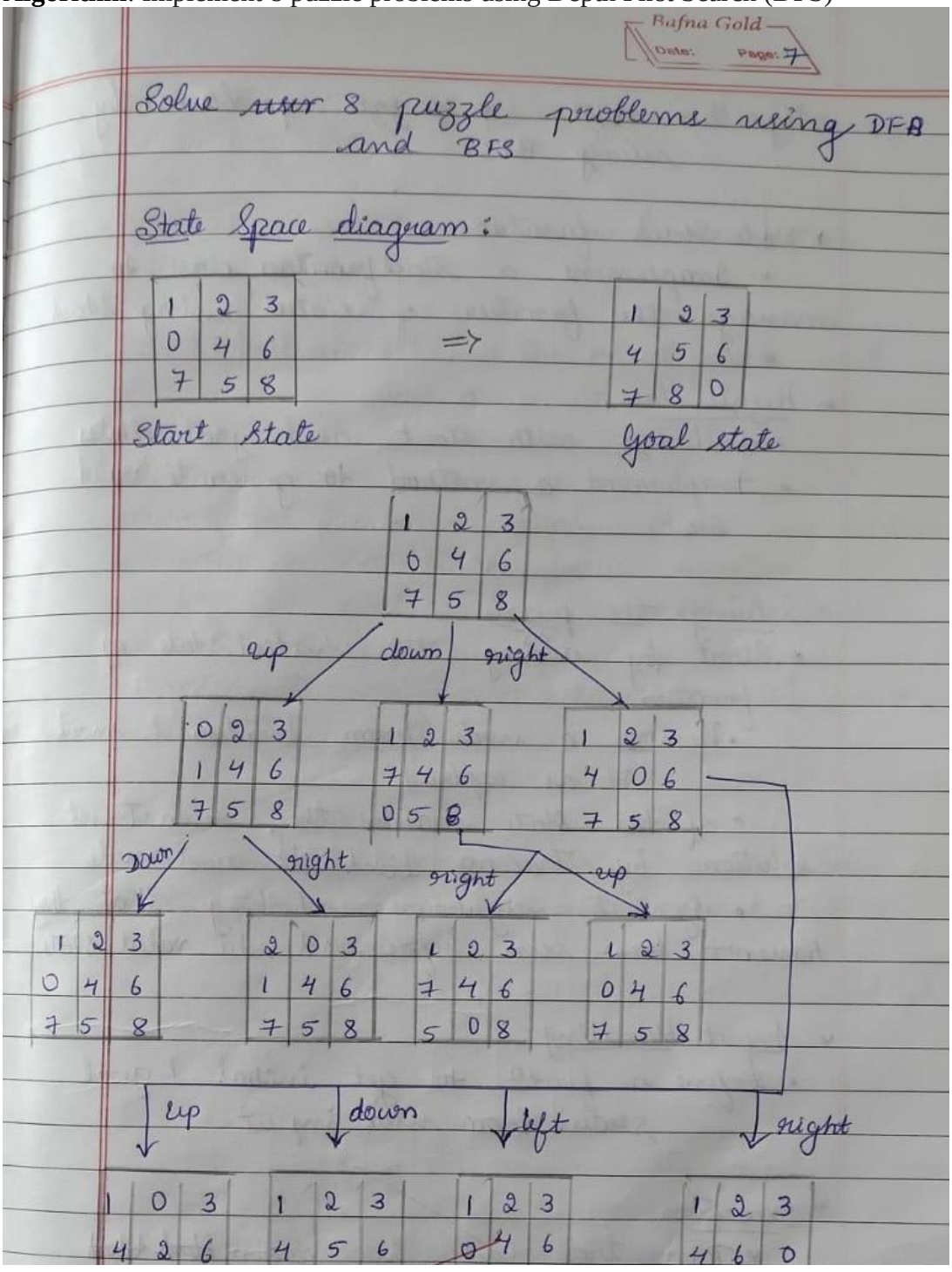
Output:

```
Enter current room either A or B: A
Is Room A dirty? (yes:1/no:0): 0
Is Room B dirty? (yes:1/no:0): 1

Initial status of rooms:
Room A: Clean
Room B: Dirty

Moving to Room B...
Current status:
Room A: Clean
Room B: Dirty
Cleaning Room B...
Current status:
Room A: Clean
Room B: Clean

Both rooms are now clean! Total cost: 1
Enter current room (A, B, C, or D): C
Is Room A dirty? (yes:1/no:0): 1
Is Room B dirty? (yes:1/no:0): 0
Is Room C dirty? (yes:1/no:0): 1
Is Room D dirty? (yes:1/no:0): 0

Initial status of rooms:
Room A: Dirty
Room B: Clean
Room C: Dirty
Room D: Clean

Cleaning Room C...
Moving to Room D...
Moving to Room A...
Cleaning Room A...

Current status:
Room A: Clean
Room B: Clean
Room C: Clean
Room D: Clean

All rooms are now clean! Total cost: 2
```

Program 2

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

Algorithm: Implement 8 puzzle problems using Depth First Search (DFS)



Algorithm for 8 puzzle problems by using BFS

* Node Stack frontier:
  * Implement a StackFrontier class to manage the frontier of states using Stack.
  *

* Puzzle class:
  * Initialize with start and goal States
  * Implement a method to generate valid States

* Solving the puzzle:
  * Start by adding the initial state of frontier
    • Remove a node from frontier & mark it as explored
    • If the state matches the, reconstruct solution by tracing back to root node
    • Generate neighbours, checking that they haven't been seen before and add valid state

* Input Handling:
  • Define a funtn to get initial & goal state from user input

* Execution:
  * Run the algo/ in main ~~block~~ blocks.

Bafna Gold
Date: Page: 8

Algorithm for 8 puzzles problems by using DFS

* Input : Read the start and goal states from user

* Initialization : Set up the start node and frontier.

* Loop:
  - If the frontier is empty, report no solution
  - Remove a node from frontier
  - If its goal state, backtrack to find the solution path.
  - Generate neighbours & add valid new states to frontier.

* Solution process:
  - check if the new state has not been seen before [not in the frontier and not explored].
  - If valid create a new node and add it to frontier.

* Output:
  - After finding a solution, the print_solution method output initial & goal states. the number of explored states and series of moves taken to reach goal.

9/10/2024

Code:

Implement 8 puzzle problems using Depth First Search (DFS)

```
import numpy as np


class Node:
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action


class StackFrontier:
    def __init__(self):
        self.frontier = []

    def add(self, node):
        self.frontier.append(node)

    def contains_state(self, state):
        return any((node.state[0] == state[0]).all() for node in self.frontier)

    def empty(self):
        return len(self.frontier) == 0

    def remove(self):
        if self.empty():
            raise Exception("Empty Frontier")
        else:
            node = self.frontier[-1]  # Remove from the end (LIFO)
            self.frontier = self.frontier[:-1]
            return node


class Puzzle:
    def __init__(self, start, startIndex, goal, goalIndex):
        self.start = [start, startIndex]
        self.goal = [goal, goalIndex]
        self.solution = None

    def neighbors(self, state):
        mat, (row, col) = state
        results = []
```

```
    if row > 0:  # Move up
        mat1 = np.copy(mat)
        mat1[row][col] = mat1[row - 1][col]
        mat1[row - 1][col] = 0
        results.append(('up', [mat1, (row - 1, col)]))
    if col > 0:  # Move left
        mat1 = np.copy(mat)
        mat1[row][col] = mat1[row][col - 1]
        mat1[row][col - 1] = 0
        results.append(('left', [mat1, (row, col - 1)]))
    if row < 2:  # Move down
        mat1 = np.copy(mat)
        mat1[row][col] = mat1[row + 1][col]
        mat1[row + 1][col] = 0
        results.append(('down', [mat1, (row + 1, col)]))
    if col < 2:  # Move right
        mat1 = np.copy(mat)
        mat1[row][col] = mat1[row][col + 1]
        mat1[row][col + 1] = 0
        results.append(('right', [mat1, (row, col + 1)]))

    return results

def print_solution(self):
    if self.solution is not None:
        print("Start State:\n", self.start[0], "\n")
        print("Goal State:\n", self.goal[0], "\n")
        print("\nStates Explored: ", self.num_explored, "\n")
        for idx, state in enumerate(self.explored):
            print(f"{idx + 1}. \n{state[0]}\n")
        print("Solution Steps:")
        for action, cell in zip(self.solution[0], self.solution[1]):
            print(f"Move {action}:\n{cell[0]}\n")
        print("Goal Reached!!")
    else:
        print("No solution found.")

def does_not_contain_state(self, state):
    for st in self.explored:
        if (st[0] == state[0]).all():
            return False
    return True

def solve(self):
    self.num_explored = 0
    self.explored = []  # Initialize explored states
```

```
        start = Node(state=self.start, parent=None, action=None)
        frontier = StackFrontier()  # Use StackFrontier for DFS
        frontier.add(start)

        while True:
            if frontier.empty():
                raise Exception("No solution")

            node = frontier.remove()
            self.num_explored += 1
            self.explored.append(node.state)  # Add current state to explored

            if (node.state[0] == self.goal[0]).all():
                actions = []
                cells = []
                while node.parent is not None:
                    actions.append(node.action)
                    cells.append(node.state)
                    node = node.parent
                actions.reverse()
                cells.reverse()
                self.solution = (actions, cells)
                return

            for action, state in self.neighbors(node.state):
                if not frontier.contains_state(state) and self.does_not_contain_state(state):
                    child = Node(state=state, parent=node, action=action)
                    frontier.add(child)


def input_puzzle():
    print("Enter the initial state (3x3 matrix, use spaces to separate numbers):")
    start = np.array([list(map(int, input().split())) for _ in range(3)])

    print("Enter the goal state (3x3 matrix, use spaces to separate numbers):")
    goal = np.array([list(map(int, input().split())) for _ in range(3)])

    startIndex = tuple(map(int, np.argwhere(start == 0)[0]))  # Find position of 0 in start
    goalIndex = tuple(map(int, np.argwhere(goal == 0)[0]))  # Find position of 0 in goal

    return start, startIndex, goal, goalIndex

if __name__ == "__main__":
    start, startIndex, goal, goalIndex = input_puzzle()
    p = Puzzle(start, startIndex, goal, goalIndex)
    p.solve()
    p.print_solution()
```

Output:

```
Enter the start state (use 0 for the blank):
1 2 3
0 4 6
7 5 8
Enter the goal state (use 0 for the blank):
1 2 3
4 5 6
7 0 8
Enter the maximum depth for search: 9

Searching at depth level: 0

Searching at depth level: 1

Searching at depth level: 2

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

1 2 3
4 0 6
7 5 8

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

Code:

```
import sys
import numpy as np


class Node:
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action


class QueueFrontier:
    def __init__(self):
        self.frontier = []

    def add(self, node):
        self.frontier.append(node)

    def contains_state(self, state):
        return any((node.state[0] == state[0]).all() for node in self.frontier)

    def empty(self):
        return len(self.frontier) == 0

    def remove(self):
        if self.empty():
            raise Exception("Empty Frontier")
        else:
            node = self.frontier[0]  # Remove from front
            self.frontier = self.frontier[1:]
            return node


class Puzzle:
    def __init__(self, start, startIndex, goal, goalIndex):
        self.start = [start, startIndex]
        self.goal = [goal, goalIndex]
        self.solution = None

    def neighbors(self, state):
        mat, (row, col) = state
        results = []

        if row > 0:  # Move up
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row - 1][col]
            mat1[row - 1][col] = 0
```

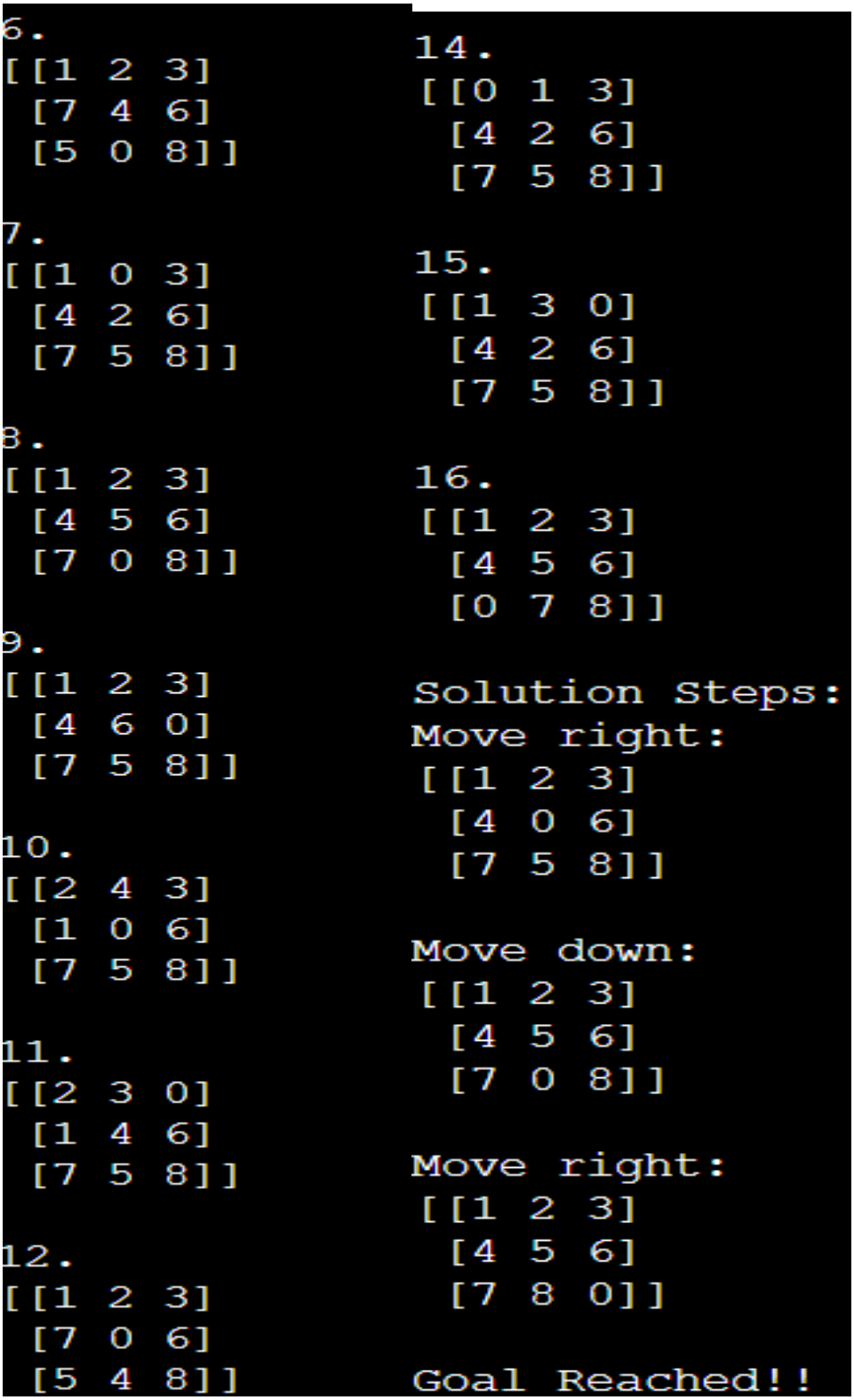
```
            results.append(('up', [mat1, (row - 1, col)]))
        if col > 0:  # Move left
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row][col - 1]
            mat1[row][col - 1] = 0
            results.append(('left', [mat1, (row, col - 1)]))
        if row < 2:  # Move down
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row + 1][col]
            mat1[row + 1][col] = 0
            results.append(('down', [mat1, (row + 1, col)]))
        if col < 2:  # Move right
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row][col + 1]
            mat1[row][col + 1] = 0
            results.append(('right', [mat1, (row, col + 1)]))

        return results

    def print_explored_states(self):
        print("All Explored States:")
        for idx, state in enumerate(self.explored):
            print(f"{idx + 1}. \n{state[0]}\n")

    def print_solution(self):
        if self.solution is not None:
            print("Solution Steps:")
            for action, cell in zip(self.solution[0], self.solution[1]):
                print(f"Move {action}:")
                print(f"{cell[0]}\n")
            print("Goal Reached!!")
        else:
            print("No solution found.")

    def does_not_contain_state(self, state):
        for st in self.explored:
            if (st[0] == state[0]).all():
                return False
        return True

    def solve(self):
        self.num_explored = 0

        start = Node(state=self.start, parent=None, action=None)
        frontier = QueueFrontier()
        frontier.add(start)
```

```
        self.explored = []

        while True:
            if frontier.empty():
                raise Exception("No solution")

            node = frontier.remove()
            self.num_explored += 1

            if (node.state[0] == self.goal[0]).all():
                actions = []
                cells = []
                while node.parent is not None:
                    actions.append(node.action)
                    cells.append(node.state)
                    node = node.parent
                actions.reverse()
                cells.reverse()
                self.solution = (actions, cells)
                return

            self.explored.append(node.state)

            for action, state in self.neighbors(node.state):
                if not frontier.contains_state(state) and self.does_not_contain_state(state):
                    child = Node(state=state, parent=node, action=action)
                    frontier.add(child)


def input_puzzle():
    print("Enter the initial state (3x3 matrix, use spaces to separate numbers):")
    start = np.array([list(map(int, input().split())) for _ in range(3)])

    print("Enter the goal state (3x3 matrix, use spaces to separate numbers):")
    goal = np.array([list(map(int, input().split())) for _ in range(3)])

    startIndex = tuple(map(int, np.argwhere(start == 0)[0]))  # Find position of 0 in start
    goalIndex = tuple(map(int, np.argwhere(goal == 0)[0]))  # Find position of 0 in goal

    return start, startIndex, goal, goalIndex


if __name__ == "__main__":
    start, startIndex, goal, goalIndex = input_puzzle()
    p = Puzzle(start, startIndex, goal, goalIndex)
    p.solve()
    p.print_explored_states()
```

p.print_solution()

Output:

```
Enter the initial state (3x3 matrix, use spaces to separate numbers):
1 2 3
0 4 6
7 5 8
Enter the goal state (3x3 matrix, use spaces to separate numbers):
1 2 3
4 5 6
7 8 0
All Explored States:
1.
[[1 2 3]
 [0 4 6]
 [7 5 8]]

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

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

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

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



Algorithm: Implement Iterative deepening search algorithm

Lab - 5

22/10/24

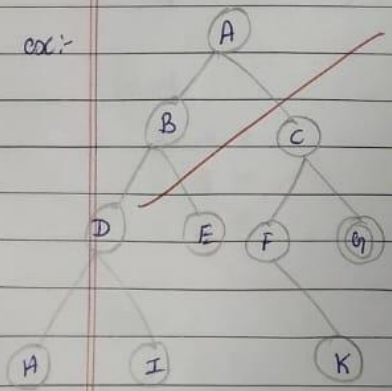
Implement Iterative Deepening Search Algorithm

Algorithm :-

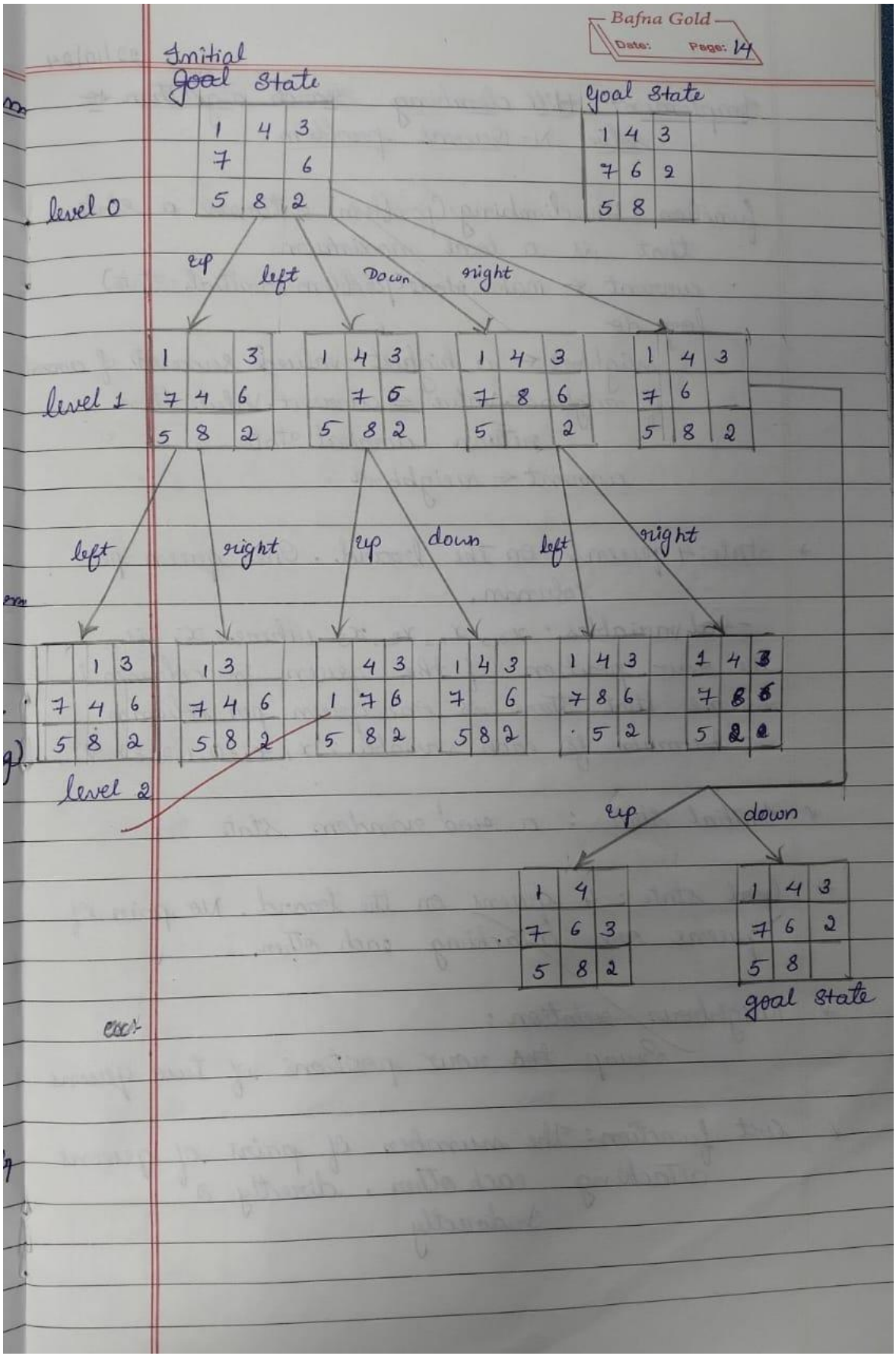
function Iterative-Deping Search (problem) returns a solution, or failure
  for depth = 0 to ∞ do
    result ← Depth-limited-Search (problem, depth)
    if result ≠ cutoff then return result

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

ex :-



1st iteration → A
2nd iteration → A, B, C
3rd iteration → A, B, D, E, C, F, G
4th iteration → A, B, D, E, H, I, E, F, K, G


Bafna Gold
Date: Page: 14
Initial
goal State
1 4 3
7 6
5 8 2
level 0
goal State
1 4 3
7 6 2
5 8
up
left
Down
right
level 1
left
right
up
down
left
right
level 2
up
down
goal State

Code:

```
import numpy as np
import heapq

class PuzzleState:
    def __init__(self, board, level=0, parent=None):
        self.board = board
        self.level = level
        self.parent = parent
        self.blank_pos = self.find_blank()
        self.goal = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 0]])

    def find_blank(self):
        return np.argwhere(self.board == 0)[0]

    def is_goal(self):
        return np.array_equal(self.board, self.goal)

    def get_neighbors(self):
        neighbors = []
        x, y = self.blank_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 = self.board.copy()
                new_board[x, y], new_board[new_x, new_y] = new_board[new_x, new_y], new_board[x,
y]
                neighbors.append(PuzzleState(new_board, self.level + 1, self))
        return neighbors

    def heuristic(self):
        return np.sum(self.board != self.goal) - 1  # Count of misplaced tiles

    def __lt__(self, other):
        return (self.level + self.heuristic()) < (other.level + other.heuristic())

def a_star_search(initial_state):
    open_set = []
    heapq.heappush(open_set, initial_state)
    closed_set = set()

    while open_set:
        current_state = heapq.heappop(open_set)

        if current_state.is_goal():
```

```
            return current_state

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

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

    return None

def print_solution(solution):
    path = []
    while solution:
        path.append(solution.board)
        solution = solution.parent
    path.reverse()

    for level, board in enumerate(path):
        print(f"Level {level}:\n{board}\n")

if __name__ == "__main__":
    initial_input = input("Enter initial state (e.g., '1 2 3 4 5 6 0 7 8'): ")
    initial_state = np.array(list(map(int, initial_input.split()))).reshape(3, 3)
    initial_puzzle = PuzzleState(initial_state)

    solution = a_star_search(initial_puzzle)
    if solution:
        print("Solution found:")
        print_solution(solution)
    else:
        print("No solution exists.")
```

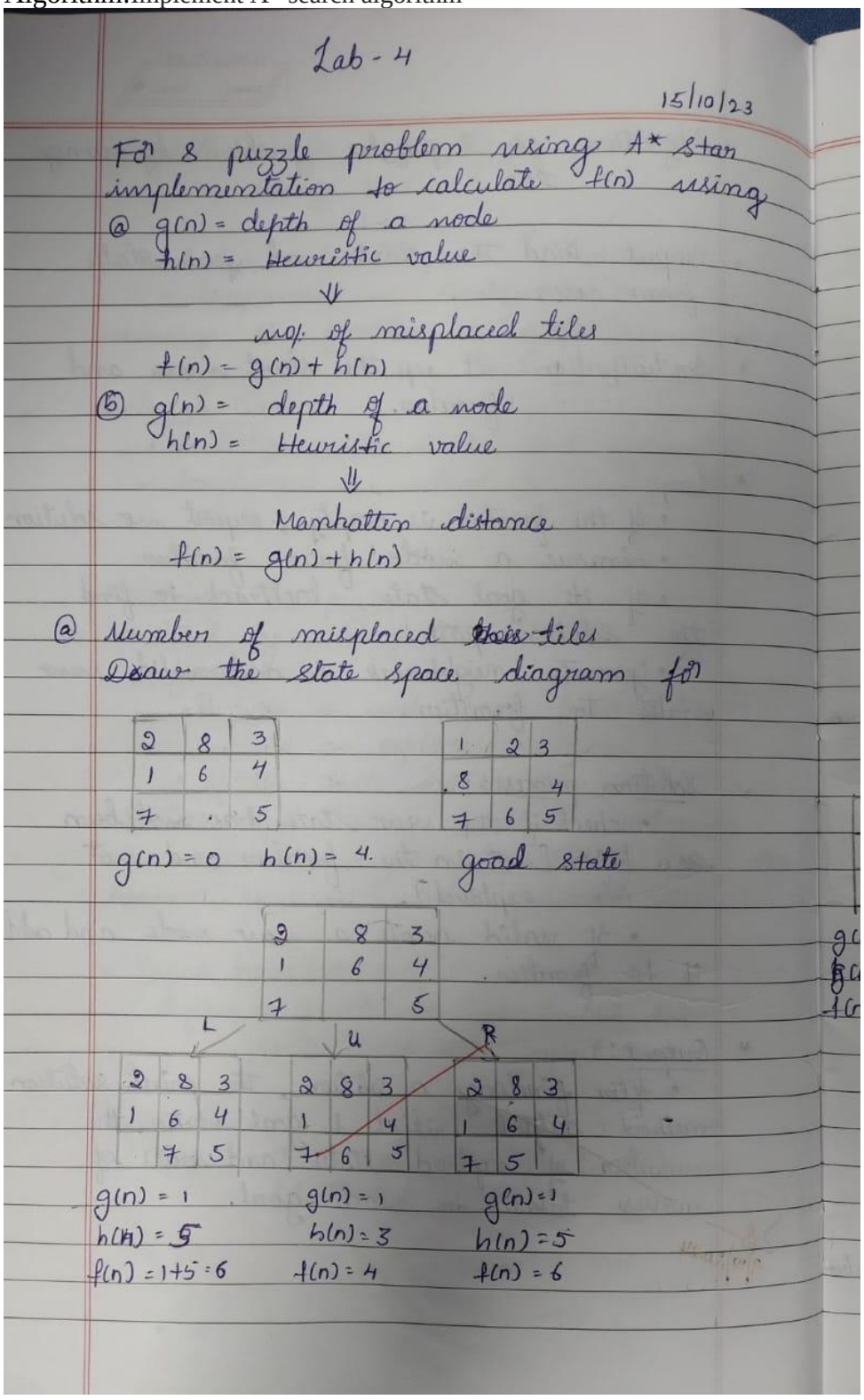
Output:

```
Enter initial state (e.g., '1 2 3 4 5 6 0 7 8'): 1 2 3 4 0 5 6 7 8
Solution found:
Level 0:
[[1 2 3]
 [4 0 5]
 [6 7 8]]

Level 1:
[[1 2 3]
 [4 5 0]
 [6 7 8]]

Level 2:
[[1 2 3]
 [4 5 8]
 [6 7 0]]

Level 3:
[[1 2 3]
 [4 5 8]
 [6 0 7]]

Level 4:
[[1 2 3]
 [4 5 8]
 [0 6 7]]

Level 5:
[[1 2 3]
 [0 5 8]
 [4 6 7]]

Level 6:
[[1 2 3]
 [5 0 8]
 [4 6 7]]

Level 7:
[[1 2 3]
 [5 6 8]
 [4 0 7]]

Level 8:
[[1 2 3]
 [5 6 8]
 [4 7 0]]

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

Program 3

Implement A* search algorithm

Algorithm:Implement A* search algorithm



Lab - 4

15/10/23

For 8 puzzle problem using A* star 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 of a node
h(n) = Heuristic value
⇓
Manhattan distance
f(n) = g(n) + h(n)

a) Number of misplaced tiles
Draw the state space diagram for

2	8	3
1	6	4
7		5

1	2	3
8		4
7	6	5

g(n) = 0 h(n) = 4. good state

2	8	3
1	6	4
7		5

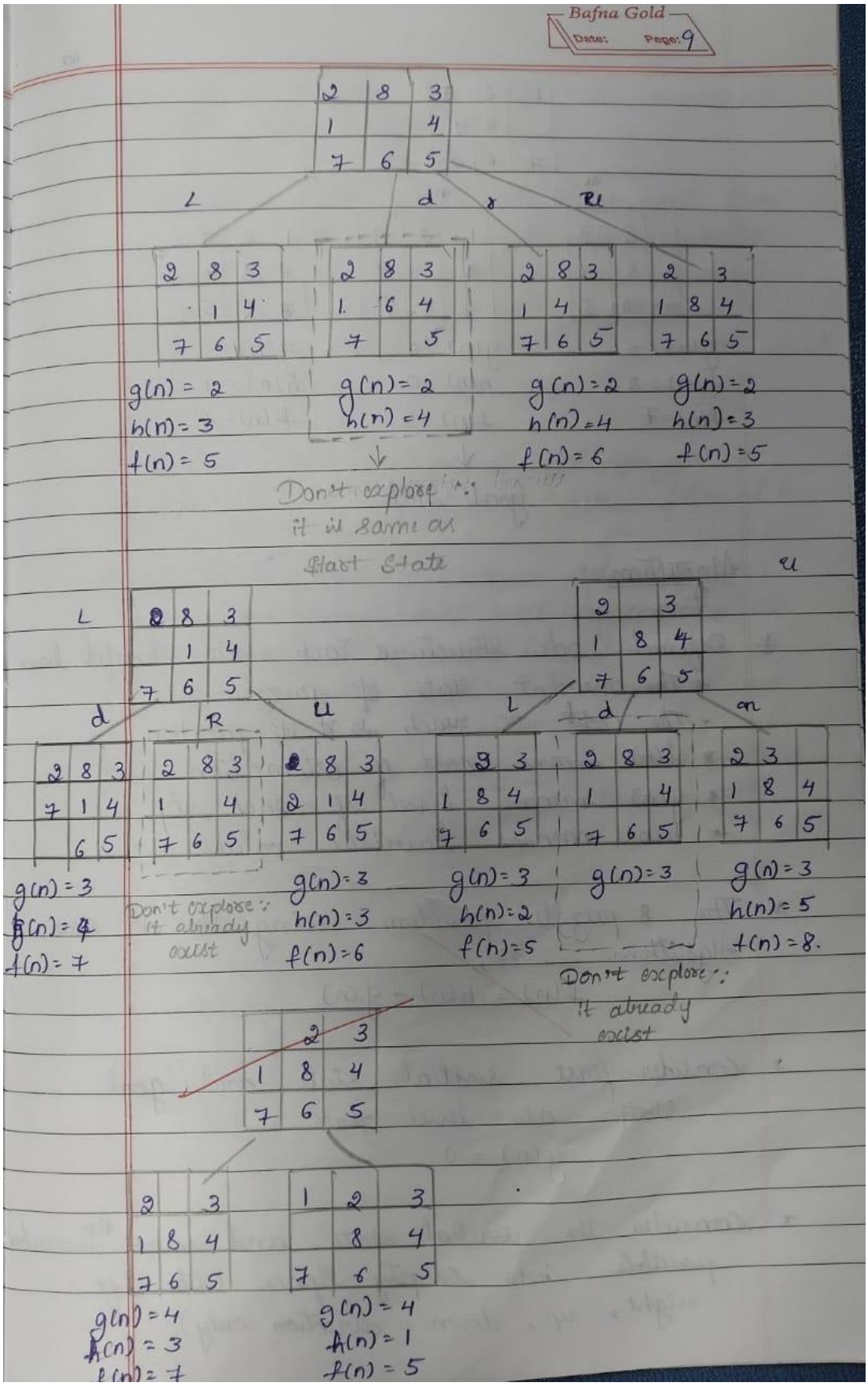
L U R

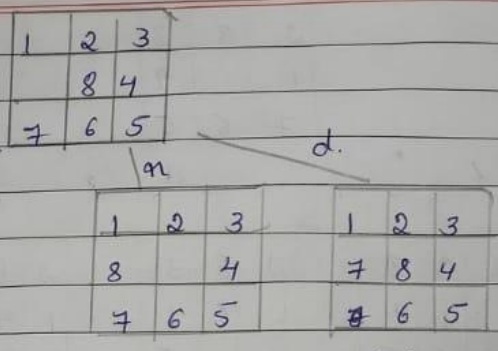
2	8	3
1	6	4
	7	5

2	8	3
1		4
7	6	5

2	8	3
1	6	4
7	5	

g(n) = 1 g(n) = 1 g(n) = 1
h(n) = 5 h(n) = 3 h(n) = 5
f(n) = 1+5 = 6 f(n) = 4 f(n) = 6


L
d
R
g(n) = 2
h(n) = 3
f(n) = 5
g(n) = 2
h(n) = 4
Don't explore ∵ it is same as start state
g(n) = 2
h(n) = 4
f(n) = 6
g(n) = 2
h(n) = 3
f(n) = 5
L
d
R
u
g(n) = 3
f(n) = 7
Don't explore ∵ it already exist
g(n) = 3
h(n) = 3
f(n) = 6
g(n) = 3
h(n) = 2
f(n) = 5
g(n) = 3
h(n) = 5
f(n) = 8.
Don't explore ∵ it already exist
g(n) = 4
h(n) = 3
g(n) = 4
h(n) = 1
f(n) = 5



↓

Goal State

Algorithm :

* Defined node structure Each node should have
  * The current state of puzzle
  * The cost to reach to it its position
  * f(n) means cost of estimation
  * g(n) means level of each step
  * h(n) means heuristic values

* The 8 puzzle problem using the A* Star Algorithm . so,

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

* Consider first initial state and goal state as level zero

$$g(n) = 0.$$

* Consider the initial state and move the number ~~possible~~ into empty space . (like left, right, up, down direction only)

* In each & step consider the each levels and find heuristic value.

* Compare the 8 puzzle table with goal state compare each cell if it doesn't match consider as 1 and add the 1 How much cells of elements does not match with cell of goal state. calculate the all counts as heuristic value.

* f(n) is cost estimation of that puzzle it has formula $f(n) = h(n) + g(n)$ followed #

* which one has minimum value of h(n). [heuristic value] that will be consider in next level, it goes on like this till reaching the goal state

* If any puzzles already exist in previous level then don't explore those puzzle

(b) Manhatten distance

2	8	3
1	6	4
7		5

Initial state

$g(n) = 0$ $h(n)$

1	2	3
8		4
7	6	5

goal state

1 2 3 4 5 6 7 8.

$1+1+0+0+0+1+0+1 = 4$

2	8	3
1	6	4
7		5

l u r

2	8	3
1	6	4
	7	5

$g(n) = 1$

$h(n) = 1+1+1+1+2$

$h(n) = 6$

$f(n) = 7$

2	8	3
1		4
7	6	5

$g(n) = 1$

$h(n) = 1+1+2$

$h(n) = 4$

$f(n) = 5$

2	8	3
1	6	4
7	5	

$g(n) = 1$

$h(n) = 1+1+1+1+2 = 6$

$f(n) = 7$

2	8	3
1		4
7	6	5

L u d r

2	8	3
	1	4
7	6	5

$g(n) = 2$

$h(n) = 2+1+2$

$= 5$

$f(n) = 7$

2		3
1	8	4
7	6	5

$g(n) = 2$

$h(n) = 1+1+1$

$h(n) = 3$

$f(n) = 5$

2	8	3
1	6	4
7		5

$g(n) = 2$

$h(n) = 1+1+1+2$

$h(n) = 5$

$f(n) = 7$

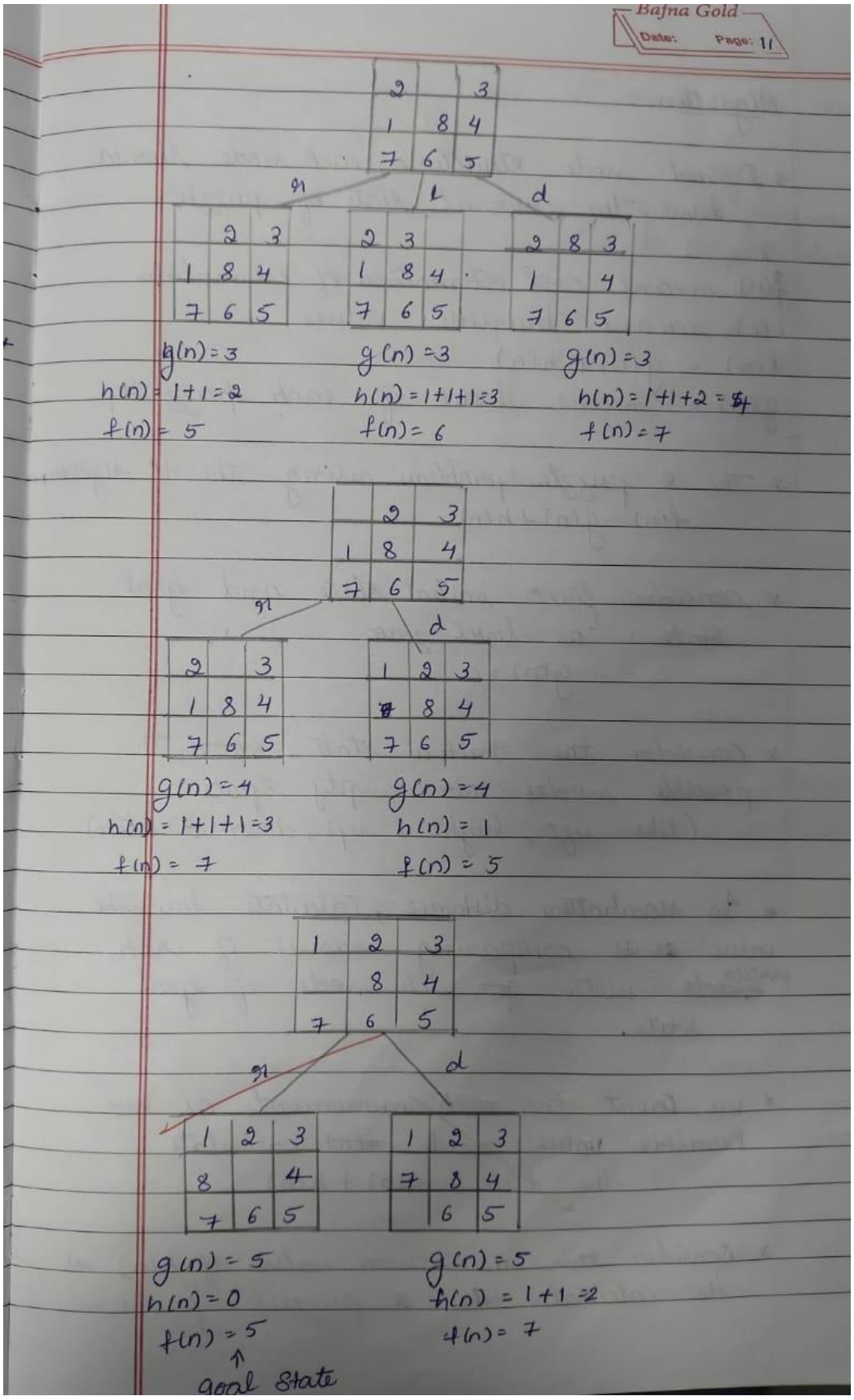
2	8	3
1	4	
7	6	5

$g(n) = 2$

$h(n) = 1+1+1+2$

$= 5$

$f(n) = 7$


2 3
1 8 4
7 6 5
r
l
d
g(n) = 3
h(n) = 1+1 = 2
f(n) = 5
g(n) = 3
h(n) = 1+1+1 = 3
f(n) = 6
g(n) = 3
h(n) = 1+1+2 = 4
f(n) = 7
r
d
g(n) = 4
h(n) = 1+1+1 = 3
f(n) = 7
g(n) = 4
h(n) = 1
f(n) = 5
r
d
g(n) = 5
h(n) = 0
f(n) = 5
goal state
g(n) = 5
h(n) = 1+1 = 2
f(n) = 7

Algorithm:

* Defined node structure each node should have the current state of puzzle

g(n) means cost estimation of estimation
h(n) means heuristic values
f(n) = g(n) + h(n).
g(n) = means level of each fuzz step

* The 8 puzzle problem using the A* Algorithm
f(n) = g(n) + h(n)

* consider first initial state and goal state as level zero
g(n) = 0

* Consider the Initial state. move the possible nodes into empty space.
(like left, right, up, down direction)

* In Manhatten distance, Calculate heuristic value as it is comparing moves of each node with go each node of goal state.

* we count the each movements as heuristic value and f(n) calculate the f(n) = g(n) + h(n)

* Consider the minimum value of h(n) as to calculate the 8 puzzles problem

Code:

```
import heapq

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

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

def generate_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))

    return neighbors

def reconstruct_path(came_from, current):

    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
    path.reverse()
    return path

def a_star(start, goal):
    open_list = []
    heapq.heappush(open_list, (0 + misplaced_tile(start, goal), 0, start))

    g_score = {start: 0}
    came_from = {}

    visited = set()

    while open_list:
        _, g, current = heapq.heappop(open_list)
```

```
        if current == goal:
            path = reconstruct_path(came_from, current)
            return path, g

        visited.add(current)

        for neighbor in generate_neighbors(current):
            if neighbor in visited:
                continue
            tentative_g = g_score[current] + 1

            if tentative_g < g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score = tentative_g + misplaced_tile(neighbor, goal)  # f(n) = g(n) + h(n)

                heapq.heappush(open_list, (f_score, tentative_g, neighbor))

    return None, None

def print_state(state):

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

def get_state_from_user(prompt):

    state = []
    for i in range(3):
        row = input(f"{prompt} row {i+1} (space-separated): ")
        state.append(tuple(map(int, row.split())))
    return tuple(state)

if __name__ == "__main__":
    print("Enter the initial state:")
    start_state = get_state_from_user("Initial state")
    print("\nEnter the goal state:")
    goal_state = get_state_from_user("Goal state")

    print("\nInitial State:")
    print_state(start_state)

    print("\nGoal State:")
    print_state(goal_state)

    solution, cost = a_star(start_state, goal_state)

    if solution:
        print(f"\nSolution found with cost: {cost}")
        print("Steps:")
```

```
    for step in solution:
        print_state(step)
else:
    print("\nNo solution found.")
```

Output:

```
Enter the initial state:
Initial state row 1 (space-separated): 2 8 3
Initial state row 2 (space-separated): 1 6 4
Initial state row 3 (space-separated): 7 0 5

Enter the goal state:
Goal state row 1 (space-separated): 1 2 3
Goal state row 2 (space-separated): 8 0 4
Goal state row 3 (space-separated): 7 6 5

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


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


Solution found with cost: 5
Steps:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

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

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

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

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

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

Code:

```
import heapq

def manhattan_distance(state, goal_state):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:
                goal_i, goal_j = find_position(value, goal_state)
                distance += abs(i - goal_i) + abs(j - goal_j)
    return distance

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

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

def generate_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))

    return neighbors

def reconstruct_path(came_from, current):
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
    path.reverse()
    return path

def a_star(start, goal):
    open_list = []
    heapq.heappush(open_list, (0 + manhattan_distance(start, goal), 0, start))

    g_score = {start: 0}
```

```
    came_from = {}

    visited = set()

    while open_list:
        _, g, current = heapq.heappop(open_list)

        if current == goal:
            path = reconstruct_path(came_from, current)
            return path, g

        visited.add(current)

        for neighbor in generate_neighbors(current):
            if neighbor in visited:
                continue
            tentative_g = g_score[current] + 1

            if tentative_g < g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score = tentative_g + manhattan_distance(neighbor, goal)

                heapq.heappush(open_list, (f_score, tentative_g, neighbor))

    return None, None

def print_state(state):
    for row in state:
        print(row)
    print()

def get_state_from_user(prompt):
    state = []
    for i in range(3):
        row = input(f"{prompt} row {i+1} (space-separated): ")
        state.append(tuple(map(int, row.split())))
    return tuple(state)

if __name__ == "__main__":
    print("Enter the initial state:")
    start_state = get_state_from_user("Initial state")
    print("\nEnter the goal state:")
    goal_state = get_state_from_user("Goal state")

    print("\nInitial State:")
    print_state(start_state)

    print("\nGoal State:")
    print_state(goal_state)

    solution, cost = a_star(start_state, goal_state)
```

```
    if solution:
        print(f"\nSolution found with cost: {cost}")
        print("Steps:")
        for step in solution:
            print_state(step)
    else:
        print("\nNo solution found.")
```

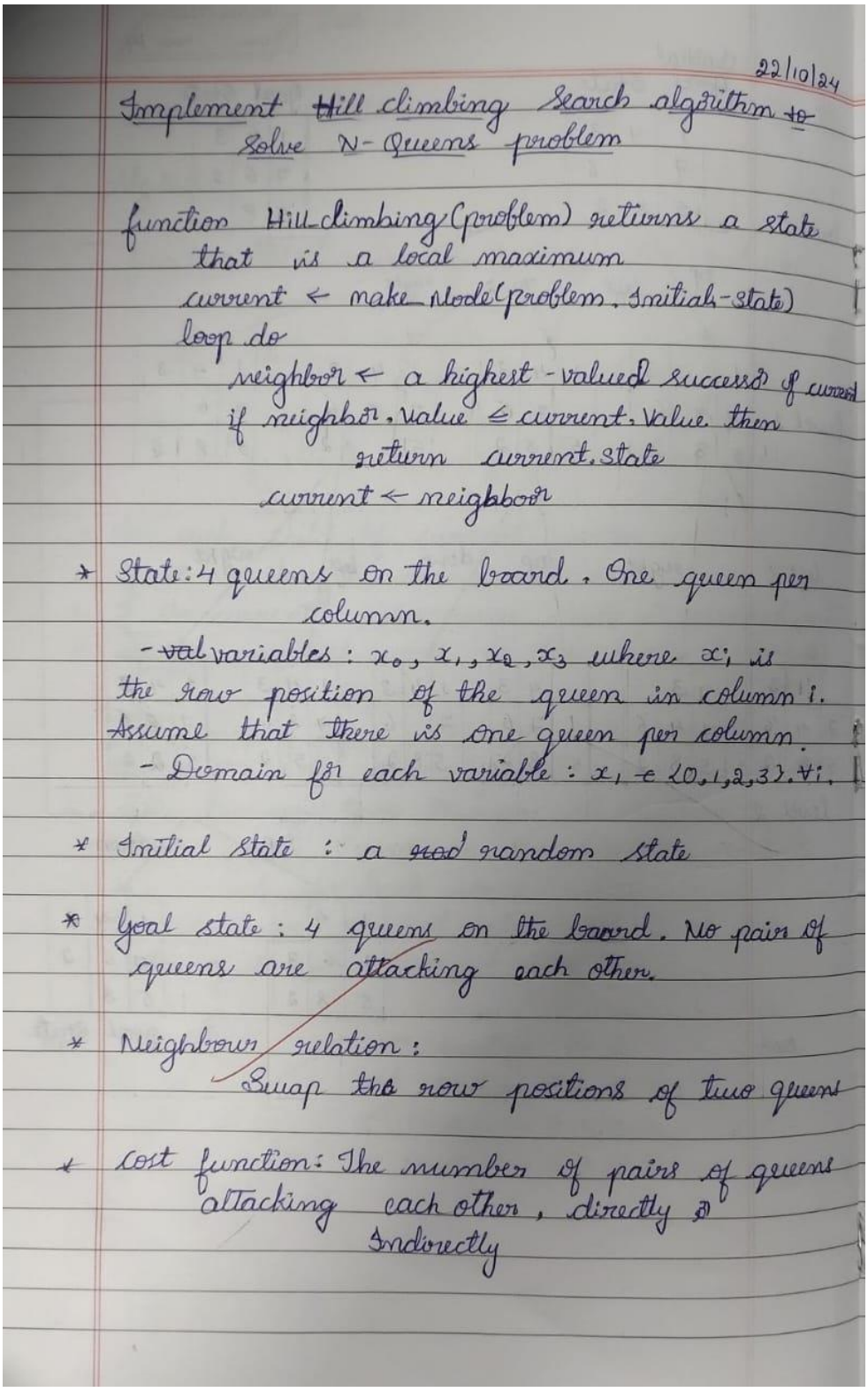
Output:

```
Enter the initial state:
Initial state row 1 (space-separated): 1 5 3
Initial state row 2 (space-separated): 4 2 6
Initial state row 3 (space-separated): 7 0 8

Enter the goal state:
Goal state row 1 (space-separated): 1 2 3
Goal state row 2 (space-separated): 4 5 6
Goal state row 3 (space-separated): 7 8 0

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


Goal State:
(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



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
neighbor ← a highest-valued successor of current
if neighbor.Value ≤ current.Value then
return current.State
current ← neighbor

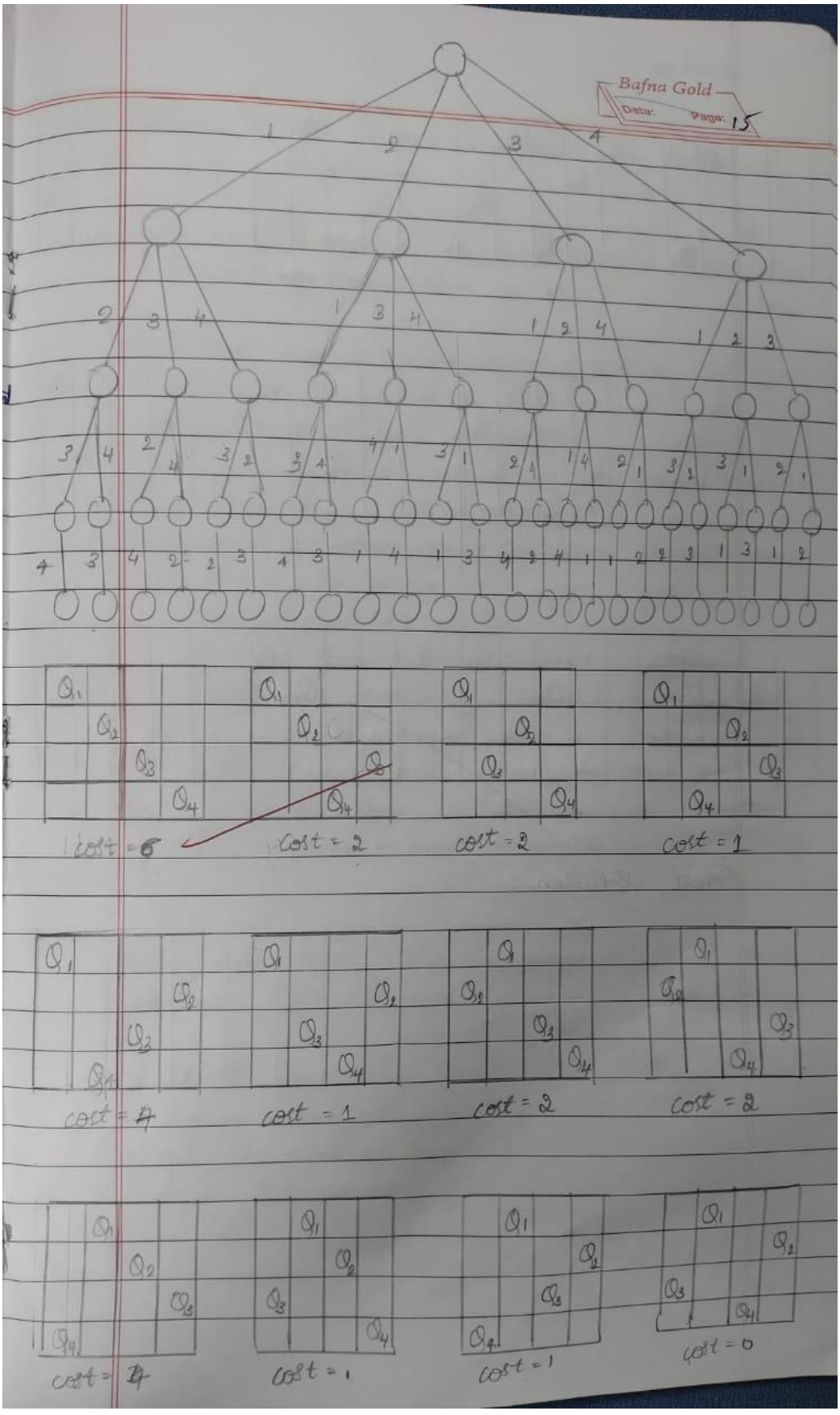
* State: 4 queens on the board. One queen per column.
- variables: x_0, x_1, x_2, x_3 where x_i is the row position of the queen in column i. Assume that there is one queen per column.
- Domain for each variable: $x_i \in \{0,1,2,3\}, \forall i$.

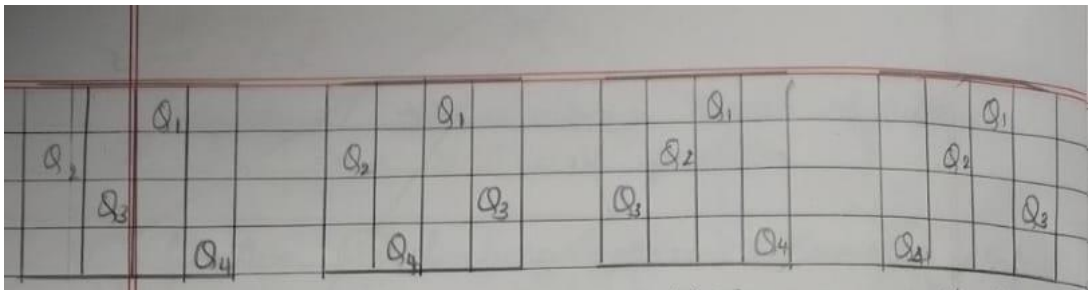
* Initial State: a random state

* Goal state: 4 queens on the board. No pair of queens are attacking each other.

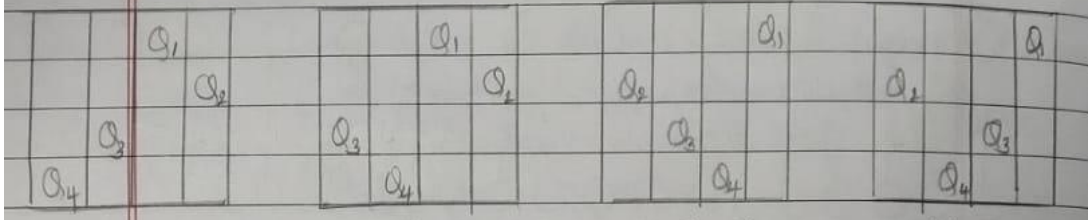
* Neighbour relation:
Swap the row positions of two queens

* cost function: The number of pairs of queens attacking each other, directly & Indirectly

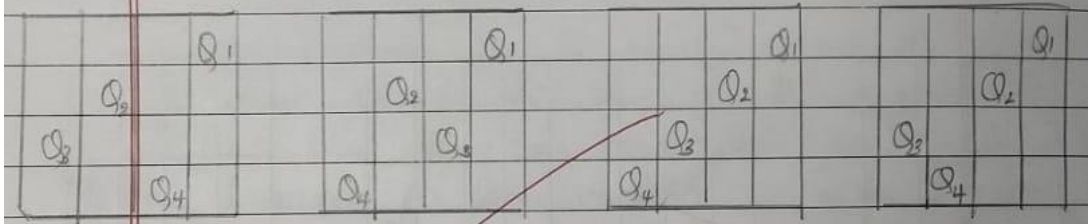

Bafna Gold
Date:
Page: 15
cost = 6
cost = 2
cost = 2
cost = 1
cost = 4
cost = 1
cost = 2
cost = 2
cost = 4
cost = 1
cost = 1
cost = 0



cost = 1 cost = 0 cost = 4 cost = 1

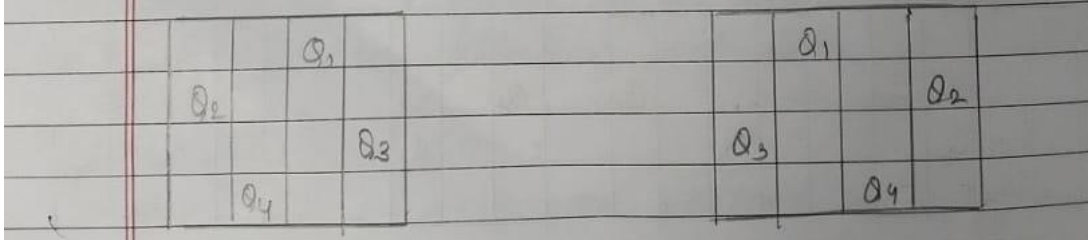


cost = 2 cost = 2 cost = 4 cost = 1



cost = 1 cost = 2 cost = 6 cost = 2

Best Solution :-



Code:

```
from random import randint
N = int(input("Enter the number of queens:"))

def configureRandomly(board, state):

	for i in range(N):
		state[i] = randint(0, 100000) % N;
		board[state[i]][i] = 1;

def printBoard(board):
	for i in range(N):
		print(*board[i])

def printState( state):
	print(*state)


def compareStates(state1, state2):
	for i in range(N):
		if (state1[i] != state2[i]):
			return False;

	return True;


def fill(board, value):
	for i in range(N):
		for j in range(N):
			board[i][j] = value;


def calculateObjective( board, state):

	attacking = 0;

	for i in range(N):

		row = state[i]
		col = i - 1;
		while (col >= 0 and board[row][col] != 1) :
			col -= 1

		if (col >= 0 and board[row][col] == 1) :
			attacking += 1;

		row = state[i]
```

```
            col = i + 1;
            while (col < N and board[row][col] != 1):
                col += 1;

            if (col < N and board[row][col] == 1) :
                attacking += 1;

            row = state[i] - 1
            col = i - 1;
            while (col >= 0 and row >= 0 and board[row][col] != 1) :
                col-= 1;
                row-= 1;

            if (col >= 0 and row >= 0 and board[row][col] == 1) :
                attacking+= 1;

            row = state[i] + 1
            col = i + 1;
            while (col < N and row < N and board[row][col] != 1) :
                col+= 1;
                row+= 1;

            if (col < N and row < N and board[row][col] == 1) :
                attacking += 1;

            row = state[i] + 1
            col = i - 1;
            while (col >= 0 and row < N and board[row][col] != 1) :
                col -= 1;
                row += 1;

            if (col >= 0 and row < N and board[row][col] == 1) :
                attacking += 1;

            row = state[i] - 1
            col = i + 1;
            while (col < N and row >= 0 and board[row][col] != 1) :
                col += 1;
                row -= 1;

            if (col < N and row >= 0 and board[row][col] == 1) :
                attacking += 1;


    return int(attacking / 2);

def generateBoard( board, state):
```

```
        fill(board, 0);
        for i in range(N):
                board[state[i]][i] = 1;


def copyState( state1, state2):

        for i in range(N):
                state1[i] = state2[i];


def getNeighbour(board, state):

        opBoard = [[0 for _ in range(N)] for _ in range(N)]
        opState = [0 for _ in range(N)]

        copyState(opState, state);
        generateBoard(opBoard, opState);


        opObjective = calculateObjective(opBoard, opState);


        NeighbourBoard = [[0 for _ in range(N)] for _ in range(N)]

        NeighbourState = [0 for _ in range(N)]
        copyState(NeighbourState, state);
        generateBoard(NeighbourBoard, NeighbourState);


        for i in range(N):
                for j in range(N):

                        if (j != state[i]) :

                                NeighbourState[i] = j;
                                NeighbourBoard[NeighbourState[i]][i] = 1;
                                NeighbourBoard[state[i]][i] = 0;

                                temp = calculateObjective( NeighbourBoard, NeighbourState);


                                if (temp <= opObjective) :
                                        opObjective = temp;
                                        copyState(opState, NeighbourState);
                                        generateBoard(opBoard, opState);
```

```
                                NeighbourBoard[NeighbourState[i]][i] = 0;
                                NeighbourState[i] = state[i];
                                NeighbourBoard[state[i]][i] = 1;


        copyState(state, opState);
        fill(board, 0);
        generateBoard(board, state);

def hillClimbing(board, state):


        neighbourBoard = [[0 for _ in range(N)] for _ in range(N)]
        neighbourState = [0 for _ in range(N)]

        copyState(neighbourState, state);
        generateBoard(neighbourBoard, neighbourState);

        while True:

                # Copying the neighbour board and
                # state to the current board and
                # state, since a neighbour
                # becomes current after the jump.

                copyState(state, neighbourState);
                generateBoard(board, state);

                # Getting the optimal neighbour

                getNeighbour(neighbourBoard, neighbourState);

                if (compareStates(state, neighbourState)) :

                        # If neighbour and current are
                        # equal then no optimal neighbour
                        # exists and therefore output the
                        # result and break the loop.

                        printBoard(board);
                        break;

                elif (calculateObjective(board, state) == calculateObjective(
neighbourBoard,neighbourState)):

                        # If neighbour and current are
                        # not equal but their objectives
```

```
                # are equal then we are either
                # approaching a shoulder or a
                # local optimum, in any case,
                # jump to a random neighbour
                # to escape it.

                # Random neighbour
                neighbourState[randint(0, 100000) % N] = randint(0, 100000) % N;
                generateBoard(neighbourBoard, neighbourState);

# Driver code
state = [0] * N
board = [[0 for _ in range(N)] for _ in range(N)]
configureRandomly(board, state);
hillClimbing(board, state);
```

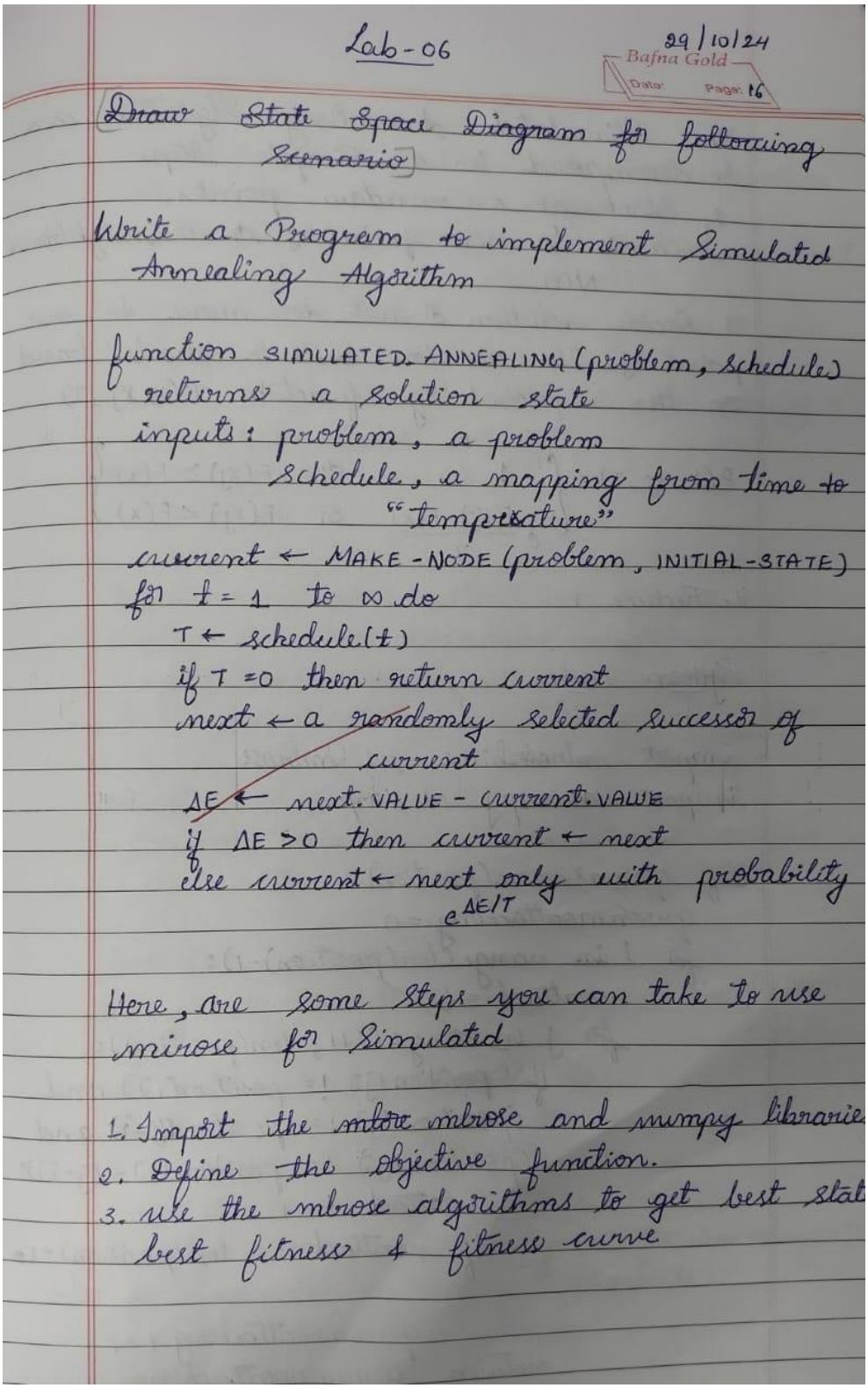
Output:

```
Enter the number of queens:8
0 0 0 0 0 1 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0
1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0
0 0 1 0 0 0 0 0
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm: Simulated Annealing to Solve 8-Queens problem



Lab - 06

29/10/24

Bafna Gold

Date: Page: 16

Draw State Space Diagram for following Scenario

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 $e^{\Delta E/T}$

Here, are some Steps you can take to use mlrose for Simulated

1. Import the mlrose and numpy libraries
2. Define the objective function.
3. use the mlrose algorithms to get best state best fitness & fitness curve

The Stimulated Annealing Algorithm can be decomposed in 4 Simple Steps

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

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

4. Reduce T

Python code

```
import mlrose_hiive as mlrose
import mumpy as np

def queensmax(position):
    queennoattacking = 0
    for i in range(len(position)-1):
        noattack = 0
        for j in range(i+1, len(position)):
            if (position[j] != position[i]) and
               (position[j] != position[j-i]) and
               (position[j] != position[i] - (j-i)):
                noattack += 1
            if noattack == len(position)-1
                -i:

                queennoattacking += 1
    return queennoattacking
```

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) # Round and convert to integers for queen positions
 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):
 # Check if queens are on the same row or on the same diagonal
 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, 8) for _ in range(8)]
1
# 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 # Flip sign to get the number of non-attacking queens
print('The best position found is:', best_position)
print('The number of queens that are not attacking each other is:', best_objective)
```

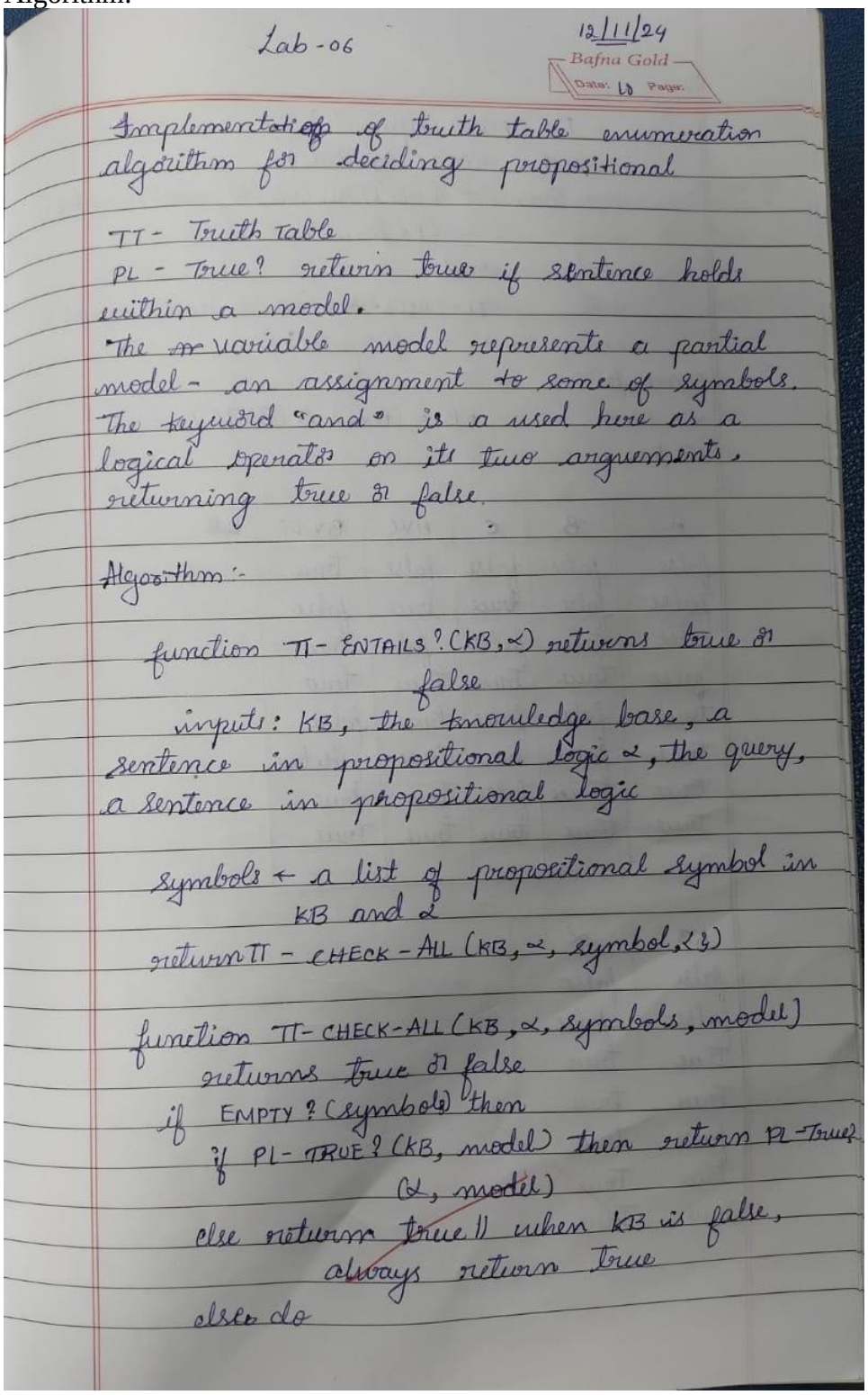
Output:

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

Program 6

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

Algorithm:



Lab - 06

12/11/24

Bafna Gold

Date: 19 Page:

Implementation of truth table enumeration algorithm for deciding propositional

TT - Truth Table

PL - True? returns true if sentence holds within a model.

The variable model represents a partial model - an assignment to some of symbols.

The keyword "and" is a used here as a logical operator on its two arguements, returning true or false.

Algorithm:-

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

inputs: KB, the knowledge base, a sentence in propositional logic α, the query, a sentence in propositional logic

symbols ← a list of propositional symbol in KB and α

return TT - CHECK - ALL (KB, α, symbol, {})

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

A	B	C	AVC	BV ¬C	KB
false	false	false	false	true	
false	false	true	true	false	
false	true	false	false	true	
false	true	true	true	true	
true	false	false	true	false	
true	false	true	true	~~false~~ true	
true	true	false	true	true	
true	true	true	true	true	

KB	α
false	false
false	false
true	true
true	true
false	true
true	true
true	true

Bafna Gold
Date: Page: 19

Output

A	B	C	KB	Alpha	KB entails Alpha?
false	false	false	false	false	No
false	false	true	false	false	No
false	false	false	false	true	No
false	false	true	true	true	Yes
true	false	false	true	true	Yes
true	false	true	false	true	No
true	true	false	true	true	Yes
true	true	true	true	true	Yes

Does KB entails alpha ? True

14/11/2024

Code:

```
import itertools

def evaluate_formula(formula, valuation):
    """
    Evaluate the propositional formula under the given truth assignment (valuation).
    The formula is a string of logical operators like 'AND', 'OR', 'NOT', and can contain variables 'A', 'B', 'C'.
    """
    # Create a local environment (dictionary) for variable assignments
    env = {var: valuation[i] for i, var in enumerate(['A', 'B', 'C'])}

    # Replace logical operators with Python equivalents
    formula = formula.replace('AND', 'and').replace('OR', 'or').replace('NOT', 'not')

    # Replace variables in the formula with their corresponding truth values
    for var in env:
        formula = formula.replace(var, str(env[var]))

    # Evaluate the formula and return the result (True or False)
    try:
        return eval(formula)
    except Exception as e:
        raise ValueError(f"Error in evaluating formula: {e}")

def truth_table(variables):
    """
    Generate all possible truth assignments for the given variables.
    """
    return list(itertools.product([False, True], repeat=len(variables)))

def entails(KB, alpha):
    """
    Decide if KB entails alpha using a truth-table enumeration algorithm.
    KB is a propositional formula (string), and alpha is another propositional formula (string).
    """
    # Generate all possible truth assignments for A, B, and C
    assignments = truth_table(['A', 'B', 'C'])

    print(f"{'A':<10}{'B':<10}{'C':<10}{'KB':<15}{'alpha':<15}{'KB entails alpha?'}")  # Header for the truth
table
    print("-" * 70)  # Separator for readability

    for assignment in assignments:
        # Evaluate KB and alpha under the current assignment
        KB_value = evaluate_formula(KB, assignment)
        alpha_value = evaluate_formula(alpha, assignment)

        # Print the current truth assignment and the results for KB and alpha

print(f"{str(assignment[0]):<10}{str(assignment[1]):<10}{str(assignment[2]):<10}{str(KB_value):<15}{str(al
pha_value):<15}{'Yes' if KB_value and alpha_value else 'No'}")
```

```
        # If KB is true and alpha is false, then KB does not entail alpha
        if KB_value and not alpha_value:
            return False

    # If no counterexample was found, then KB entails alpha
    return True

# Define the formulas for KB and alpha
alpha = 'A OR B'
KB = '(A OR C) AND (B OR NOT C)'

# Check if KB entails alpha
result = entails(KB, alpha)

# Print the final result of entailment
print(f"\nDoes KB entail alpha? {result}")
```

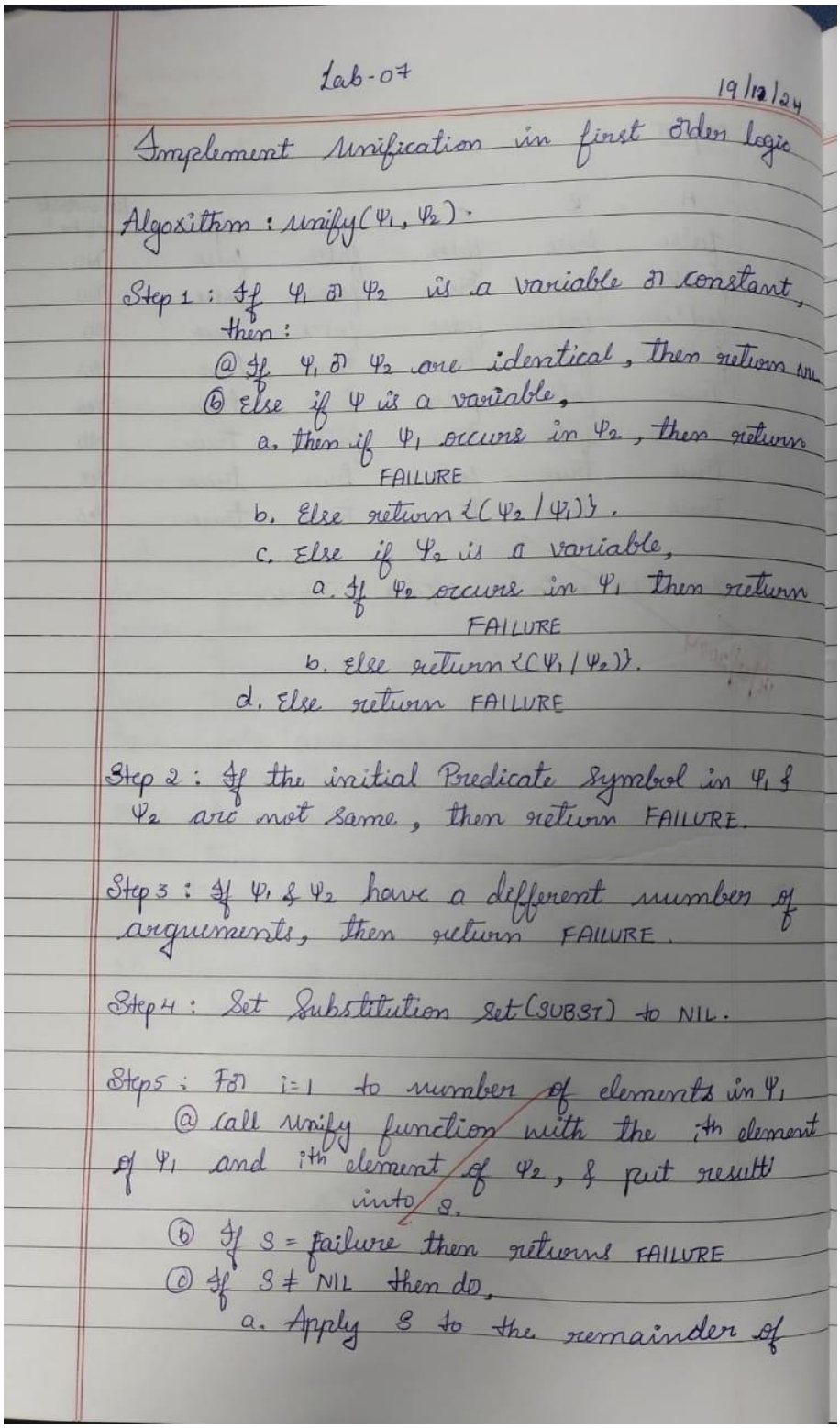
Output:

```
A         B         C         KB              alpha           KB entails alpha?
-------------------------------------------------------------------------------
False     False     False     False           False           No
False     False     True      False           False           No
False     True      False     False           True            No
False     True      True      True            True            Yes
True      False     False     True            True            Yes
True      False     True      False           True            No
True      True      False     True            True            Yes
True      True      True      True            True            Yes

Does KB entail alpha? True
```

Program 7

Implement unification in first order logic

Algorithm:



Lab-07

19/12/24

Implement Unification in first Order logic

Algorithm : Unify(Ψ_1, Ψ_2).

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 is a variable,

a. then if Ψ_1 occurs in Ψ_2, then return FAILURE

b. Else return {(Ψ_2 / Ψ_1)}.

c. Else if Ψ_2 is a variable,

a. If Ψ_2 occurs in Ψ_1 then return FAILURE

b. Else return {(Ψ_1 / Ψ_2)}.

d. Else return FAILURE

Step 2 : If the initial Predicate Symbol in Ψ_1 & Ψ_2 are not same, then return FAILURE.

Step 3 : If Ψ_1 & 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 number of elements in Ψ_1

(a) call Unify function with the ith element of Ψ_1 and ith element of Ψ_2, & put 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 & L2
b. SUBST = APPEND(S, SUBST)

Step6 : Return SUBST

* example 1:

P(X, F(Y)) — ①
P(a, F(g(x)) — ②

① & ② are identical if x is replaced with a in ①
a/x

P(a, F(Y)) — ①

if y is replaced with g(x)
P(a, F(g(x)) — ①.

NOW, ① & ② are equal so it is 'give as output "TRUE"

* example 2:
Q(a, g(x, a), f(y)) — ①
Q(a, g(f(b), a), x) — ②

if x is replaced with f(b) in ①. f(b)/x

Q(a, g(f(b), a), f(y)) — ①

~~if f(y) is replaced with b x in ① ↑~~
~~Q(a, g(f(b), a), x) — ①. x/f(y)~~

~~now, ① & ② are equal~~

Cannot able to replace with x in ①
because the same variable ea didn't
replaced twicely

so, it is not unified. it is
FAILURE

* example 3 =

$\psi_1 = p(b, x, f(g(z)))$ —①
$\psi_2 = p(z, f(y), f(y))$ —②

if z is replaced with b in ②

$\psi_2 = p(b, f(y), f(y))$ —②

if x is replaced with f(y) in ①

$\psi_1 = p(b, f(y), f(g(z)))$ —①

if y is replaced with g(z) in ②

$\psi_2 = p(b, f(y), f(g(z)))$ —②

Now, ① & ② are equal so, it is unified

* example 4 =

$\psi_1 = P(f(a), g(y))$
$\psi_2 = P(x, x)$

x cannot replaced twice so it
is failure.

Code:

```
class Term:
    def __init__(self, symbol, args=None):
        self.symbol = symbol
        self.args = args if args else []

    def __str__(self):
        if not self.args:
            return str(self.symbol)
        return f"{self.symbol}({','.join(str(arg) for arg in self.args)})"

    def is_variable(self):
        return isinstance(self.symbol, str) and self.symbol.isupper() and not self.args

def occurs_check(var, term, substitution):
    """Check if variable occurs in term"""
    if term.is_variable():
        if term.symbol in substitution:
            return occurs_check(var, substitution[term.symbol], substitution)
        return var.symbol == term.symbol
    return any(occurs_check(var, arg, substitution) for arg in term.args)

def substitute(term, substitution):
    """Apply substitution to term"""
    if term.is_variable() and term.symbol in substitution:
        return substitute(substitution[term.symbol], substitution)
    if not term.args:
        return term
    return Term(term.symbol, [substitute(arg, substitution) for arg in term.args])

def unify(term1, term2, substitution=None, iteration=1):
    """Unify two terms with detailed iteration steps"""
    if substitution is None:
        substitution = {}

    print(f"\nIteration {iteration}:")
    print(f"Attempting to unify: {term1} and {term2}")
    print(f"Current substitution: {', '.join(f'{k}->{v}' for k,v in substitution.items()) or 'empty'}")

    term1 = substitute(term1, substitution)
    term2 = substitute(term2, substitution)

    if term1.symbol == term2.symbol and not term1.args and not term2.args:
        print("Terms are identical - no substitution needed")
        return substitution
```

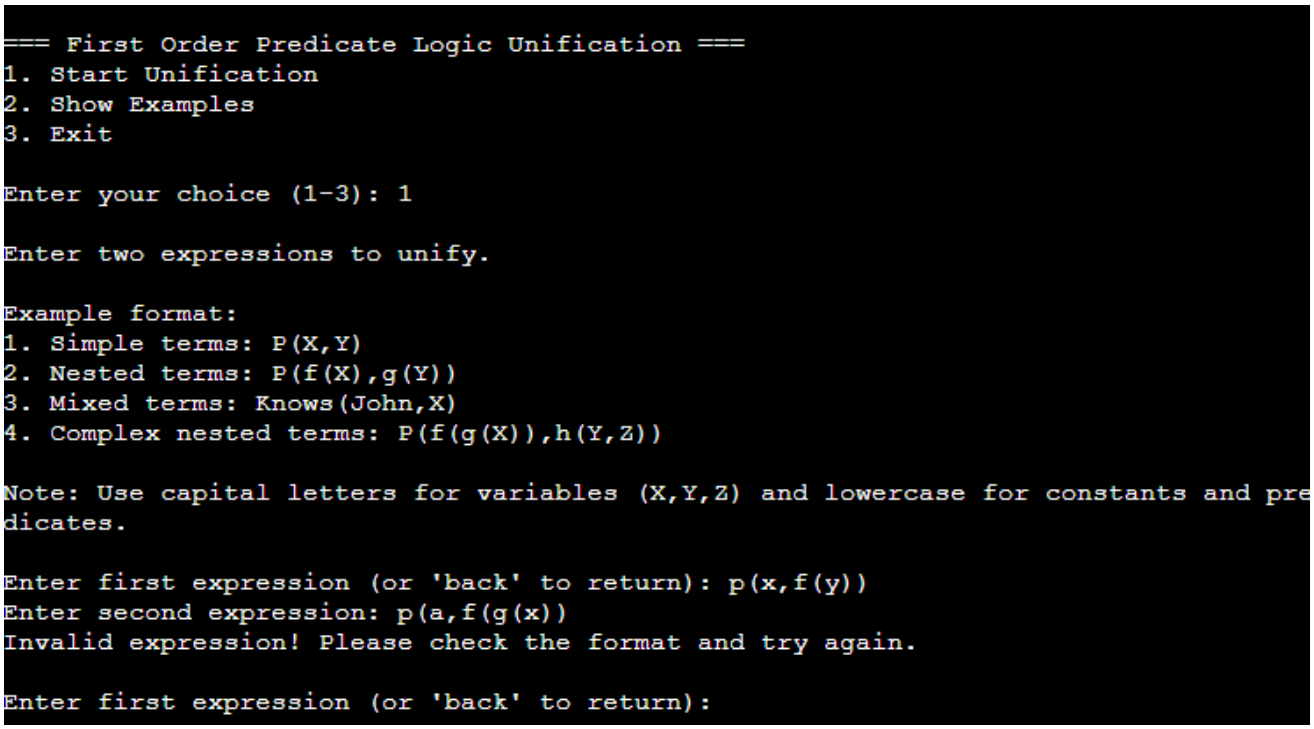
```
    if term1.is_variable():
        if occurs_check(term1, term2, substitution):
            print(f"Occurs check failed: {term1.symbol} occurs in {term2}")
            return None
        substitution[term1.symbol] = term2
        print(f"Added substitution: {term1.symbol} -> {term2}")
        return substitution

    if term2.is_variable():
        if occurs_check(term2, term1, substitution):
            print(f"Occurs check failed: {term2.symbol} occurs in {term1}")
            return None
        substitution[term2.symbol] = term1
        print(f"Added substitution: {term2.symbol} -> {term1}")
        return substitution

    if term1.symbol != term2.symbol or len(term1.args) != len(term2.args):
        print(f"Unification failed: Different predicates or argument lengths")
        return None

    for arg1, arg2 in zip(term1.args, term2.args):
        result = unify(arg1, arg2, substitution, iteration + 1)
        if result is None:
            return None
        substitution = result

    return substitution

def parse_term(s):
    """Parse terms like P(X,f(Y)) or X"""
    s = s.strip()
    if '(' not in s:
        return Term(s)

    pred = s[:s.index('(')]
    args_str = s[s.index('(')+1:s.rindex(')')]

    args = []
    current = ''
    depth = 0
    for c in args_str:
        if c == '(' or c == '[':
            depth += 1
        elif c == ')' or c == ']':
            depth -= 1
        elif c == ',' and depth == 0:
            args.append(parse_term(current.strip()))
```

```
            current = ''
            continue
        current += c
    if current:
        args.append(parse_term(current.strip()))

    return Term(pred, args)

def print_examples():
    print("\nExample format:")
    print("1. Simple terms: P(X,Y)")
    print("2. Nested terms: P(f(X),g(Y))")
    print("3. Mixed terms: Knows(John,X)")
    print("4. Complex nested terms: P(f(g(X)),h(Y,Z))")
    print("\nNote: Use capital letters for variables (X,Y,Z) and lowercase for constants and
predicates.")

def validate_input(expr):
    """Basic validation for input expressions"""
    if not expr:
        return False

    # Check balanced parentheses
    count = 0
    for char in expr:
        if char == '(':
            count += 1
        elif char == ')':
            count -= 1
        if count < 0:
            return False
    return count == 0

def main():
    while True:
        print("\n=== First Order Predicate Logic Unification ===")
        print("1. Start Unification")
        print("2. Show Examples")
        print("3. Exit")

        choice = input("\nEnter your choice (1-3): ")

        if choice == '1':
            print("\nEnter two expressions to unify.")
            print_examples()

            while True:
```

```
        expr1 = input("\nEnter first expression (or 'back' to return): ")
        if expr1.lower() == 'back':
            break

        if not validate_input(expr1):
            print("Invalid expression! Please check the format and try again.")
            continue

        expr2 = input("Enter second expression: ")
        if not validate_input(expr2):
            print("Invalid expression! Please check the format and try again.")
            continue

        try:
            term1 = parse_term(expr1)
            term2 = parse_term(expr2)

            print("\nUnification Process:")
            result = unify(term1, term2)

            print("\nFinal Result:")
            if result is None:
                print("Unification failed!")
            else:
                print("Unification successful!")
                print("Final substitutions:", ', '.join(f'{k}->{v}' for k,v in result.items()))

            retry = input("\nTry another unification? (y/n): ")
            if retry.lower() != 'y':
                break

        except Exception as e:
            print(f"Error processing expressions: {str(e)}")
            print("Please check your input format and try again.")

    elif choice == '2':
        print("\n=== Example Expressions ===")
        print("1. P(X,h(Y)) and P(a,f(Z))")
        print("2. P(f(a),g(Y)) and P(X,X)")
        print("3. Knows(John,X) and Knows(X,Elisabeth)")
        print("\nPress Enter to continue...")
        input()

    elif choice == '3':
        print("\nThank you for using the Unification Program!")
        break
```

```
        else:
            print("\nInvalid choice! Please enter 1, 2, or 3.")

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

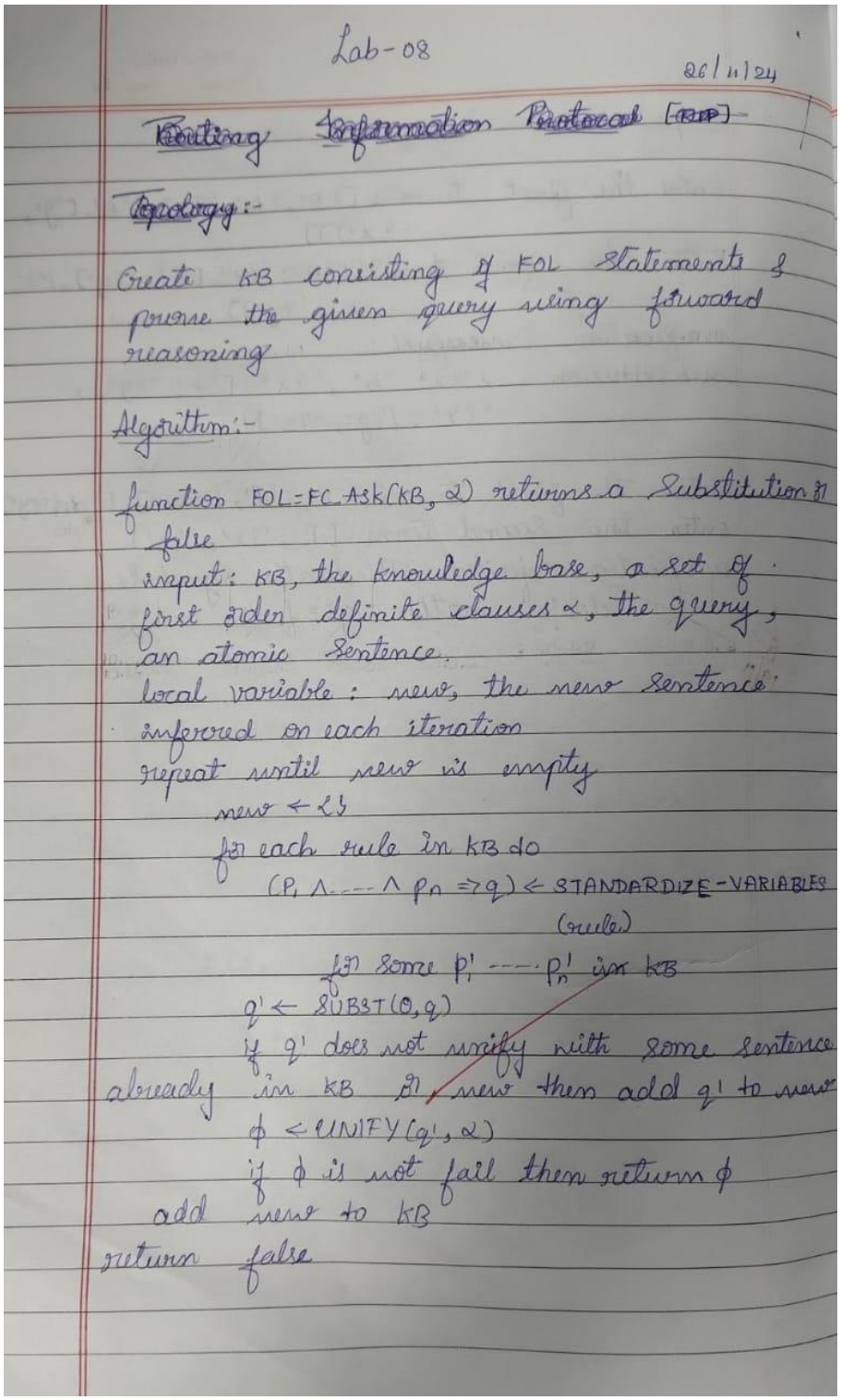
Output:



Program 8

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

Algorithm:



Lab-08

26/11/24

~~Routing Information Protocol [RIP]~~

~~Topology~~:-

Create KB consisting of FOL statements & prove the given query using forward reasoning

Algorithm:-

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

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

local variable: new, the new sentence inferred on each iteration

repeat until new is empty

new $\leftarrow \{\}$

for each rule in KB do

$(p_1 \wedge \dots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES (rule)

for some $p_1' \dots p_n'$ in KB

$q' \leftarrow$ SUBST(θ, q)

if q' does not unify with some sentence already in KB or new then add q' to new

$\phi \leftarrow$ UNIFY(q', α)

if ϕ is not fail then return ϕ

add new to KB

return false

Output :-

Enter your FOL Statements :
>American(p) ∧ Weapon(q) ∧ Sells (p, q, r) ∧ Hostile(r) => Criminal(p)
>∃x Owns (A, x) ∧ Missile(x)
>Owns(A, T1)
>Missile (T1)
>∀x Missile(x) ∧ Owns (A, x) => Sells (Robert, x, A)
>Missile (x) => Weapon(x)
>∀x Enemy (x, America) => Hostile(x)
>American(Robert)
>Enemy (A, America)
>done

Enter the query to prove : Criminal(Robert)
Proven : Criminal(Robert)

Example :-

It is a crime for an American to sell weapons to hostile nations
Let's say p, q and r are variable
American
Country a has some missiles.
&
Existential Instantiation, Introducing a new constant T1 :

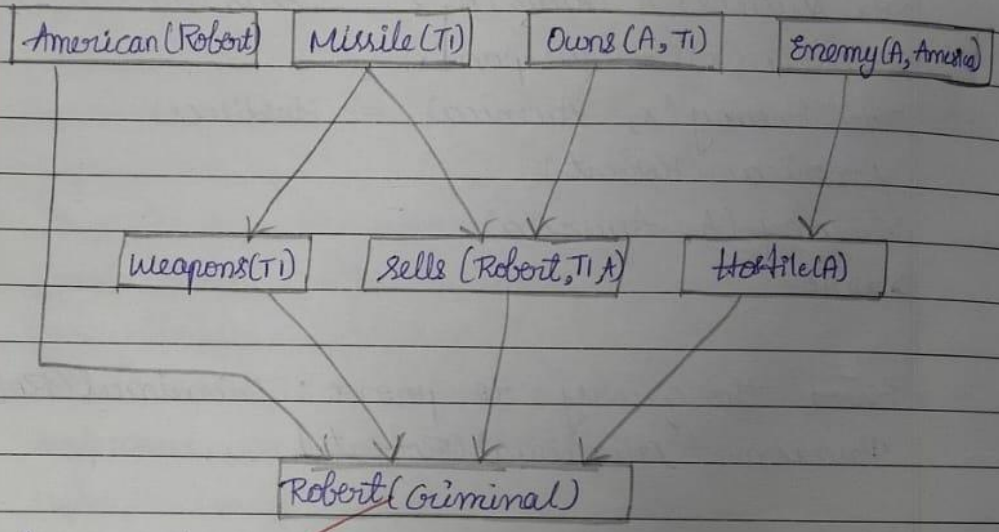
All of the missiles were sold to country A by Robert

Missiles are weapons

Enemy of America is known as hostile

Robert is an American

The country A, an enemy of America.



American(p) ∧ weapon(q) ∧ sells(p, q, x) ∧ Hostilion(x)
⇒ Criminal

3/10/2024

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 = True

        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.facts


# 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}")
```

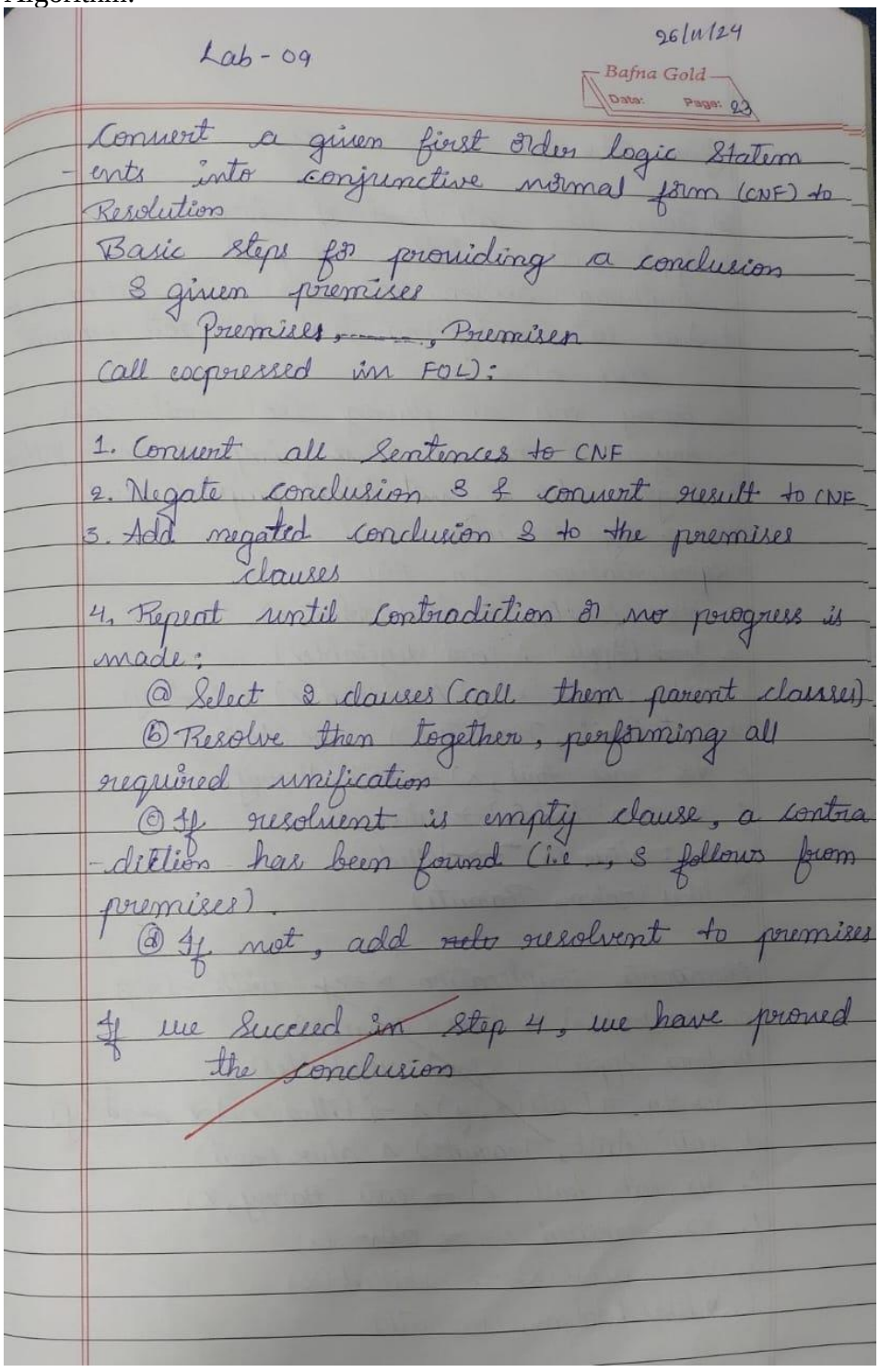
Output:

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

Final facts:
{'Sells(Robert, m1, CountryA)', 'Criminal(Robert)', 'Owns(CountryA, m1)', 'American(Robert)', 'Hostile(CountryA)', '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 - 09

26/11/24

Bafna Gold

Date: Page: 23

Convert a given first order logic Statements into conjunctive normal form (CNF) to Resolution

Basic steps for providing a conclusion S given premises

Premises, ..., Premisen

Call expressed in FOL):

1. Convert all Sentences to CNF
2. Negate conclusion S & convert result to CNF
3. Add negated conclusion S to the premises 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 unification
   - (c) If resolvent is empty clause, a contradiction has been found (i.e., S follows from premises).
   - (d) If not, add ~~rule~~ resolvent to premises

If we Succeed in Step 4, we have proved the conclusion

Give the KB & premises:

a. John likes all kind of food
b. Apple and vegetables are food
c. Anything anyone eats and not killed is food
d. Anil eats ~~everything~~ that Anil eats peanuts & still alive
e. Harry eats everything that Anil eats
f. Anyone who is alive implies not killed
g. Anyone who is not killed implies alive

Prove by resolution that:

Representation in FOL

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. $likes(John, Peanuts)$

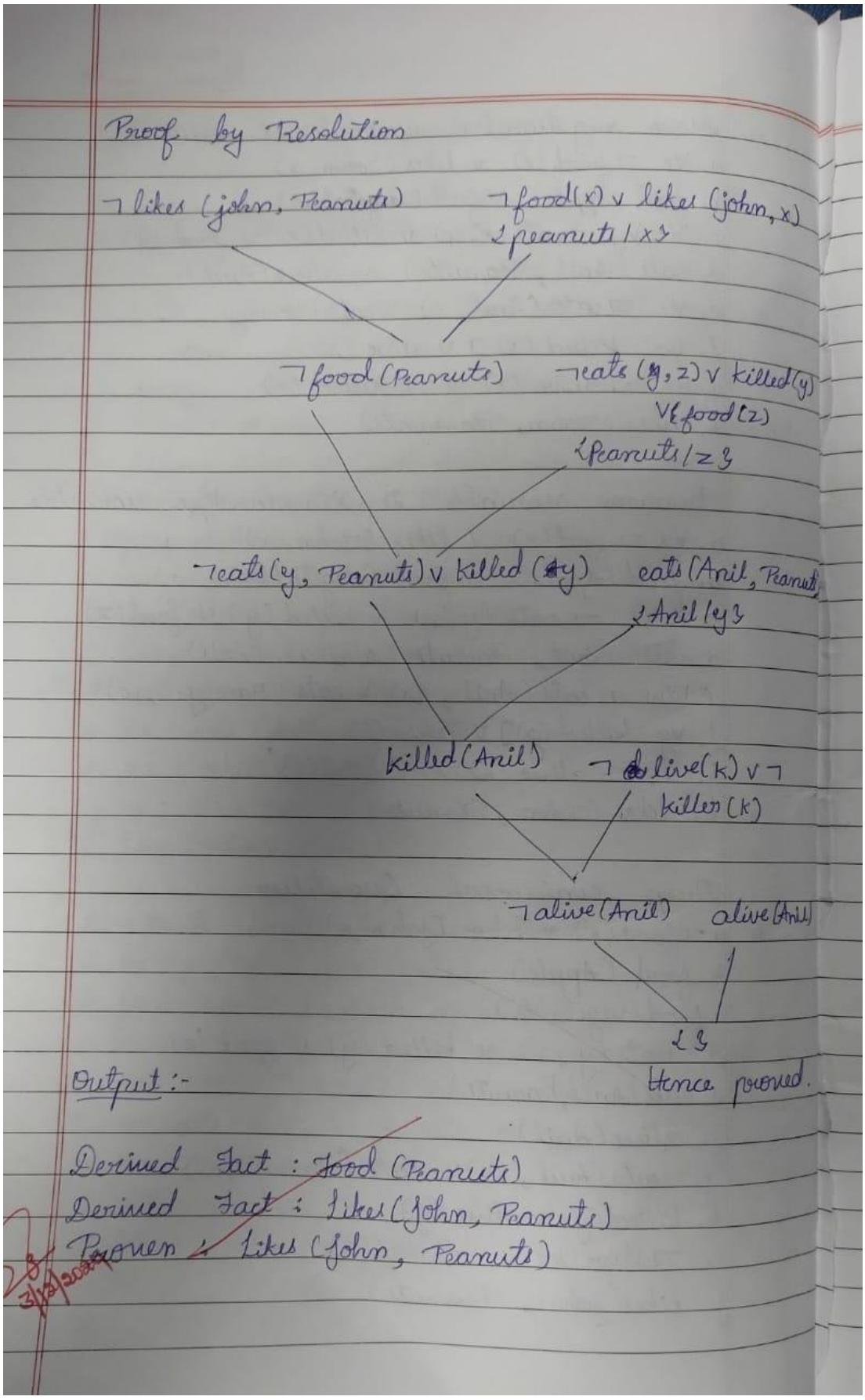
Eliminate implication $\alpha \Rightarrow \beta$ with $\neg \alpha \beta$

a. $\forall x \neg food(x) \vee likes(John, x)$
b. $food(Apple) \wedge food(vegetables)$
c. $\forall x \forall y \neg [eats(x, y) \wedge \neg killed(x)] \vee food(y)$
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. $likes(John, Peanuts)$

Move negation (¬) inwards & rewrite
a. $\forall x \neg food(x) \vee likes(John, x)$
b. $food(Apple) \wedge food(vegetables)$
c. $\forall x \forall y \neg eats(x, y) \vee killed(x) \vee food(y)$
d. $eats(Anil, Peanuts) \wedge alive(Anil)$
e. $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$
f. $\forall x killed(x)] \vee alive(x)$
g. $\forall x \neg alive(x) \vee \neg killed(x)$
h. $likes(John, Peanuts)$

Rename variable or Standardize variable
a. $\forall x \neg food(x) \vee likes(John, x)$
b. $food(Apple) \wedge food(vegetables)$
c. $\forall y \forall z \neg eats(y, z) \vee killed(y) \vee food(z)$
d. $eats(Anil, Peanuts) \wedge alive(Anil)$
e. $\forall w \neg eats(Anil, w) \vee eats(Harry, w)$
f. $\forall g killed(g)] \vee alive(g)$
g. $\forall k \neg alive(k) \vee \neg killed(k)$
h. $likes(John, Peanuts)$

Drop universal Quantifier
6. a. $\neg food(x) \vee likes(John, x)$
b. $food(Apple)$
c. $food(vegetable)$
d. $\neg eats(y, z) \vee killed(y) \vee food(z)$
e. $eats(Anil, Peanuts)$
f. $alive(Anil)$
g. $\neg eats(Anil, w) \vee eats(Harry, w)$
h. $killed(g) \vee alive(g)$
i. $\neg alive(k) \vee \neg killed(k)$
j. $likes(John, Peanuts)$


Proof by Resolution
¬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)
¬alive (Anil)
alive (Anil)
{}
Output :-
Hence proved.
Derived Fact : Food (Peanuts)
Derived Fact : Likes (John, Peanuts)
Proven : 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'}")
```

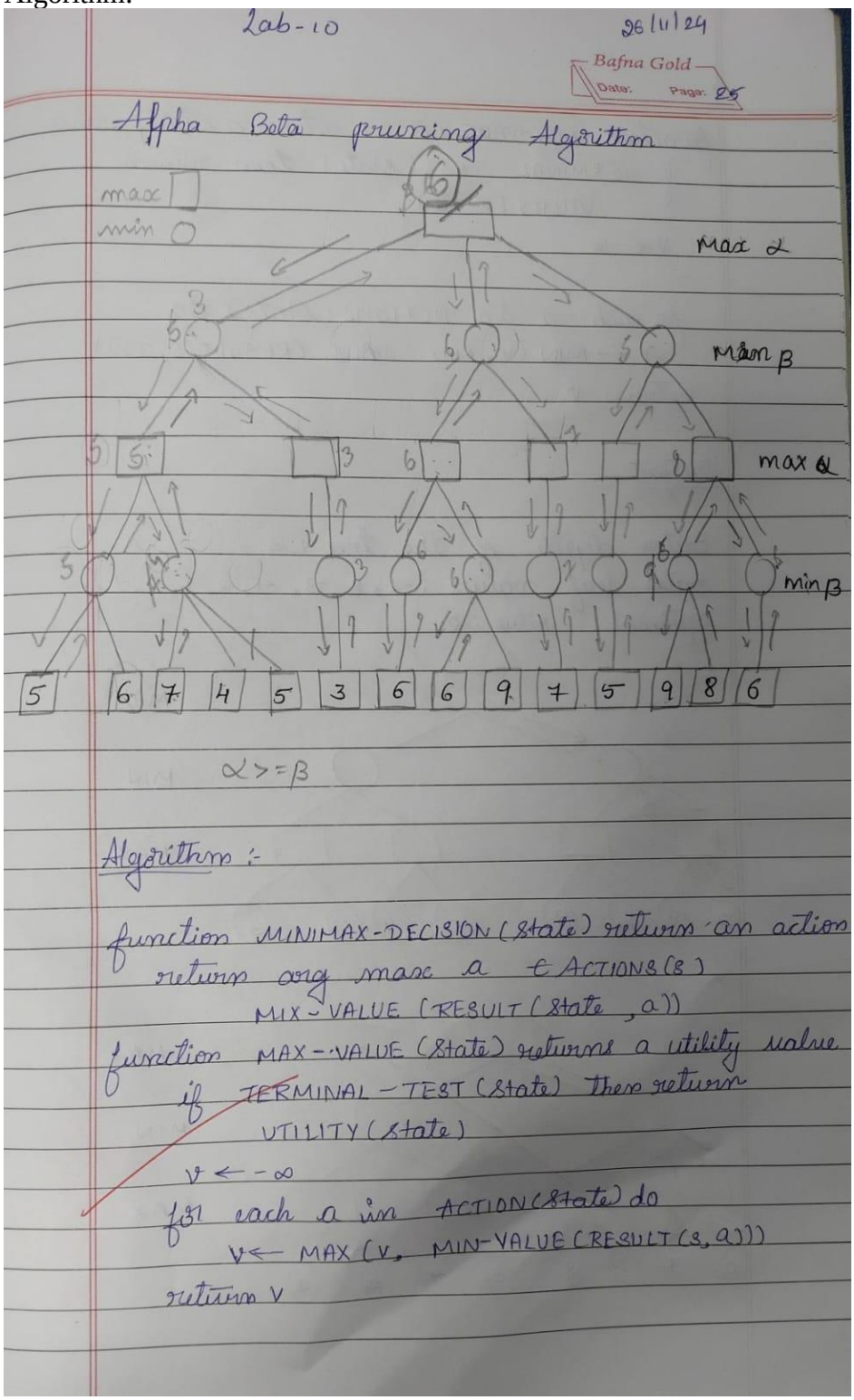
Output:

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

Program 10

Implement Alpha-Beta Pruning.

Algorithm:



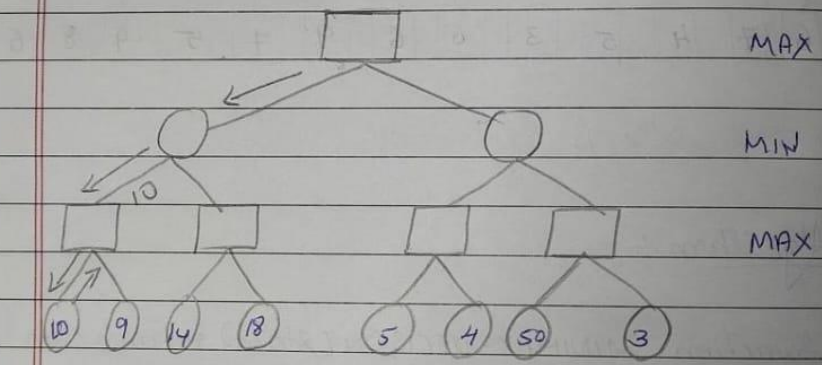
function MIN-VALUE (State) return a utility value
if TERMINAL-TEST (state) then return
UTILITY (State)

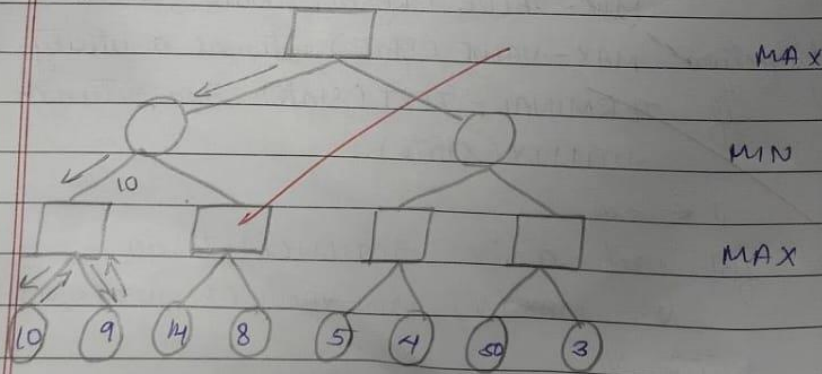
$v \leftarrow \infty$

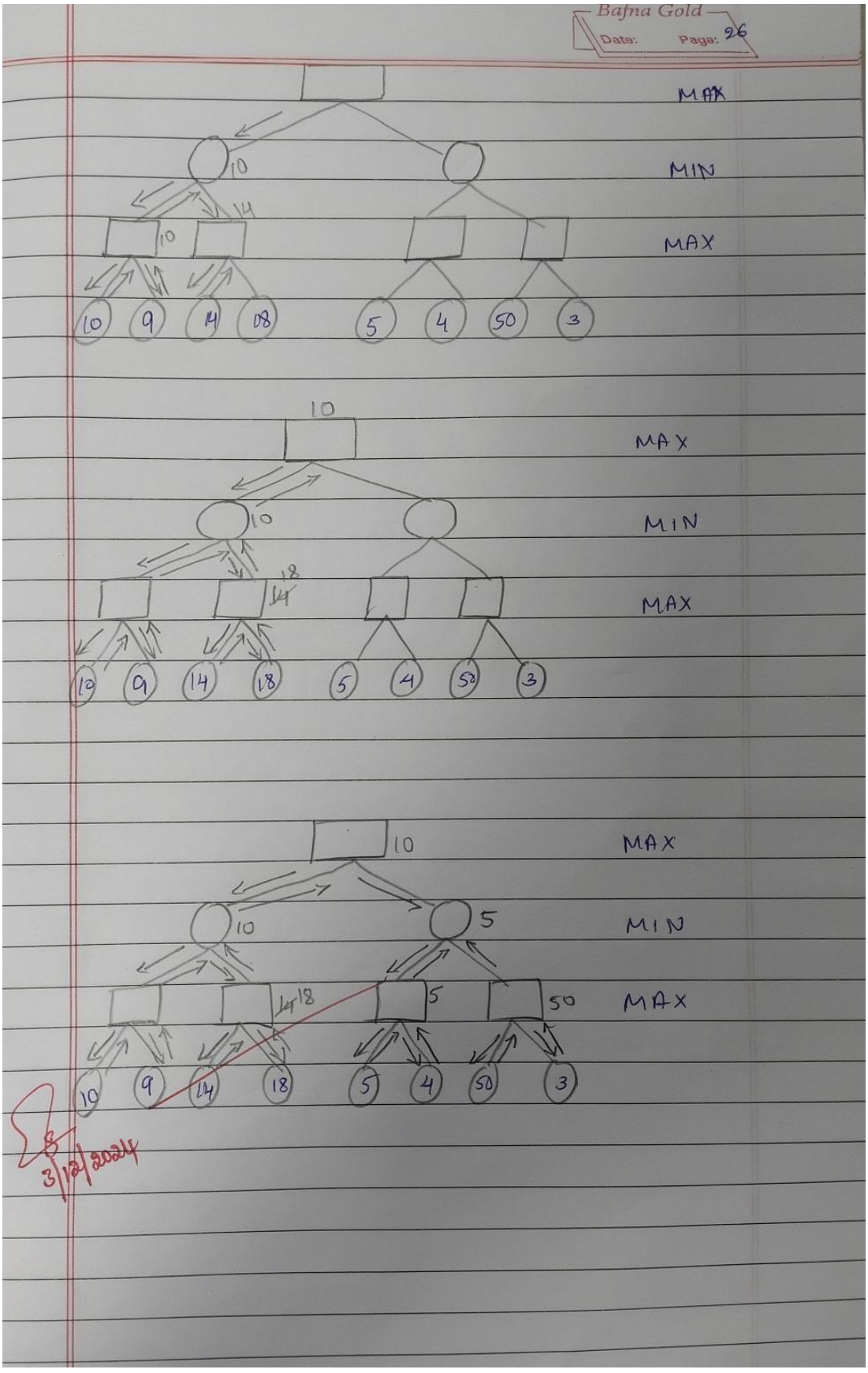
for each a in ACTIONS (state) do
$v \leftarrow MIN(v, MAX\text{-}VALUE(RESULT(s,a)))$
return v

Output

Enter depth of the tree : 3
Enter leaf values : -1, 8, -3, -1, 2, 1, 3, 4
Optimal value : 2






Bafna Gold
Date:
Page: 26
MAX
MIN
MAX
10
14
10
10 9 14 18 5 4 50 3
10
MAX
MIN
MAX
10
10
18
14
10 9 14 18 5 4 50 3
10
MAX
MIN
MAX
10
5
10
14 18
5
50
10 9 14 18 5 4 50 3
3/12/2024

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
```