VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Shashank Patel C J (1BM22CS255)

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 **Shashank Patel C J (1BM22CS255),** 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.

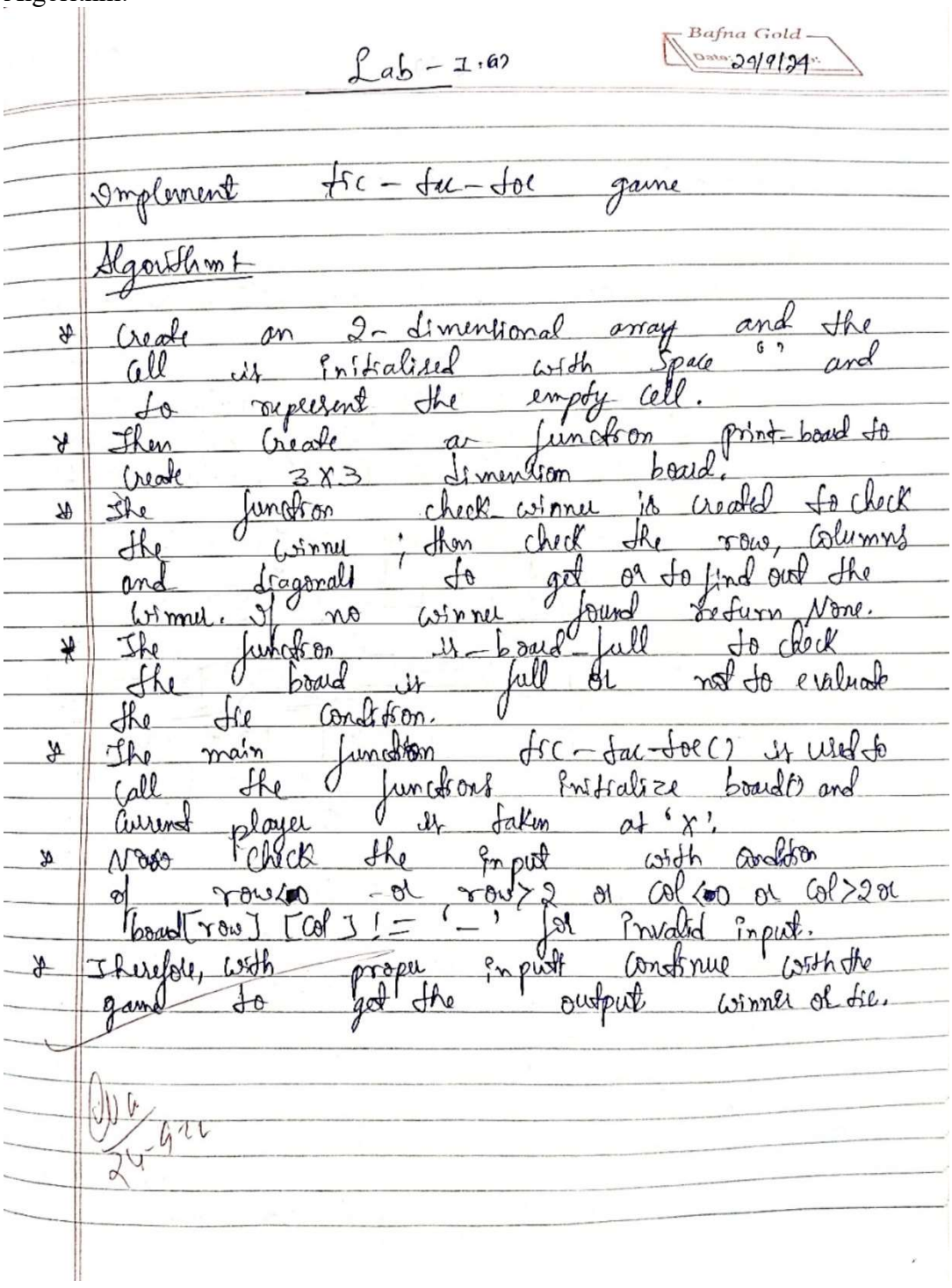
Sonika Sharma D Assistant Professor Department of CSE, BMSCE	Dr. Jyothi S Nayak Professor & HOD Department of CSE, BMSCE

Index

GitHub Link:
https://github.com/SP212004/AI-Lab

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

Algorithm:



Lab - 1.a)

Bafna Gold
Date: 29/9/24

Implement tic-tac-toe game

Algorithm

* Create an 2-dimensional array and the all is initialised with Space ' ' and to represent the empty cell.
* Then Create a function print_board to create 3X3 dimension board.
* The function check_winner is created to check the winner ; then check the row, columns and diagonals to get or to find out the winner. If no winner found return None.
* The function is_board_full to check the board is full or not to evaluate the tie condition.
* The main function tic-tac-toe() is used to call the functions initialize board() and current player is taken as 'x'.
* Now check the input with condition of row<0 or row>2 or col<0 or col>2 or board[row][col] != ' ' for invalid input.
* Therefore, with proper input continue with the game to get the output winner or tie.

01/10/24

Lab-2(b)

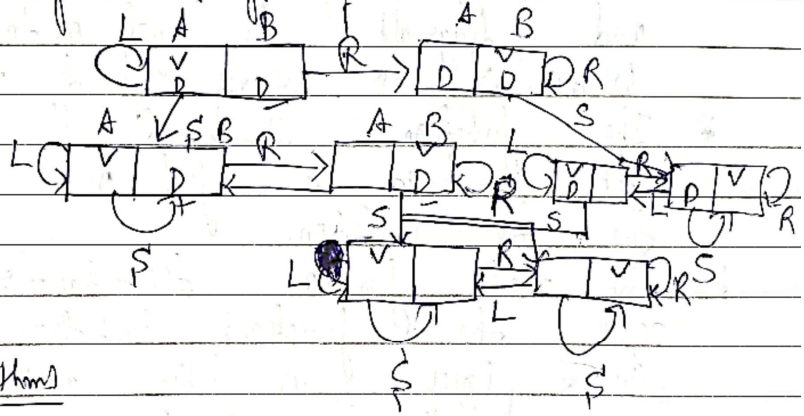
Implement Vacuum world cleaner.

Pseudo code

function REFLEX-VACUUM-AGENT([Location, status])
returns an action.

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

State-space Diagram of Vacuum world.



* Algorithms

* The function implemented in the cleaner, print_f and the main function.
* The main function contains the floor list where the user enters the row and columns.
* It randomly 'I' as dirty and '0' as clean.
* In the clean function It contains row, col and the even row Iterate from left to right and add row

Bafna Gold
Date: Page:

Iterates through right to left.

* The printf contains to update each iteration and evaluate the status and each operation or action.
* Also the '1's is changed to '0' and "Goal is achieved".

SS
11/10/24.

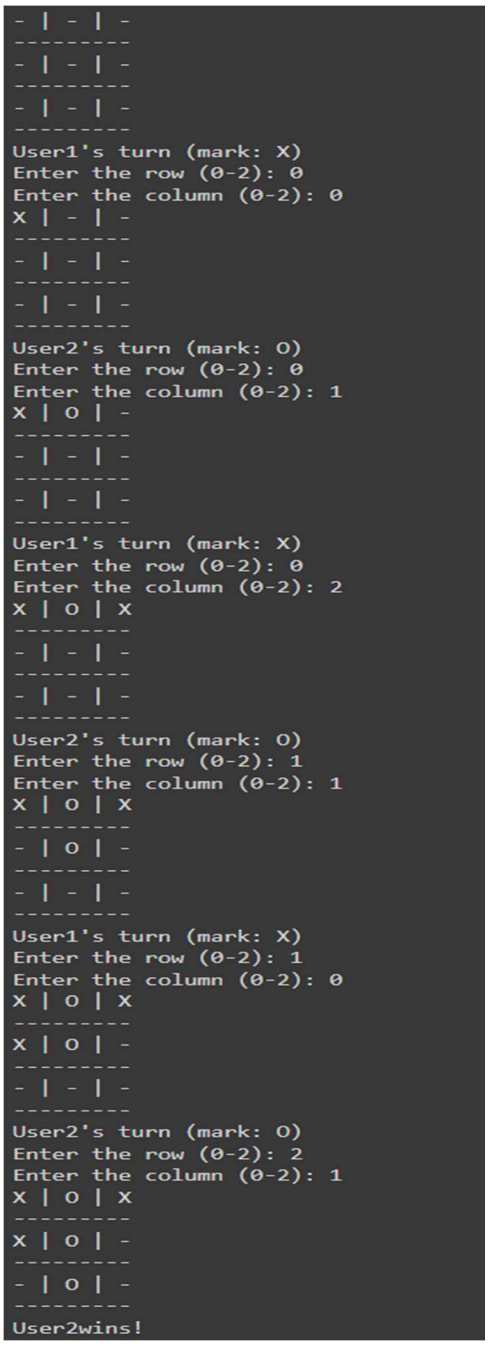
Code:
1: Tic - Tac - Toe

```
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)

def check_winner(board):
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != '-':
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != '-':
            return board[0][i]
    if board[0][0] == board[1][1] == board[2][2] != '-':
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != '-':
        return board[0][2]
    return None
def is_board_full(board):
    return all(cell != '-' for row in board for cell in row)

def play_game():
    board = [['-' for i in range(3)] for j in range(3)]
    current_player = 'X'
    player_names = {'X': 'User1', 'O': 'User2'}
    while True:
        print_board(board)
        print(f"{player_names[current_player]}'s turn (mark: {current_player})")
        row = input("Enter the row (0-2): ")
        col = input("Enter the column (0-2): ")
        if not (row.isdigit() and col.isdigit()):
            print("Invalid input. Please enter numbers between 0 and 2.")
            continue
        row = int(row)
        col = int(col)
        if 0 <= row < 3 and 0 <= col < 3:
            if board[row][col] == '-':
                board[row][col] = current_player
            else:
                print("Cell already taken. Try again.")
            winner = check_winner(board)
            if winner:
                print_board(board)
                print(player_names[winner] +"wins!")
                break
            if is_board_full(board):
```

```
            print_board(board)
            print("It's a draw!")
            break
        current_player = 'O' if current_player == 'X' else 'X'
    else:
        print("Invalid input. Please enter numbers between 0 and 2.")
play_game()
```

Output:

Win :


- | - | -

- | - | -

- | - | -

User1's turn (mark: X)
Enter the row (0-2): 0
Enter the column (0-2): 0
X | - | -

- | - | -

- | - | -

User2's turn (mark: O)
Enter the row (0-2): 0
Enter the column (0-2): 1
X | O | -

- | - | -

- | - | -

User1's turn (mark: X)
Enter the row (0-2): 0
Enter the column (0-2): 2
X | O | X

- | - | -

- | - | -

User2's turn (mark: O)
Enter the row (0-2): 1
Enter the column (0-2): 1
X | O | X

- | O | -

- | - | -

User1's turn (mark: X)
Enter the row (0-2): 1
Enter the column (0-2): 0
X | O | X

X | O | -

- | - | -

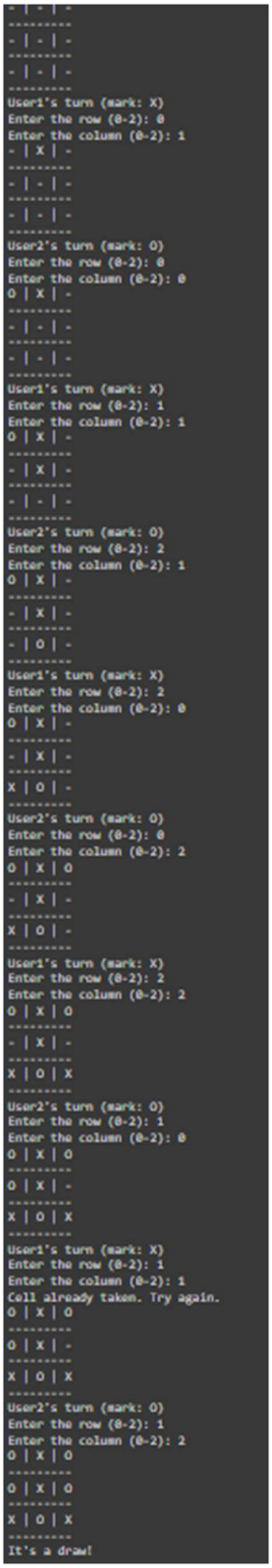
User2's turn (mark: O)
Enter the row (0-2): 2
Enter the column (0-2): 1
X | O | X

X | O | -

- | O | -

User2wins!

Draw :


- | - | -

- | - | -

- | - | -

User1's turn (mark: X)
Enter the row (0-2): 0
Enter the column (0-2): 1
- | X | -

- | - | -

- | - | -

User2's turn (mark: O)
Enter the row (0-2): 0
Enter the column (0-2): 0
O | X | -

- | - | -

- | - | -

User1's turn (mark: X)
Enter the row (0-2): 1
Enter the column (0-2): 1
O | X | -

- | X | -

- | - | -

User2's turn (mark: O)
Enter the row (0-2): 2
Enter the column (0-2): 1
O | X | -

- | X | -

- | O | -

User1's turn (mark: X)
Enter the row (0-2): 2
Enter the column (0-2): 0
O | X | -

- | X | -

X | O | -

User2's turn (mark: O)
Enter the row (0-2): 0
Enter the column (0-2): 2
O | X | O

- | X | -

X | O | -

User1's turn (mark: X)
Enter the row (0-2): 2
Enter the column (0-2): 2
O | X | O

- | X | -

X | O | X

User2's turn (mark: O)
Enter the row (0-2): 1
Enter the column (0-2): 0
O | X | O

O | X | -

X | O | X

User1's turn (mark: X)
Enter the row (0-2): 1
Enter the column (0-2): 1
Cell already taken. Try again.
O | X | O

O | X | -

X | O | X

User2's turn (mark: O)
Enter the row (0-2): 1
Enter the column (0-2): 2
O | X | O

O | X | O

X | O | X

It's a draw!

2. Vacuum Cleaner :

```
initial_states = []
cost = 0
goal_state = ["A", 0, "B", 0]

def vacuum_world(location):
    global cost
    current_state = ["A", initial_states[0], "B", initial_states[1]]
    print("Current state:",current_state)
    while current_state[1] != 0 or current_state[3] != 0:
        if location == "A":
            if current_state[1] == 1:
                print("Location A is dirty.")
                print("Cleaning Location A...")
                cost += 1
                current_state[1] = 0
                print("Cost for suck:", cost)
            print("Moving to B...")
            location = "B"
        elif location == "B":
            if current_state[3] == 1:
                print("Location B is dirty.")
                print("Cleaning Location B...")
                cost += 1
                current_state[3] = 0
                print("Cost for suck:", cost)
            print("Moving to A...")
            location = "A"
        print("Current state:",current_state)
    print("Final state:", current_state)
    print("Total cost:", cost)

for room in ['A', 'B']:
    state = input(f"Is Room {room} dirty? (1 for dirty, 0 for clean): ").strip()
    while state not in ['0', '1']:
        print("Invalid input. Please enter 1 for dirty or 0 for clean.")
        state = input(f"Is Room {room} dirty? (1 for dirty, 0 for clean): ").strip()
    initial_states.append(int(state))

location = input("Enter the starting location of the vacuum cleaner (A or B): ").strip().upper()
while location not in ['A', 'B']:
    print("Invalid location. Please enter 'A' or 'B'.")
    location = input("Enter the starting location of the vacuum cleaner (A or B): ").strip().upper()

vacuum_world(location)
```

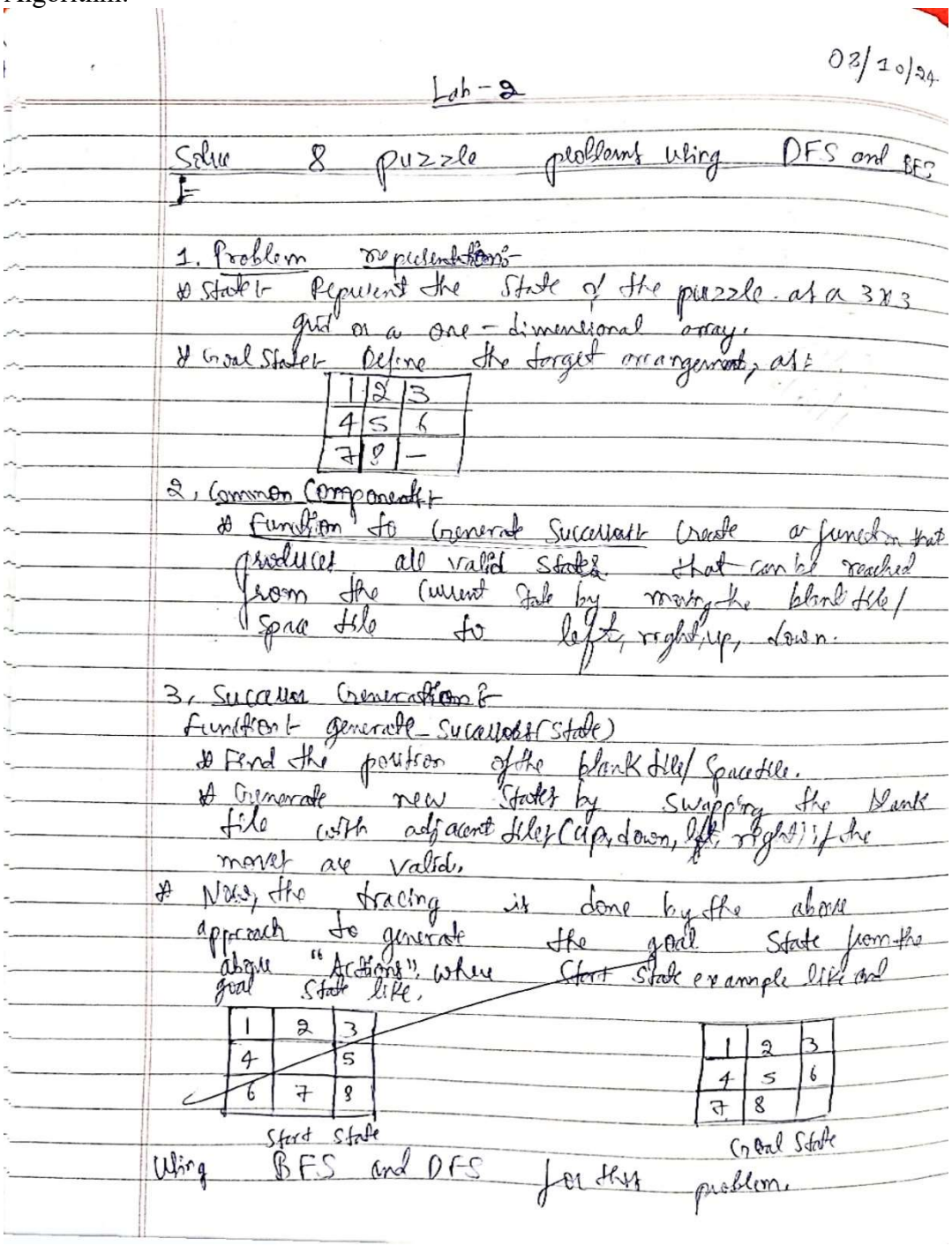
Output :

```
Is Room A dirty? (1 for dirty, 0 for clean): 1
Is Room B dirty? (1 for dirty, 0 for clean): 1
Enter the starting location of the vacuum cleaner (A or B): A
Current state: ['A', 1, 'B', 1]
Location A is dirty.
Cleaning Location A...
Cost for suck: 1
Moving to B...
Current state: ['A', 0, 'B', 1]
Location B is dirty.
Cleaning Location B...
Cost for suck: 2
Moving to A...
Current state: ['A', 0, 'B', 0]
Final state: ['A', 0, 'B', 0]
Total cost: 2
```

Program 2

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

Algorithm:



Lab - 2

03/10/24

Solve 8 puzzle problems using DFS and BFS

1. Problem representation
   - State: Represent the state of the puzzle as a 3x3 grid or a one-dimensional array.
   - Goal State: Define the target arrangement, as:

1	2	3
4	5	6
7	8	-

2. Common Components
   - Function to Generate Successors: Create a function that produces all valid states that can be reached from the current state by moving the blank tile/space tile to left, right, up, down.

3. Successor Generation
   Function generate_successors(state)
   - Find the position of the blank tile/space tile.
   - Generate new states by swapping the blank tile with adjacent tiles (up, down, left, right) if the moves are valid.

Now, the tracing is done by the above approach to generate the goal state from the above "Actions". Where start state example like and goal state like.

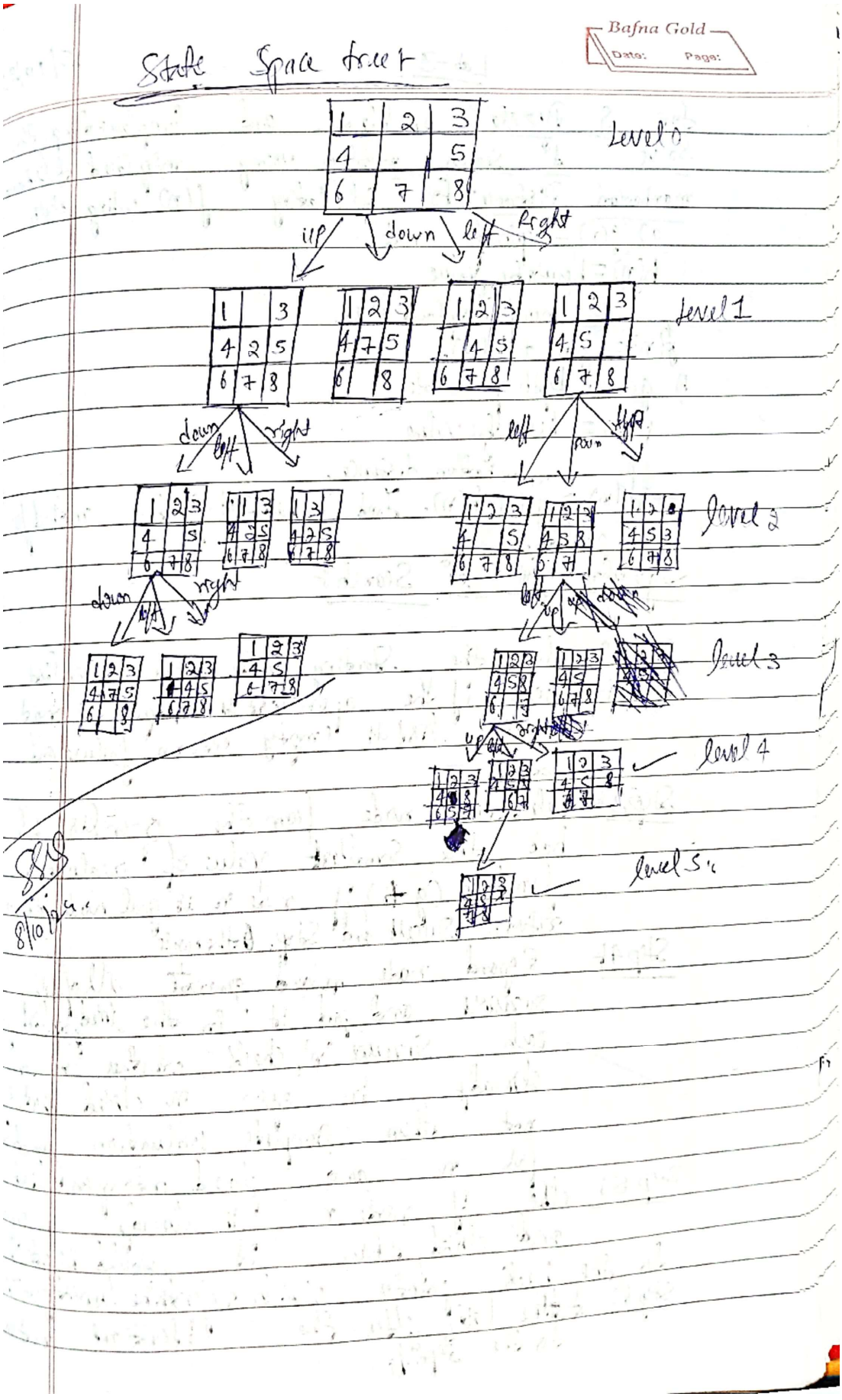
1	2	3
4		5
6	7	8

Start State

1	2	3
4	5	6
7	8	

Goal State

Using BFS and DFS for this problem.


Bafna Gold
Date: Page:
State Space tree
Level 0
up
down
left
Right
level 1
down
left
right
left
down
up
level 2
level 3
level 4
level 5

Bafna Gold Date: Page:

22/10/24

Lab - 4.(a)

8 puzzle problem and solving using Iterative deepening approach. (search).

Algorithm

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 1.
7. If we have reached all leaf (bottom) nodes, the goal node doesn't exist.

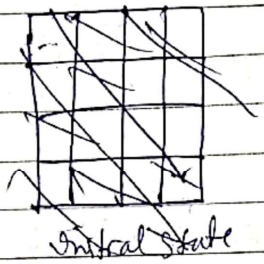
function Iterative_deepening_Search(problem) returns a solution or failure

for depth = 0 to ∞ do

result ← depth-limited-Search(problem, depth)

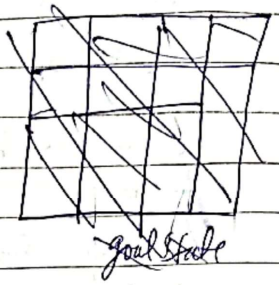
If result ≠ cutoff then return result.

State Space tree Non heuristic approach



2	8	3
1	6	4
7		5

Initial state



level 0

2	8	3
1	6	4
7		5

Initial State

Level = 0
depth = 0

1	2	3
4	5	6
7	8	

Goal State

level 1

limit = 2

depth = 0
level = 0

2	8	3
1	6	4
7		5

L

2	8	3
1	6	4
	7	5

U

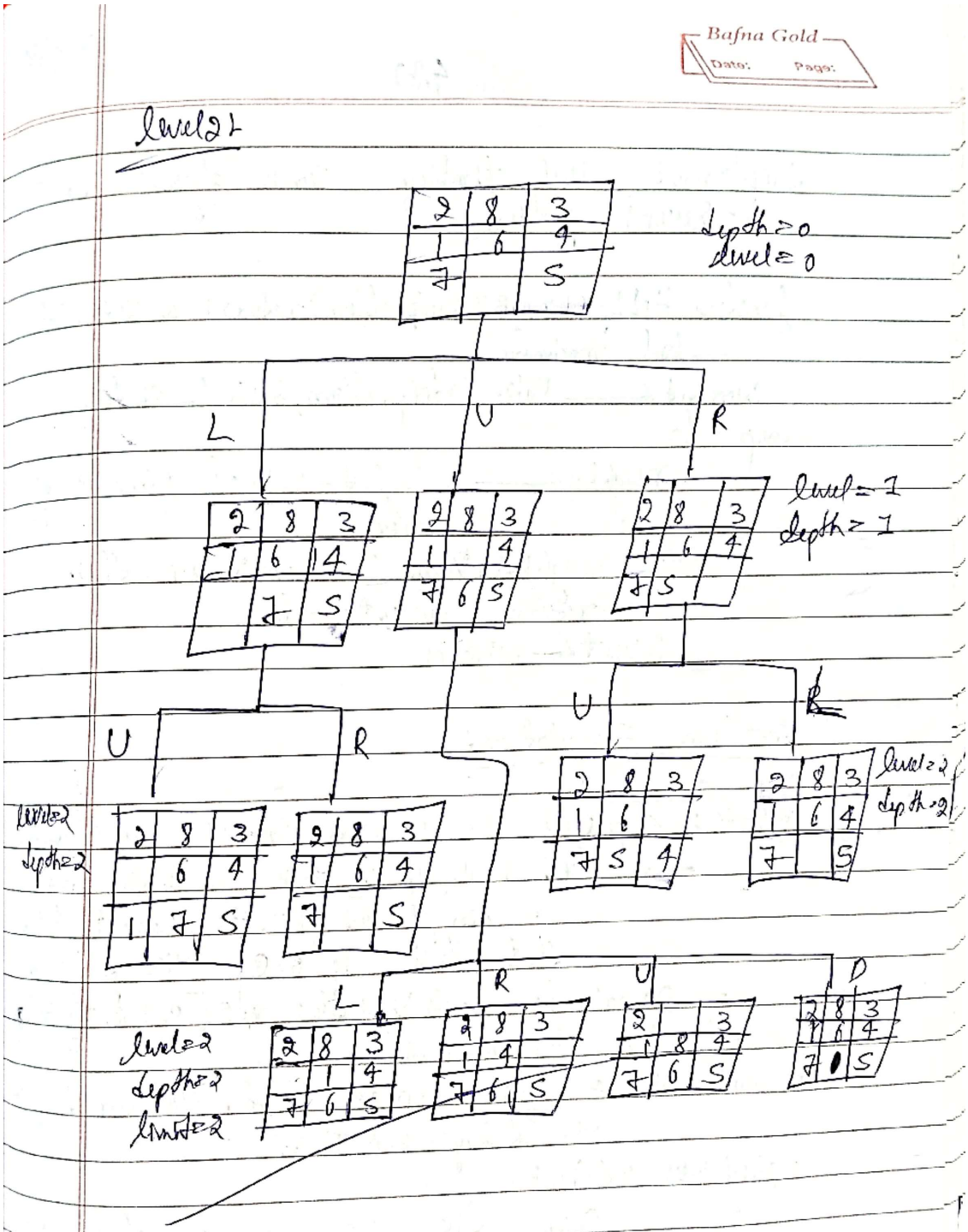
2	8	3
1		4
7	6	5

R

2	8	3
1	6	4
7	5	

level = 1
depth = 1

level 2 :



Code :
1: DFS

```
GOAL_STATE = [[1, 2, 3], [4, 5, 6], [7, 8, None]]
MOVES = [(-1, 0), (1, 0), (0, -1), (0, 1)]

def generate_successors(state):
    successors = []
    blank_i, blank_j = next((i, j) for i in range(3) for j in range(3) if state[i][j] is None)

    for di, dj in MOVES:
        new_i, new_j = blank_i + di, blank_j + dj
        if 0 <= new_i < 3 and 0 <= new_j < 3:
            new_state = [row[:] for row in state]
            new_state[blank_i][blank_j], new_state[new_i][new_j] = new_state[new_i][new_j],
new_state[blank_i][blank_j]
            successors.append(new_state)

    return successors

def dfs(initial_state):
    stack = [(initial_state, 0)]
    visited = {tuple(map(tuple, initial_state))}

    while stack:
        current_state, moves = stack.pop()

        if current_state == GOAL_STATE:
            print_state(current_state)
            return moves

        for next_state in generate_successors(current_state):
            state_tuple = tuple(map(tuple, next_state))
            if state_tuple not in visited:
                visited.add(state_tuple)
                stack.append((next_state, moves + 1))
    return -1

def print_state(state):
    print("Goal State Reached:")
    for row in state:
        print(' '.join(str(x) if x is not None else '_' for x in row))
    print()

if __name__ == "__main__":
    initial_state = [[1, 2, 3], [None, 4, 6], [7, 5, 8]]
    moves_to_goal = dfs(initial_state)
```

```
if moves_to_goal != -1:
    print("DFS Solution Found: True")
    print(f"Number of moves to reach the goal state: {moves_to_goal}")
else:
    print("DFS Solution Found: False")
```

Output :

```
Goal State Reached:
1 2 3
4 5 6
7 8 _

DFS Solution Found: True
Number of moves to reach the goal state: 49285
```

Code :

2 : Iterative deepening search

```
class PuzzleState:
    def __init__(self, board, depth=0):
        self.board = board
        self.empty_tile_pos = self.find_empty_tile()
        self.depth = depth

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

    def is_goal(self, goal_state):
        return self.board == goal_state

    def generate_successors(self):
        successors = []
        x, y = self.empty_tile_pos
        moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for dx, dy in moves:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = [row[:] for row in self.board]
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
                successors.append(PuzzleState(new_board, self.depth + 1))
        return successors

def depth_limited_search(state, goal_state, depth):
    if state.is_goal(goal_state):
        return state
    if depth == 0:
        return None

    for successor in state.generate_successors():
        result = depth_limited_search(successor, goal_state, depth - 1)
        if result is not None:
            return result

    return None

def iterative_deepening_search(initial_state, goal_state, max_depth):
    for depth_limit in range(max_depth + 1):
```

```
        result = depth_limited_search(initial_state, goal_state, depth_limit)
        if result is not None:
            return True
    return False

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

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

initial_state = PuzzleState(initial_board)
max_depth = 3

if iterative_deepening_search(initial_state, goal_board, max_depth):
    print("Goal is reachable within the maximum depth.")
else:
    print("Goal is not reachable within the maximum depth.")
```

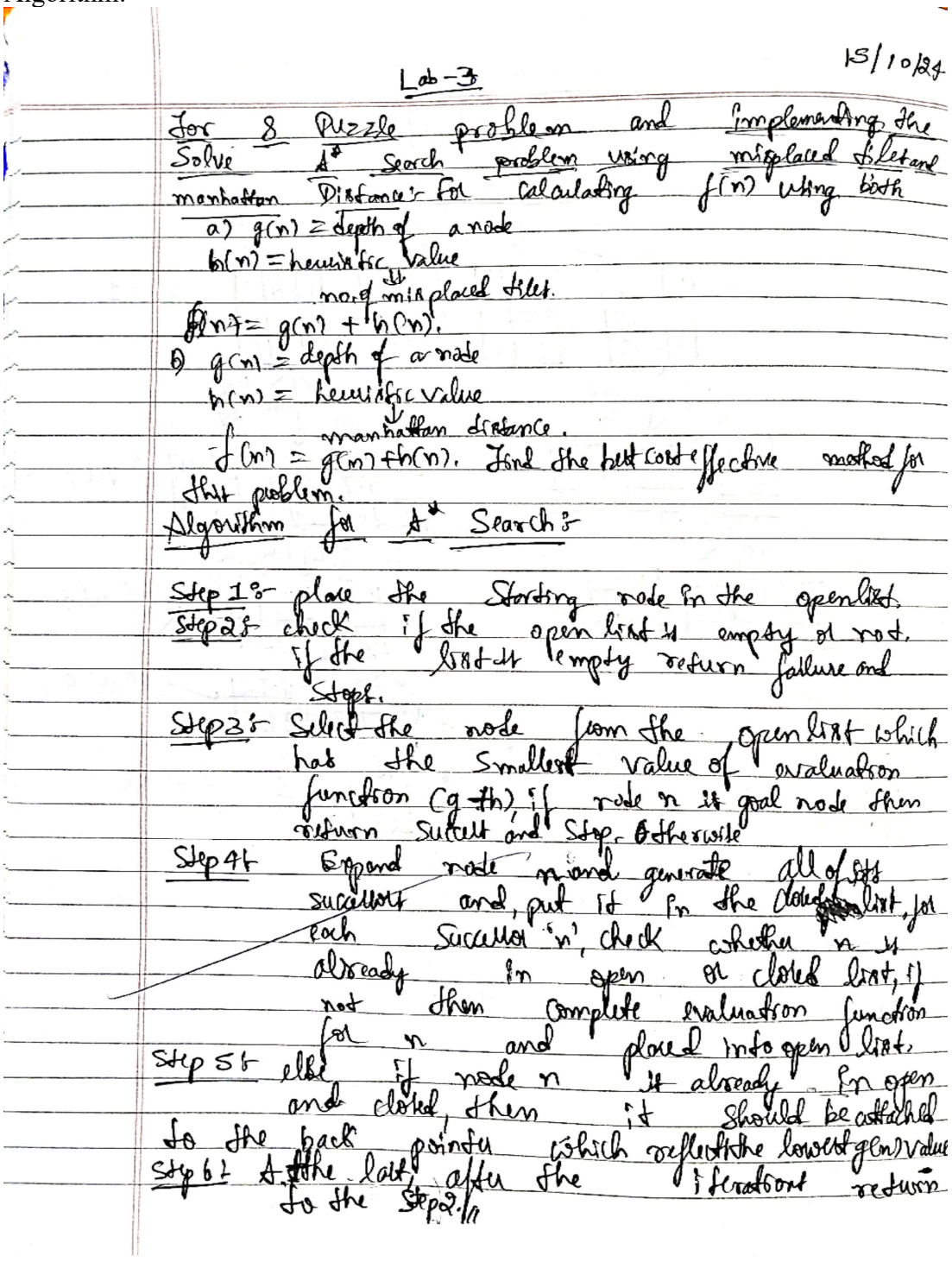
Output :

```
Goal is reachable within the maximum depth.
```

Program 3

Implement A* search algorithm

Algorithm:



15/10/24

Lab-3

For 8 Puzzle problem and implementing the Solve A* search problem using misplaced tiles and manhattan Distance for calculating f(n) using both

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). Find the best cost effective method for this problem.

Algorithm for A* Search:-

Step 1:- place the Starting node in the openlist.

Step 2:- check if the open list is empty or not. If the list is empty return failure and Stops.

Step 3:- Select the node from the open list which has the smallest value of evaluation function (g+h), if node n is goal node then return Success and Stop. Otherwise

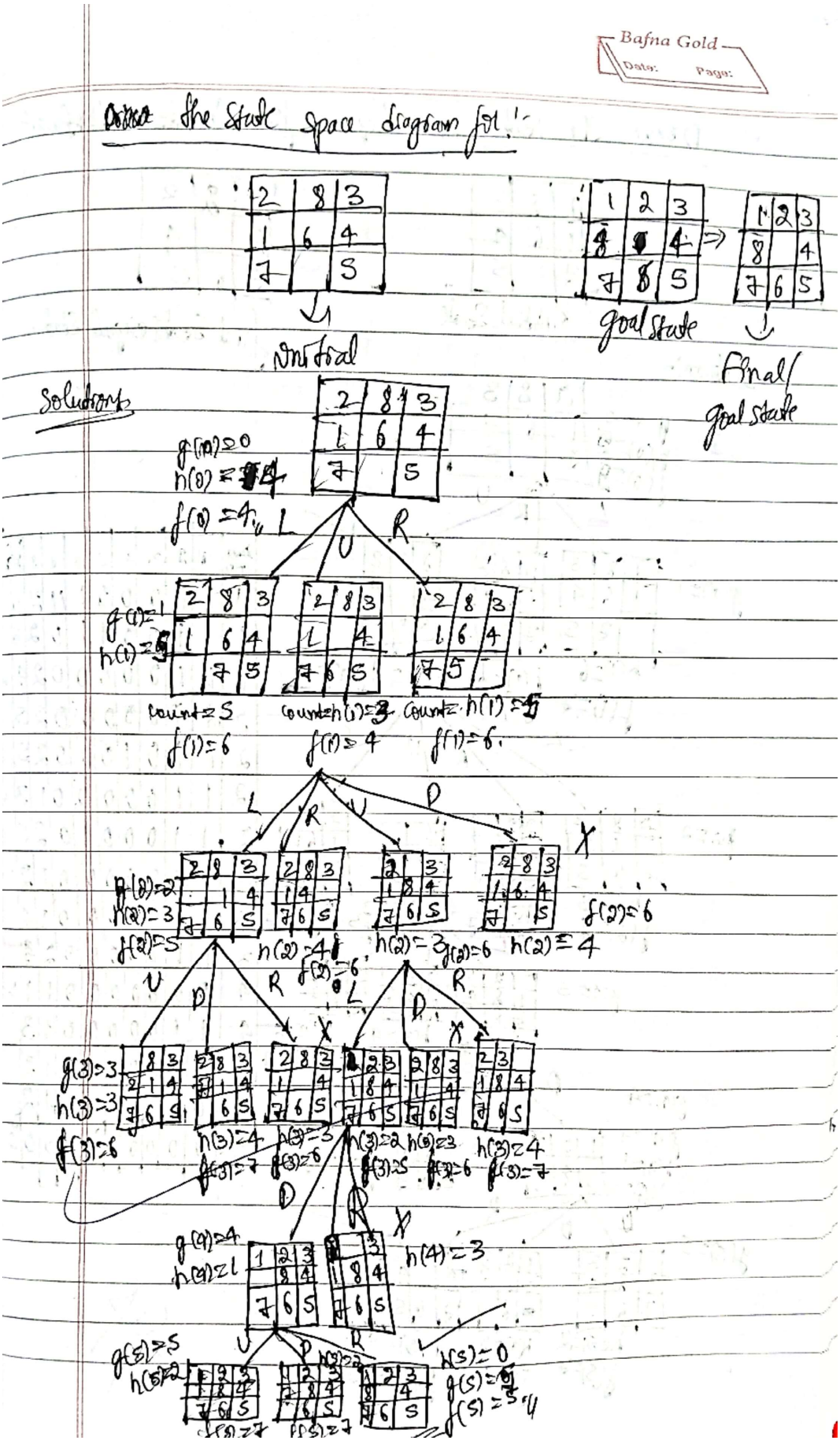
Step 4:- Expand node n and generate all of its successors and, put it in the closed list, for each Successor 'n', check whether n is already in open or closed list, if not then compute evaluation function for n and placed into open list.

Step 5:- else if node n is already in open and closed, then it should be attached to the back pointer which reflects the lowest g(n) value

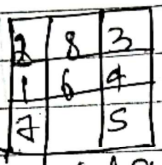
Step 6:- At the last, after the iteration return to the Step 2.

Bafna Gold
Date: Page:

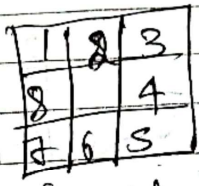
Draw the state space diagram for :-



Draw the state space diagram for Manhattan Distance:-

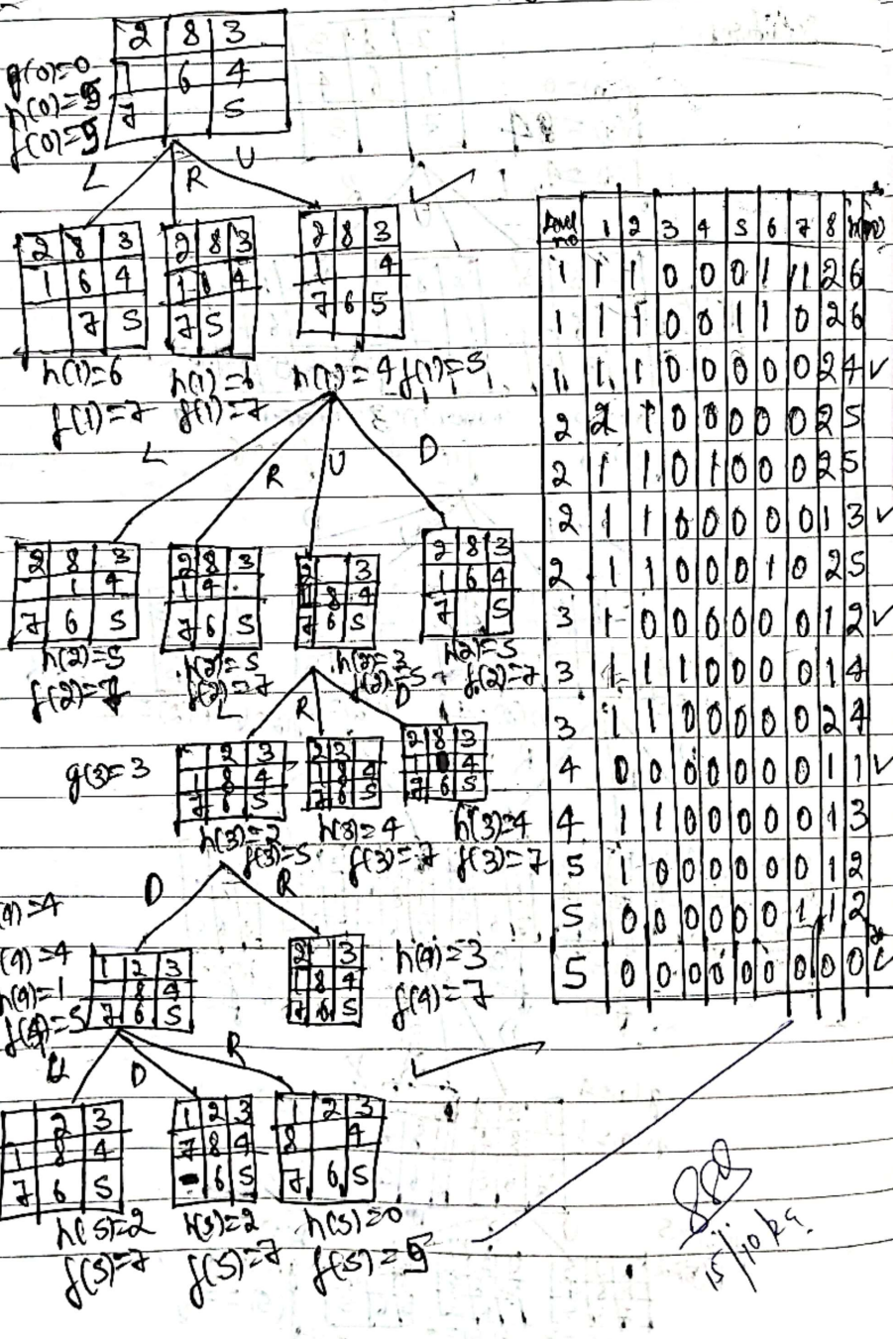


Initial State



final State (or goal State)

Solution:-



Node no	1	2	3	4	5	6	7	8	h(n)	
1	1	1	0	0	0	1	1	2	6	
1	1	1	0	0	1	1	0	2	6	
1	1	1	0	0	0	0	0	2	4	✓
2	2	1	0	0	0	0	0	2	5	
2	1	1	0	1	0	0	0	2	5	
2	1	1	0	0	0	0	0	1	3	✓
2	1	1	0	0	0	1	0	2	5	
3	1	0	0	0	0	0	0	1	2	✓
3	1	1	1	0	0	0	0	1	4	
3	1	1	0	0	0	0	0	2	4	
4	0	0	0	0	0	0	0	1	1	✓
4	1	1	0	0	0	0	0	1	3	
5	1	0	0	0	0	0	0	1	2	
5	0	0	0	0	0	0	1	1	2	
5	0	0	0	0	0	0	0	0	0	✓

15/10/24

Code:

Misplaced Tiles :

```
GOAL_STATE = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]
MOVES = [(0, 1), (1, 0), (0, -1), (-1, 0)]

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

def h1_misplaced_tiles(state):
    return sum([1 for i in range(3) for j in range(3) if state[i][j] != 0 and state[i][j] !=
GOAL_STATE[i][j]])

def generate_new_states(state):
    empty_x, empty_y = find_empty(state)
    new_states = []
    for move in MOVES:
        new_x, new_y = empty_x + move[0], empty_y + move[1]
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [row[:] for row in state]
            new_state[empty_x][empty_y], new_state[new_x][new_y] = new_state[new_x][new_y],
new_state[empty_x][empty_y]
            new_states.append(new_state)
    return new_states

def a_star_search_misplaced(initial_state):
    priority_queue = [(0 + h1_misplaced_tiles(initial_state), 0, initial_state, [])]
    visited = set()
    visited.add(tuple(map(tuple, initial_state)))

    while priority_queue:
        min_index = 0
        for i in range(len(priority_queue)):
            if priority_queue[i][0] < priority_queue[min_index][0]:
                min_index = i
        f, g, current_state, path = priority_queue.pop(min_index)

        print(f"Current State at Depth {g}: g(n)={g}, h(n)={h1_misplaced_tiles(current_state)},
f(n)={f}")
        for row in current_state:
            print(row)
        print()
```

```
        if current_state == GOAL_STATE:
            return path + [current_state], g

        for new_state in generate_new_states(current_state):
            if tuple(map(tuple, new_state)) not in visited:
                visited.add(tuple(map(tuple, new_state)))
                priority_queue.append((g + 1 + h1_misplaced_tiles(new_state), g + 1, new_state, path +
[current_state]))

    return None, None

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

solution_misplaced, total_cost = a_star_search_misplaced(initial_state)
print(f"Total cost to reach goal: {total_cost}")
```

Output :

```
Current State at Depth 0: g(n)=0, h(n)=4, f(n)=4
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

Current State at Depth 1: g(n)=1, h(n)=3, f(n)=4
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

Current State at Depth 2: g(n)=2, h(n)=3, f(n)=5
[2, 8, 3]
[0, 1, 4]
[7, 6, 5]

Current State at Depth 2: g(n)=2, h(n)=3, f(n)=5
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

Current State at Depth 3: g(n)=3, h(n)=2, f(n)=5
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

Current State at Depth 4: g(n)=4, h(n)=1, f(n)=5
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

Current State at Depth 5: g(n)=5, h(n)=0, f(n)=5
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

Total cost to reach goal: 5
```

Code :

Manhattan distance approach

```
import heapq

GOAL_STATE = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]
MOVES = [(0, 1), (1, 0), (0, -1), (-1, 0)]

def find_goal_position(tile):
    for i in range(3):
        for j in range(3):
            if GOAL_STATE[i][j] == tile:
                return i, j

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

def h_manhattan_distance(state):
    total_distance = 0
    for i in range(3):
        for j in range(3):
            tile = state[i][j]
            if tile != 0:
                goal_row, goal_col = find_goal_position(tile)
                distance = abs(i - goal_row) + abs(j - goal_col)
                total_distance += distance
    return total_distance

def generate_new_states(state):
    empty_x, empty_y = find_empty(state)
    new_states = []
    for move in MOVES:
        new_x, new_y = empty_x + move[0], empty_y + move[1]
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [row[:] for row in state]
            new_state[empty_x][empty_y], new_state[new_x][new_y] = new_state[new_x][new_y],
new_state[empty_x][empty_y]
            new_states.append(new_state)
    return new_states

def a_star_search(initial_state):
    priority_queue = []
    heapq.heappush(priority_queue, (0 + h_manhattan_distance(initial_state), 0, initial_state, []))
```

```
    visited = set()
    visited.add(tuple(map(tuple, initial_state)))

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

        h_value = h_manhattan_distance(current_state)
        f_value = g + h_value

        print(f"Current State at Depth {g}: g(n)={g}, h(n)={h_value}, f(n)={f_value}")
        for row in current_state:
            print(row)
        print()

        if current_state == GOAL_STATE:
            return path + [current_state], g

        for new_state in generate_new_states(current_state):
            state_tuple = tuple(map(tuple, new_state))
            if state_tuple not in visited:
                visited.add(state_tuple)
                heapq.heappush(priority_queue, (g + 1 + h_manhattan_distance(new_state), g + 1,
new_state, path + [current_state]))

    return None, None

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

solution_manhattan, total_cost = a_star_search(initial_state)
print(f"Total cost to reach goal: {total_cost}")
```

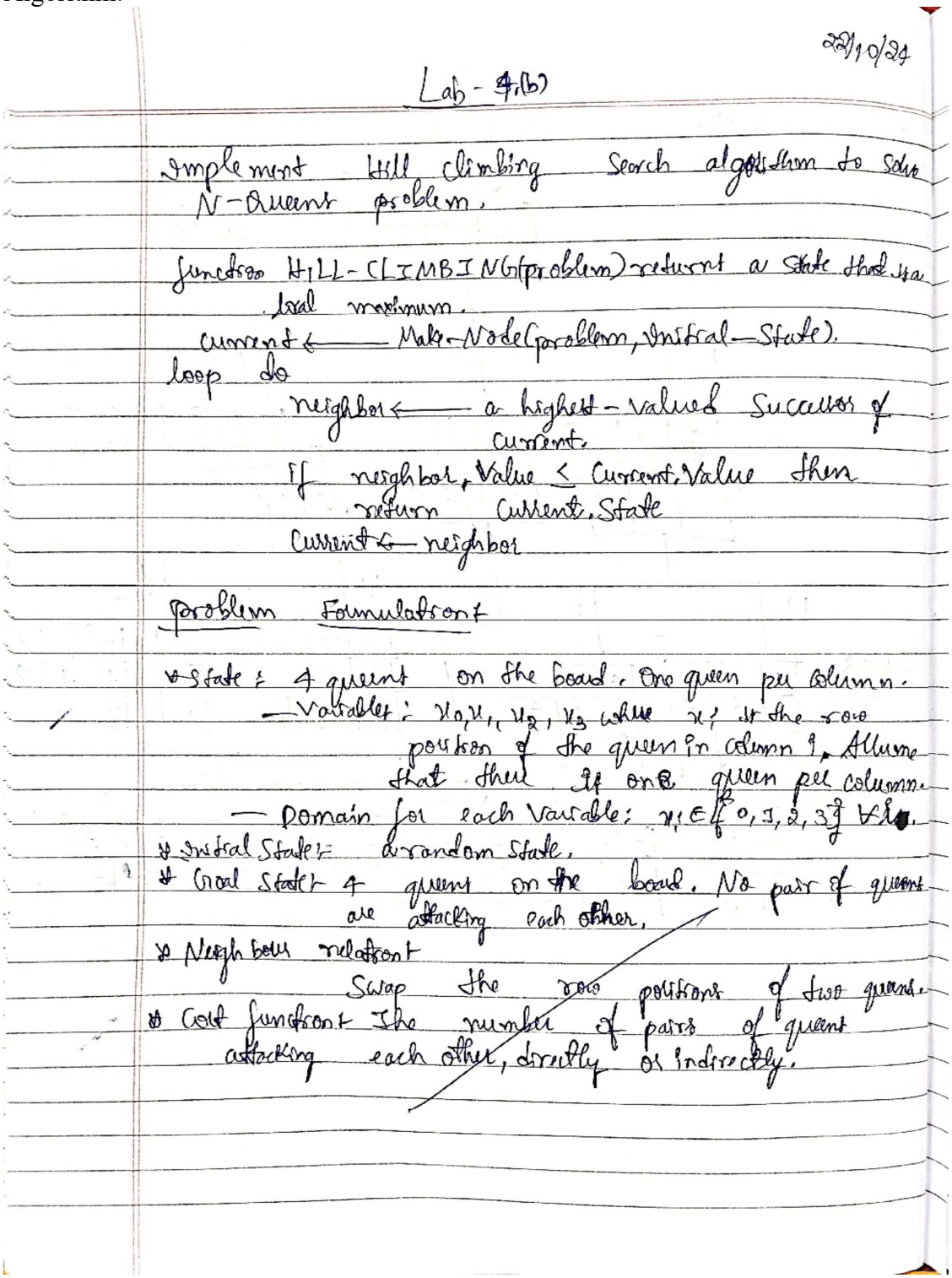
Output :

```
Current State at Depth 0: g(n)=0, h(n)=5, f(n)=5
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

Current State at Depth 1: g(n)=1, h(n)=4, f(n)=5
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

Current State at Depth 2: g(n)=2, h(n)=3, f(n)=5
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

Current State at Depth 3: g(n)=3, h(n)=2, f(n)=5
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

Current State at Depth 4: g(n)=4, h(n)=1, f(n)=5
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

Current State at Depth 5: g(n)=5, h(n)=0, f(n)=5
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

Total cost to reach goal: 5
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



22/10/24

Lab - 4(b)

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

Problem Formulation

* 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 : $x_i \in \{0, 1, 2, 3\}$ ∀i.
* Initial State : a random state.
* Goal State : 4 queens on the board. No pair of queens are attacking each other.
* Neighbours relation
Swap the row positions of two queens.
* Cost function : The number of pairs of queens attacking each other, directly or indirectly.

State Space tree

Code:

```
import random

def calculate_cost(board):
    n = len(board)
    attacks = 0

    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j]:
                attacks += 1
            if abs(board[i] - board[j]) == abs(i - j):
                attacks += 1

    return attacks

def get_neighbors(board):
    neighbors = []
    n = len(board)

    for col in range(n):
        for row in range(n):
            if row != board[col]:
                new_board = board[:]
                new_board[col] = row
                neighbors.append(new_board)

    return neighbors

def hill_climb(board):
    current_cost = calculate_cost(board)
    print("Initial board configuration:")
    print_board(board, current_cost)

    iteration = 0
    while True:
        neighbors = get_neighbors(board)
        best_neighbor = None
        best_cost = current_cost

        for neighbor in neighbors:
            cost = calculate_cost(neighbor)
            if cost < best_cost:
                best_cost = cost
                best_neighbor = neighbor
```

```
        if best_neighbor is None:
            break

        board = best_neighbor
        current_cost = best_cost
        iteration += 1
        print(f"Iteration {iteration}:")
        print_board(board, current_cost)

    return board, current_cost

def print_board(board, cost):
    n = len(board)
    display_board = [['.'] * n for _ in range(n)]
    for col in range(n):
        display_board[board[col]][col] = 'Q'
    for row in range(n):
        print(' '.join(display_board[row]))
    print(f"Cost: {cost}\n")

if __name__ == "__main__":
    n = int(input("Enter the number of queens (N): "))
    initial_state = list(map(int, input(f"Enter the initial state (row numbers for each column, space-
separated): ").split()))

    if len(initial_state) != n or any(r < 0 or r >= n for r in initial_state):
        print("Invalid initial state. Please ensure it has N elements with values from 0 to N-1.")
    else:
        solution, cost = hill_climb(initial_state)
        print(f"Final board configuration with cost {cost}:")
        print_board(solution, cost)
```

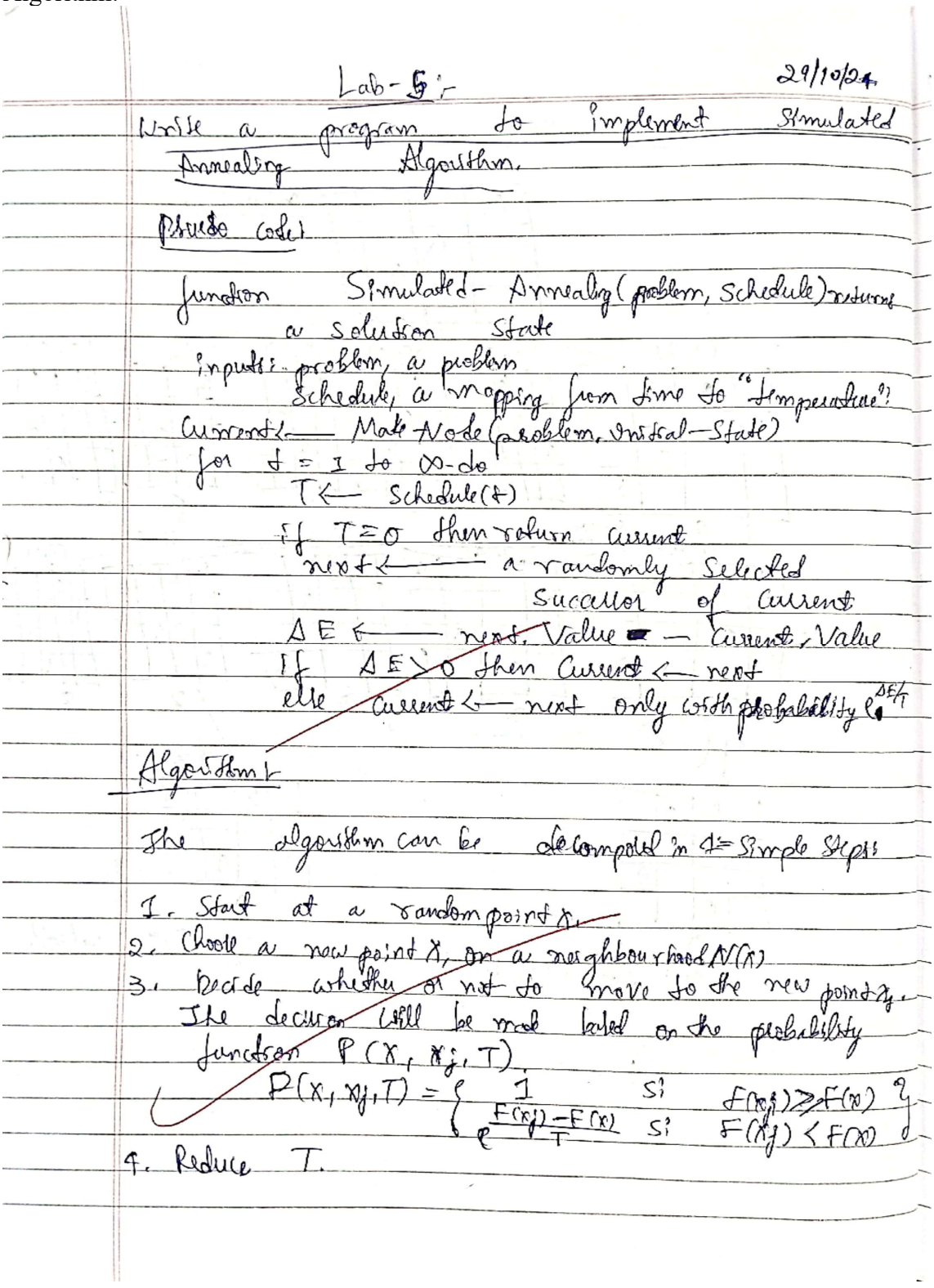
Output :

```
Enter the number of queens (N): 4
Enter the initial state (row numbers for each column, space-separated): 0 1 2 3
Initial board configuration:
Q . . .
. Q . .
. . Q .
. . . Q
Cost: 6

Iteration 1:
. . . .
Q Q . .
. . Q .
. . . Q
Cost: 4

Iteration 2:
. Q . .
Q . . .
. . Q .
. . . Q
Cost: 2

Final board configuration with cost 2:
. Q . .
Q . . .
. . Q .
. . . Q
Cost: 2
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



Lab - 5 :- 29/10/24

Write a program to implement Simulated Annealing Algorithm.

Pseudo code

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 $e^{\Delta E/T}$

Algorithm :

The algorithm can be decomposed in 4 simple steps:

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

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

4. Reduce T.

Bafna Gold
Date: Page:

1) Output: 8 Queen's problem

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

. . Q . . . . .
. . . . . Q . .
. . . . Q
. Q . . . . . .
. . . Q . . . .
Q . . . .
. . . . . Q . .
. . . . Q . . .

2) output: Tower of Hanoi problem

Best State (final configuration): [2 2 2]
Number of Correct disks on destination peg: 3.0
Tower of Hanoi Configuration:
peg 0: [ ]
peg 1: [ ]
peg 2: [0, 1, 2]

29/10/24

Code:

```
import mlrose_hiive as mlrose
import numpy as np

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

objective = mlrose.CustomFitness(queens_max)

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

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

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

print('The best position found is:', best_position)
print('The number of queens that are not attacking each other is:', best_objective)
def print_chessboard(solution):
    print("\nChessboard Configuration:")
    for row in range(8):
        line = ""
        for col in range(8):
            if solution[col] == row:
                line += " Q "
            else:
                line += " . "
        print(line)
print_chessboard(best_position)
```

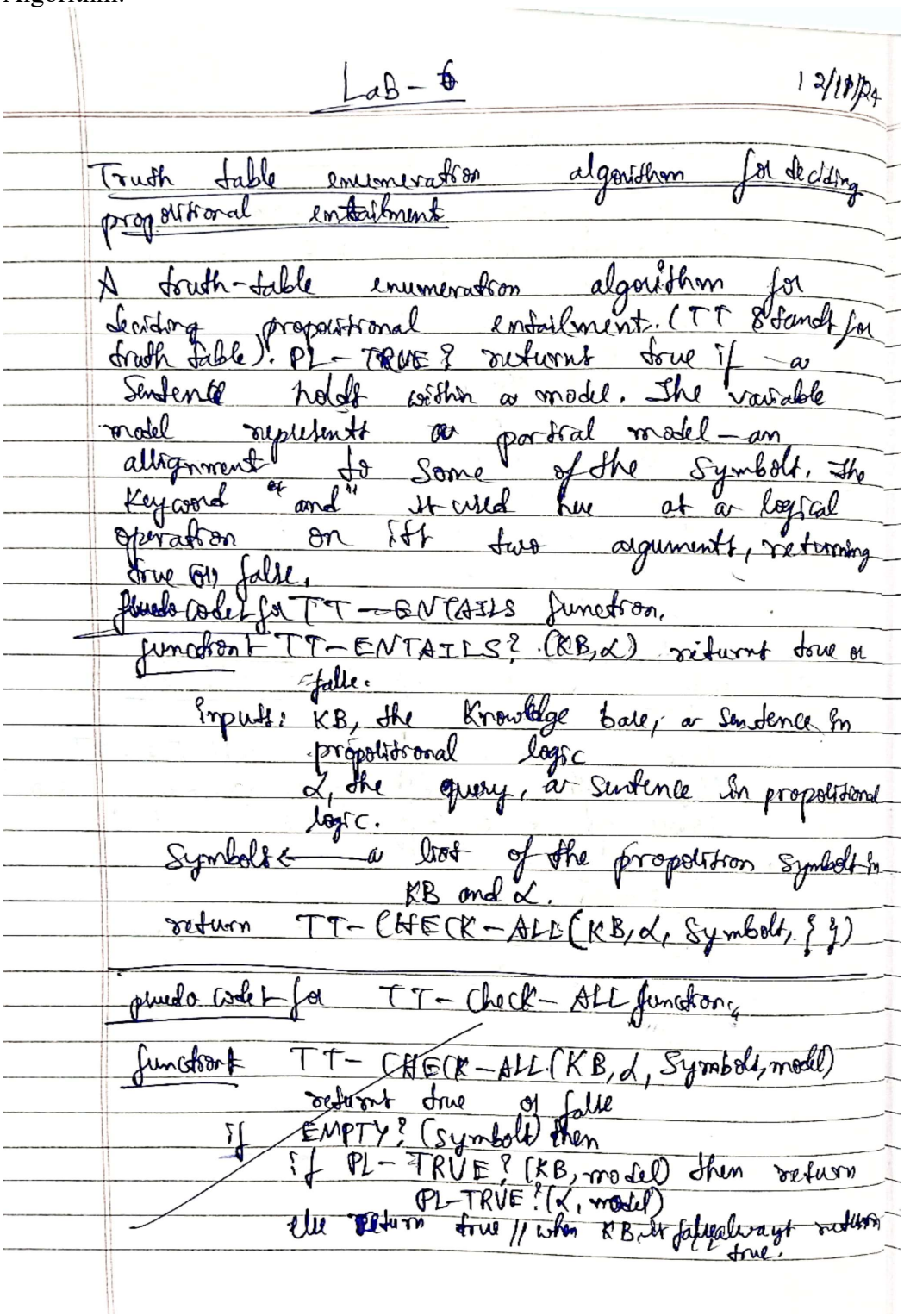
Output :

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

Chessboard Configuration:
 .  .  Q  .  .  .  .  .
 .  .  .  .  .  Q  .  .
 .  .  .  .  .  .  .  Q
 .  Q  .  .  .  .  .  .
 .  .  .  Q  .  .  .  .
 Q  .  .  .  .  .  .  .
 .  .  .  .  .  .  Q  .
 .  .  .  .  Q  .  .  .
```

Program 6

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

Algorithm:



Lab - 6 12/11/24

Truth table enumeration algorithm for deciding propositional entailment

A truth-table enumeration algorithm for deciding propositional entailment. (TT stands for truth table). PL-TRUE? returns true if a sentence holds within a model. The variable model represents a partial model - an assignment to some of the symbols. The keyword "and" is used here as a logical operation on its two arguments, returning true or false.

pseudo code for TT-ENTAILS function.

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

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

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

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

pseudo code for TT-Check-All function,

function TT-CHECK-ALL(KB, α, Symbols, model)
return true or false

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

Truth table :

Propositional Inference : Enumeration method :

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

Checking that $KB \models \alpha$

A	B	C	A∨C	B∨¬C	KB	α
false	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

KB entails α : True.

12/11/24

Code:

```
import itertools

def pl_true(sentence, model):
    A = model.get('A', False)
    B = model.get('B', False)
    C = model.get('C', False)

    if sentence == "A or B":
        return A or B
    elif sentence == "(A or C) and (B or not C)":
        return (A or C) and (B or not C)
    return False

def tt_entails(kb, alpha):
    symbols = ['A', 'B', 'C']
    return tt_check_all(kb, alpha, symbols, {})

def tt_check_all(kb, alpha, symbols, model):
    if not symbols:
        if pl_true(kb, model):
            return pl_true(alpha, model)
        else:
            return True
    else:
        p = symbols[0]
        rest = symbols[1:]

        model_true = model.copy()
        model_false = model.copy()
        model_true[p] = True
        model_false[p] = False

        return (tt_check_all(kb, alpha, rest, model_true) and
                tt_check_all(kb, alpha, rest, model_false))

kb = "(A or C) and (B or not C)"
alpha = "A or B"

result = tt_entails(kb, alpha)
print(f"KB entails α: {result}")

def generate_truth_table():
    print(f"{'A':<10}{'B':<10}{'C':<10}{'A∨C':<10}{'B∨¬C':<10}{'KB':<10}{'α (A∨B)':<10}")
    print("-" * 70)

    for A, B, C in itertools.product([False, True], repeat=3):
```

```
    A_or_C = A or C
    B_or_not_C = B or not C
    KB = (A or C) and (B or not C)
    alpha = A or B

    row =
f"{str(A):<10}{str(B):<10}{str(C):<10}{str(A_or_C):<10}{str(B_or_not_C):<10}{str(KB):<10}{str
(alpha):<10}"

    if KB and alpha:
        print(row)
        print("-" * len(row))
    else:
        print(row)

generate_truth_table()
```

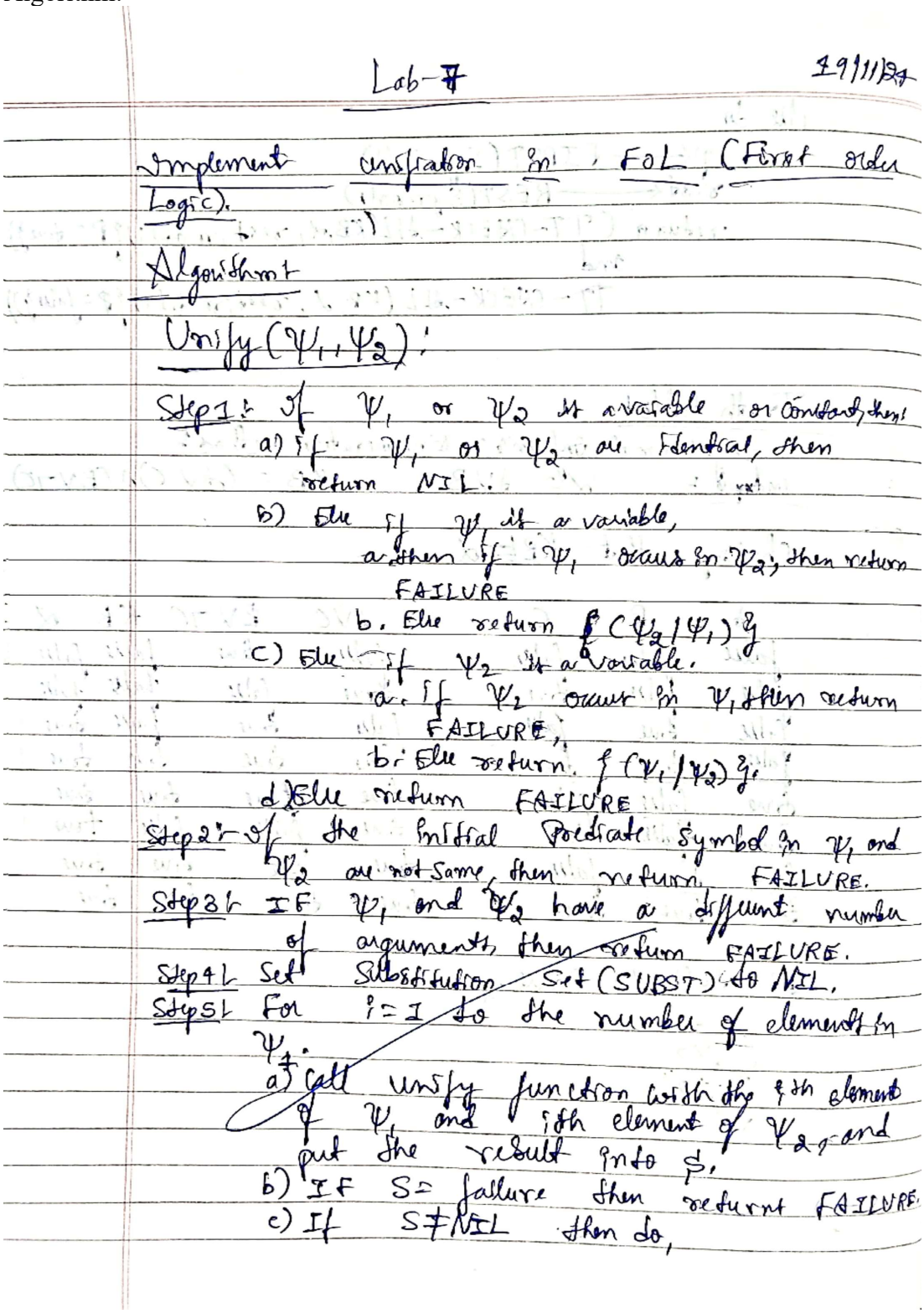
Output :

```
KB entails α: True
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
------------------------------------------------------------------------
```

Program 7

Implement unification in first order logic

Algorithm:



Lab-7 19/11/24

Implement unification in FOL (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_1 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 and ψ_2 are not same, then return FAILURE.
Step 3: IF ψ_1 and ψ_2 have a different number of arguments, then return FAILURE.
Step 4: Set Substitution Set (SUBST) to NIL.
Step 5: For i = 1 to the number of elements in ψ_1.
a) Call unify function with the ith element of ψ_1 and ith element of ψ_2, and put the result into S.
b) IF S = failure then return FAILURE
c) If S ≠ NIL then do,

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

b. SUBST = APPEND(S, SUBST).

Step 6: Return SUBST.

a) P(x, F(y)) → ①

P(a, F(g(u)) → ②

① and ② are identical if x is replaced with a in ①

P(a, F(y)) → ①

P(g, F(g(u)) → ②

If y is replaced with g(u) in ①

P(a, F(g(u))) → ①

P(a, F(g(u)) → ②

So finally both ① & ② are identical.

a) Q(a, g(a, a), f(y)) → ①

Q(a, g(f(b), a), x) → ②

① & ② are identical if x is replaced with f(b).

Q(a, g(f(b), a), f(y)) → ①

Q(a, g(f(b), a), x) → ②

If f(a) is replaced with x in ①

Q(a, g(f(b), a), x) → ①

Q(a, g(f(b), a), x) → ②

So finally both ① & ② are identical.

Ishika

Code:

```
def unify(Y1, Y2, subst=None):
    if subst is None:
        subst = {}

    # Step 1: Check if Y1 or Y2 is a variable or constant
    if Y1 == Y2:  # Identical constants or variables
        print(f"Unification Success: {Y1} and {Y2} are identical.")
        return subst
    elif is_variable(Y1):  # Y1 is a variable
        return unify_variable(Y1, Y2, subst)
    elif is_variable(Y2):  # Y2 is a variable
        return unify_variable(Y2, Y1, subst)

    # Step 2: Predicate symbols not the same
    if predicate_symbol(Y1) != predicate_symbol(Y2):
        print(f"Unification Failure: Predicate symbols {predicate_symbol(Y1)} and
{predicate_symbol(Y2)} don't match.")
        return None  # FAILURE

    # Step 3: Different number of arguments
    args1, args2 = arguments(Y1), arguments(Y2)
    if len(args1) != len(args2):
        print(f"Unification Failure: Different number of arguments in {Y1} and {Y2}.")
        return None  # FAILURE

    # Step 5: Recursively unify each element in the lists
    for a1, a2 in zip(args1, args2):
        subst = unify(a1, a2, subst)
        if subst is None:
            return None  # FAILURE

    # Step 6: Return SUBST (final substitution set)
    print(f"Unification Success: {Y1} and {Y2} unified with substitution {subst}.")
    return subst

def unify_variable(var, x, subst):
    if var in subst:
        print(f"Unification Success: Variable {var} is already in the substitution.")
        return unify(subst[var], x, subst)
    elif occurs_in(var, x):
        print(f"Unification Failure: Variable {var} occurs in {x} (circular reference).")
        return None  # FAILURE due to circular reference
    else:
        print(f"Unification Success: Substituting {var} with {x}.")
        subst[var] = x
        return subst
```

```
def predicate_symbol(expr):
    return expr[0] if isinstance(expr, list) else expr

def arguments(expr):
    return expr[1:] if isinstance(expr, list) else []

def is_variable(x):
    return isinstance(x, str) and x.islower()

def occurs_in(var, x):
    if var == x:
        return True
    elif isinstance(x, list):
        return any(occurs_in(var, xi) for xi in x)
    return False

# Example usage: Replace Y1 and Y2 with p(x, f(y)) and p(a, f(g(x)))
Y1 = ['p', 'x', ['f', 'y']]      # p(x, f(y))
Y2 = ['p', 'A', ['f', ['g', 'x']]]  # p(a, f(g(x)))
subst = unify(Y1, Y2)

if subst:
    print("Final Substitution:", subst,"Unification Successful")
else:
    print("Unification failed.")
```

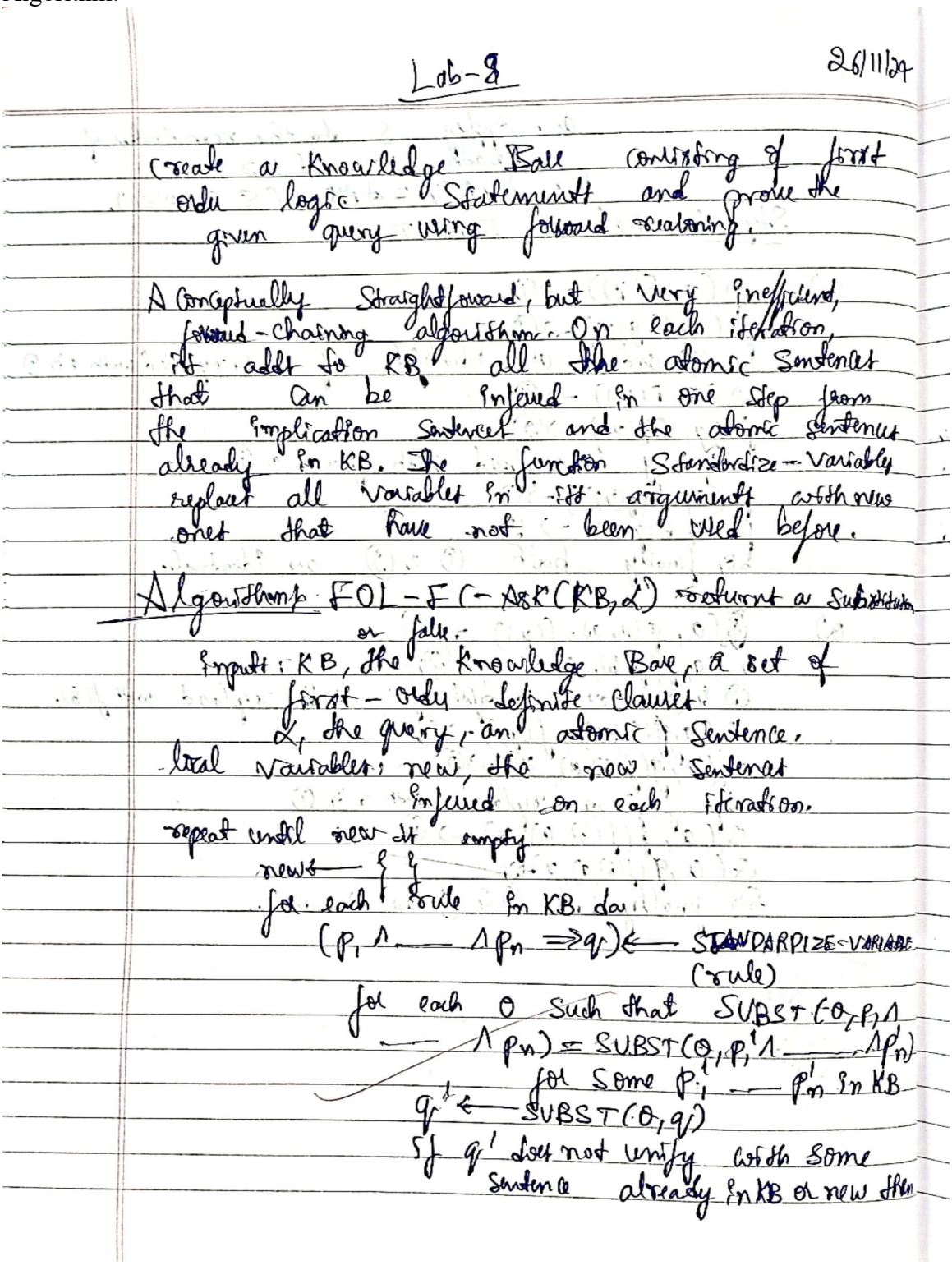
Output :

```
Unification Success: Substituting x with A.
Unification Success: Substituting y with ['g', 'x'].
Unification Success: ['f', 'y'] and ['f', ['g', 'x']] unified with substitution {'x': 'A', 'y': ['g', 'x']}.
Unification Success: ['p', 'x', ['f', 'y']] and ['p', 'A', ['f', ['g', 'x']]] unified with substitution {'x': 'A', 'y': ['g', 'x']}.
Final Substitution: {'x': 'A', 'y': ['g', 'x']} Unification Successful
```

Program 8

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

Algorithm:



Lab-8 26/11/24

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

A conceptually straightforward, but very inefficient, forward-chaining algorithm. On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB. The function STANDARDIZE-VARIABLES replaces all variables in its arguments with new ones that have not been used before.

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

Inputs: KB, the knowledge base, a set of first-order definite clauses
α, the query, an atomic sentence.
local variables: new, the new sentences inferred on each iteration.
repeat until new is empty
new ← { }
for each rule in KB do
(p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-VARIABLES (rule)
for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n) for some p'1, ..., p'n in KB
q' ← SUBST(θ, q)
if q' does not unify with some sentence already in KB or new then

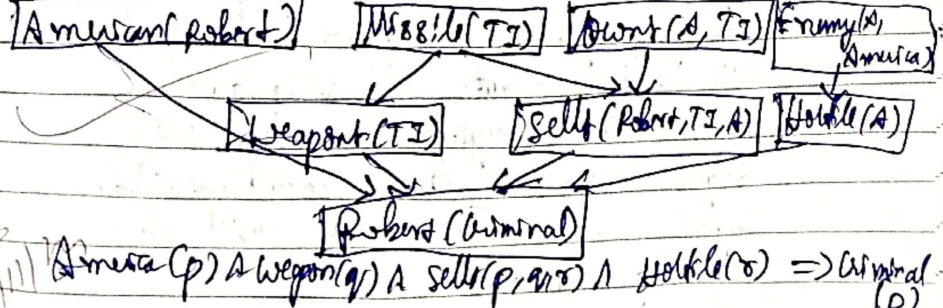
add q' to new
$\phi \leftarrow UNIFY(q', \alpha)$
if ϕ is not fail then return ϕ
add new to KB
return false

example

It is a crime for an American to sell weapons to hostile nations. Let's say p, q, and r are variables.
$American(p) \wedge Weapon(q) \wedge Sells(p, q, r) \wedge Hostile(r) \Rightarrow Criminal(p)$

Country A has some missiles
$\exists x\, Owns(A, x) \wedge Missile(x)$.
Existential instantiation: Introducing a new constant T1:
$Owns(A, T1)$
$Missile(T1)$
All of the missiles were sold to country A by Robert
$\forall x\, Missile(x) \wedge Owns(A, x) \Rightarrow Sells(Robert, x, A)$
Missiles are weapons.
$Missile(x) \Rightarrow Weapon(x)$
Enemy of America is known as hostile. The country A, an enemy of America
$\forall x\, Enemy(x, America) \Rightarrow Hostile(x)$
$Enemy(A, America)$.
Robert is an American
$American(Robert)$

To Prove: Robert is Criminal
$Criminal(Robert)$



$American(p) \wedge Weapon(q) \wedge Sells(p, q, r) \wedge Hostile(r) \Rightarrow Criminal(p)$.

Code:

```
facts = {
   "American(Robert)": True,
   "Missile(T1)": True,
   "Enemy(A, America)": True,
   "Owns(A, T1)": True,
   "Hostile(A)": False,
   "Weapon(T1)": False,
   "Sells(Robert, T1, A)": False,
   "Criminal(Robert)": False,
}

rules = [
   ("American(Robert) and Weapon(T1) and Sells(Robert, T1, A) and Hostile(A)",
"Criminal(Robert)"),
   ("Owns(A, T1) and Missile(T1)", "Weapon(T1)"),
   ("Missile(T1) and Owns(A, T1)", "Sells(Robert, T1, A)"),
   ("Enemy(A, America)", "Hostile(A)"),
]

def check_fact(fact):
   return facts.get(fact, False)

def parse_condition(condition):
   return condition.split(" and ")

def forward_reasoning():
   new_inferences = True
   while new_inferences:
      new_inferences = False
      for condition, conclusion in rules:
         condition_facts = parse_condition(condition)
         if all(check_fact(fact) for fact in condition_facts):
            if not check_fact(conclusion):
               facts[conclusion] = True
               new_inferences = True
               print(f"Inferred: {conclusion}")

def print_inferred_facts():
   forward_reasoning()
   print("\nFinal Inferred Facts:")
   for fact, value in facts.items():
      print(f"{fact} is {'TRUE' if value else 'FALSE'}")

print_inferred_facts()
```

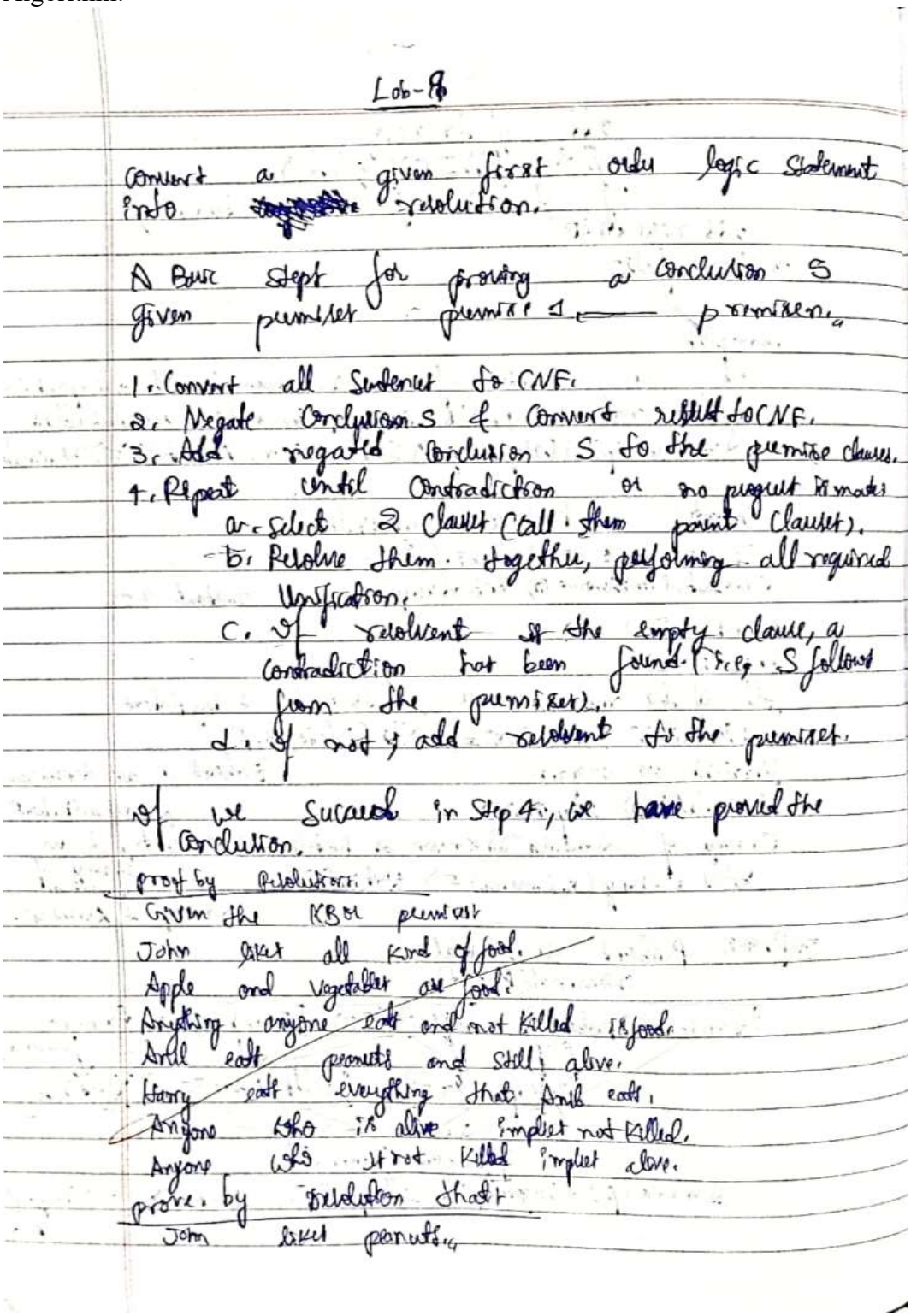
Output :

```
Inferred: Weapon(T1)
Inferred: Sells(Robert, T1, A)
Inferred: Hostile(A)
Inferred: Criminal(Robert)

Final Inferred Facts:
American(Robert) is TRUE
Missile(T1) is TRUE
Enemy(A, America) is TRUE
Owns(A, T1) is TRUE
Hostile(A) is TRUE
Weapon(T1) is TRUE
Sells(Robert, T1, A) is TRUE
Criminal(Robert) is TRUE
```

Program 9

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

Algorithm:



Lab-8

Convert a given first order logic statement into resolution.

A Basic step for proving a conclusion S given premises premise 1 ← premise n.

1. Convert all sentences 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 Unification.
   c. If resolvent is the empty clause, a contradiction has been found (i.e. S follows from the premises).
   d. If not, add resolvent to the premises.

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

Proof by Resolution:

Given the KB of premises
John likes all kind of food.
Apple and Vegetables are food.
Anything anyone eats and not killed is food.
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 that:
John likes peanuts.

Bafna Gold

Date: Page:

Representation in FOL :

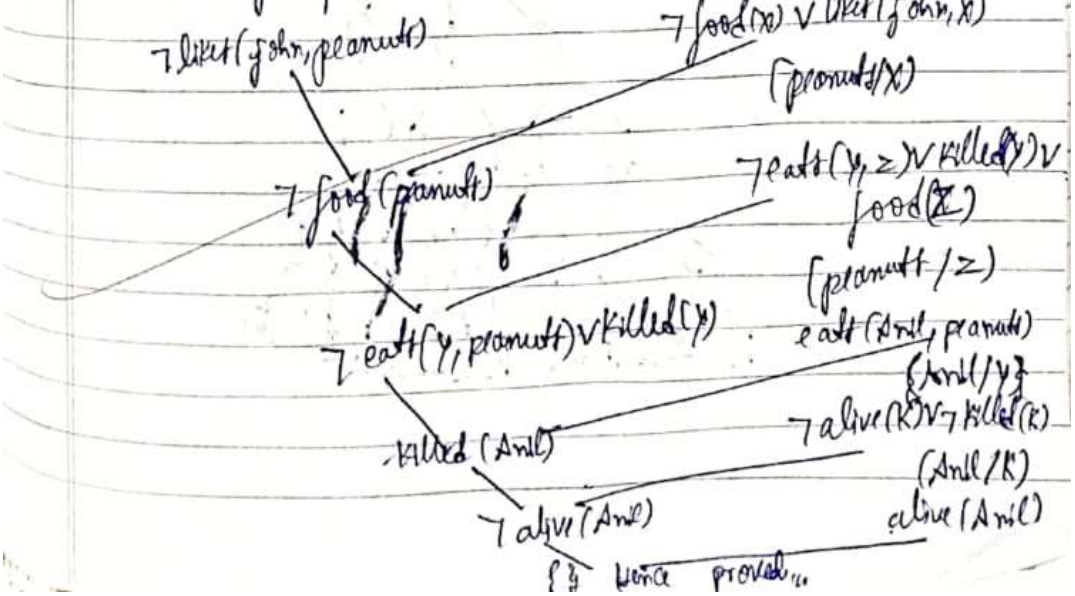
a. John likes all kind of food
b. Apple and vegetables are food.
c. Anything anyone eats and not killed is food.
d. Anil eats peanuts and still alive.
e. Harry eats everything that Anil eats.
f. Anyone who is alive implies not killed.
g. Anyone who is not killed implies alive.
h. John likes peanuts

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. $\forall x : alive(x) \rightarrow killed(x)$
h. $likes(John, peanuts)$

Prove by resolution that

Proof by Resolution

a. $\neg food(x) \vee likes(John, x)$
b. $food(apple)$
c. $food(vegetables)$
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. $likes(John, peanuts)$



Code:

```
from sympy import symbols, And, Or, Not, Implies, to_cnf

# Define constants (entities in the problem)
John, Anil, Harry, Apple, Vegetables, Peanuts, x, y = symbols('John Anil Harry Apple Vegetables
Peanuts x y')

# Define predicates as symbols (this works as a workaround)
Food = symbols('Food')
Eats = symbols('Eats')
Likes = symbols('Likes')
Alive = symbols('Alive')
Killed = symbols('Killed')

# Knowledge Base (Premises) in First-Order Logic
premises = [
    # 1. John likes all kinds of food: Food(x) → Likes(John, x)
    Implies(Food, Likes),

    # 2. Apples and vegetables are food: Food(Apple) ∧ Food(Vegetables)
    And(Food, Food),

    # 3. Anything anyone eats and is not killed is food: (Eats(y, x) ∧ ¬Killed(y)) → Food(x)
    Implies(And(Eats, Not(Killed)), Food),

    # 4. Anil eats peanuts and is still alive: Eats(Anil, Peanuts) ∧ Alive(Anil)
    And(Eats, Alive),

    # 5. Harry eats everything that Anil eats: Eats(Anil, x) → Eats(Harry, x)
    Implies(Eats, Eats),

    # 6. Anyone who is alive implies not killed: Alive(x) → ¬Killed(x)
    Implies(Alive, Not(Killed)),

    # 7. Anyone who is not killed implies alive: ¬Killed(x) → Alive(x)
    Implies(Not(Killed), Alive),
]

# Negated conclusion to prove: ¬Likes(John, Peanuts)
negated_conclusion = Not(Likes)

# Convert all premises and the negated conclusion to Conjunctive Normal Form (CNF)
cnf_clauses = [to_cnf(premise, simplify=True) for premise in premises]
cnf_clauses.append(to_cnf(negated_conclusion, simplify=True))

# Function to resolve two clauses
def resolve(clause1, clause2):
```

```
    """
    Resolve two CNF clauses to produce resolvents.
    """
    clause1_literals = clause1.args if isinstance(clause1, Or) else [clause1]
    clause2_literals = clause2.args if isinstance(clause2, Or) else [clause2]
    resolvents = []

    for literal in clause1_literals:
        if Not(literal) in clause2_literals:
            # Remove the literal and its negation and combine the rest
            new_clause = Or(
                *[l for l in clause1_literals if l != literal],
                *[l for l in clause2_literals if l != Not(literal)]
            ).simplify()
            resolvents.append(new_clause)

    return resolvents

# Function to perform resolution on the set of CNF clauses
def resolution(cnf_clauses):
    """
    Perform resolution on CNF clauses to check for a contradiction.
    """
    clauses = set(cnf_clauses)
    new_clauses = set()

    while True:
        clause_list = list(clauses)
        for i in range(len(clause_list)):
            for j in range(i + 1, len(clause_list)):
                resolvents = resolve(clause_list[i], clause_list[j])
                if False in resolvents:  # Empty clause found
                    return True  # Contradiction found; proof succeeded
                new_clauses.update(resolvents)

        if new_clauses.issubset(clauses):  # No new information
            return False  # No contradiction; proof failed

        clauses.update(new_clauses)

# Perform resolution to check if the conclusion follows
result = resolution(cnf_clauses)
print("Does John like peanuts? ", "Yes, proven by resolution." if result else "No, cannot be proven.")
```

Output :

```
Does John like peanuts?  Yes, proven by resolution.
```

Program 10

Implement Alpha-Beta Pruning.

Algorithm:

Lab -10

Implement Alpha-Beta pruning Algorithm

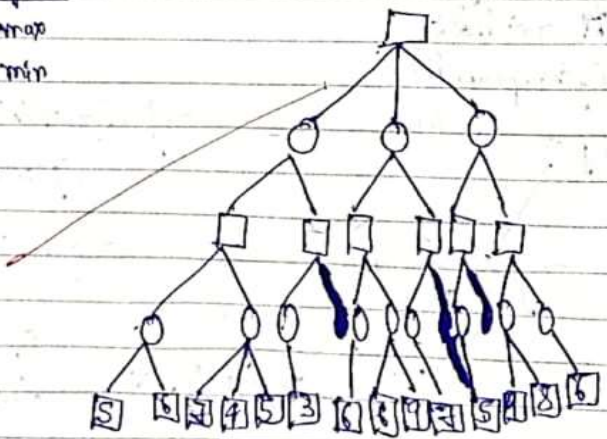
Algorithm:

- Alpha(α) – Beta(β) property to compute find the optimal path without looking at every node in the game tree.
- Max contains Alpha(α) and Min contains Beta(β) bound during the Calculation.
- In both MIN and MAX node, we return when α > β which Compare with its parent node only.
- Both minimax and Alpha(α) – Beta(β) cut off give same path.
- Alpha(α) -Beta(β) give optimal solution at its fastest left time to get the value for the root node.

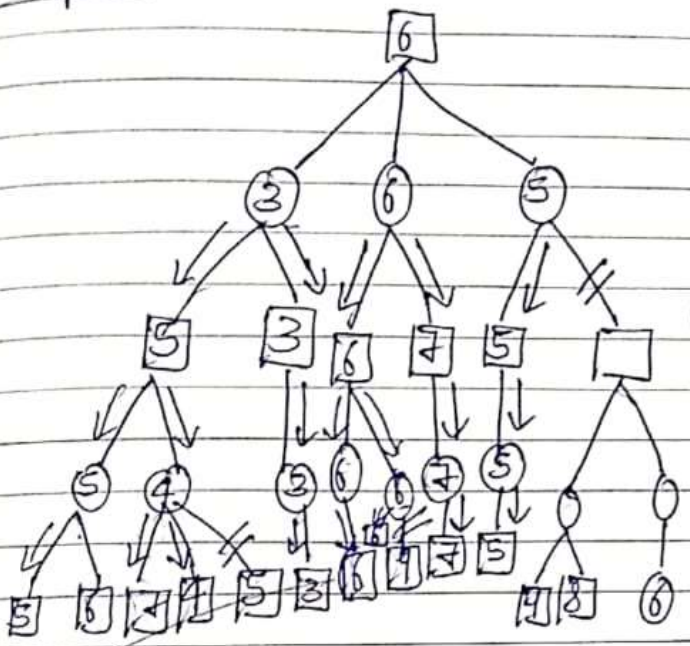
Input:-

□ max

○ min



Output



3/12/24

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
```