VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Samraat Dabolay (1BM22CS236)

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 **Samraat Dabolay (1BM22CS236),** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

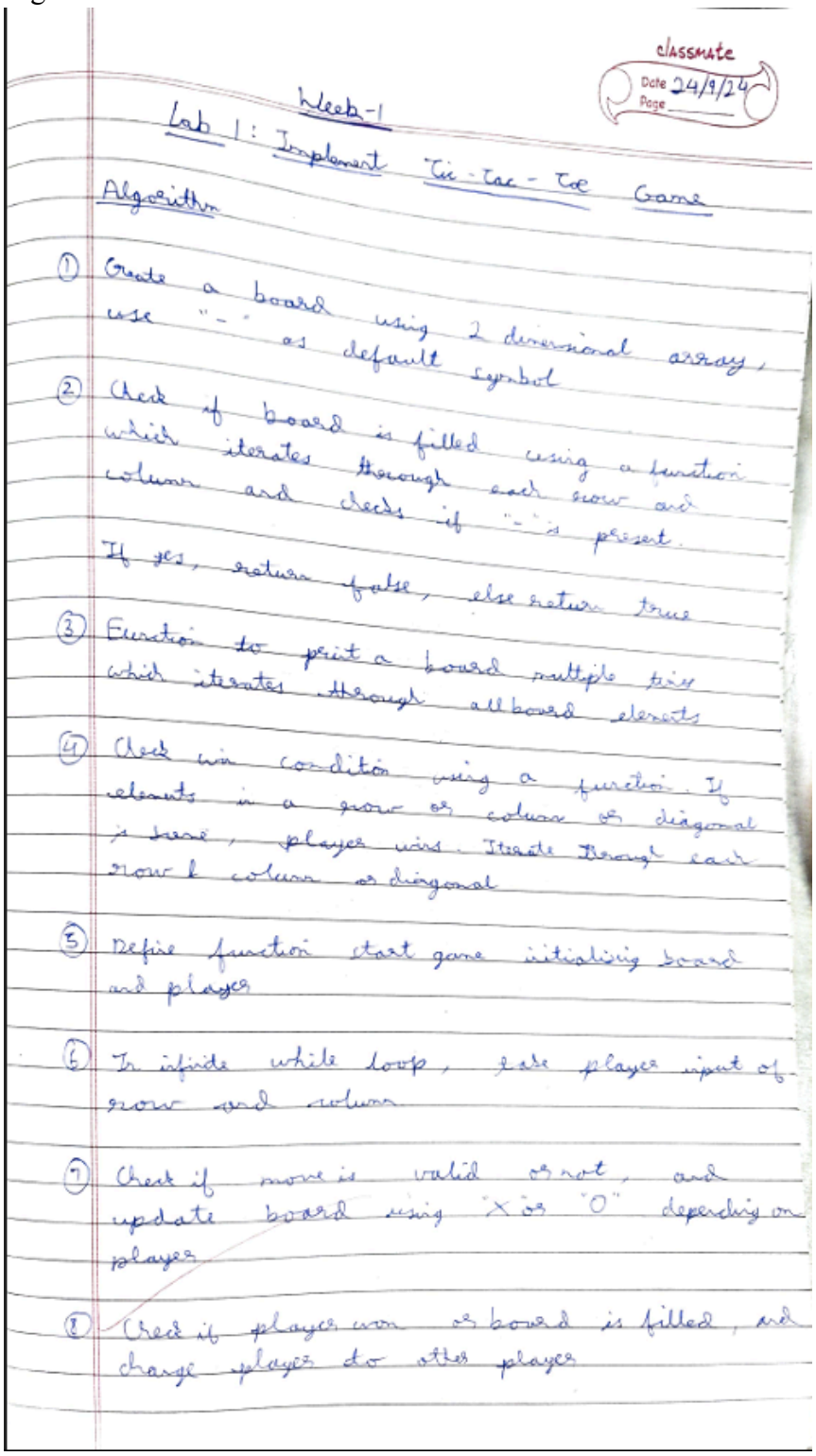
Dr. Seema Patil Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Github Link: https://github.com/samraatd/AI-Lab

Program 1
Implement Tic –Tac –Toe Game

Algorithm:



Week-1
Lab 1: Implement Tic-Tac-Toe Game
Algorithm
1. Create a board using 2 dimensional arrays, use "-" as default symbol
2. Check if board is filled using a function which iterates through each row and column and checks if "-" is present. If yes, return false, else return true
3. Function to print a board multiple times which iterates through all board elements
4. Check win condition using a functio. If elements in a row or column or diagonal is same, player wins. Iterate through each row & column & diagonal
5. Define function start game initialising board and player
6. In infinite while loop, take player input of row and column
7. Check if move is valid or not, and update board using "X" or "O" depending on player
8. Check if player won or board is filled, and change player to other player

Code:

```
def filled(board):
    for row in board:
        if "-" in row:
            return False
    return True

def print_board(board):
    for row in board:
        print(" | ".join(row))
def check_win(board):
  for row in board:
    if row[0] == row[1] == row[2] != "-":
      return True
  for col in range(3):
    if board[0][col] == board[1][col] == board[2][col] != "-":
      return True
  for diag in range(2):
    if board[0][0] == board[1][1] == board[2][2] != "-" or board[0][2] == board[1][1] == board[2][0]
!= "-":
      return True
  return False
def start_game():
  board = [["-"] * 3 for _ in range(3)]
  player = 1
  while True:
    print_board(board)
    print("Player " + str(player) + "'s turn.")
    print("Enter your move (row, col):")
    move = input()
    row, col = map(int, move.split(","))
    if row>2 or col>2 or board[row][col] != "-" :
      print("Invalid move. Try again.")
      continue
    board[row][col] = "X" if player == 1 else "O"
    if check_win(board):
      print_board(board)
      print(f"{player} wins!")
      break
    if filled(board):
      print_board(board)
      print("It's a tie!")
      break
    player = 2 if player == 1 else 1
start_game()
```

Output:

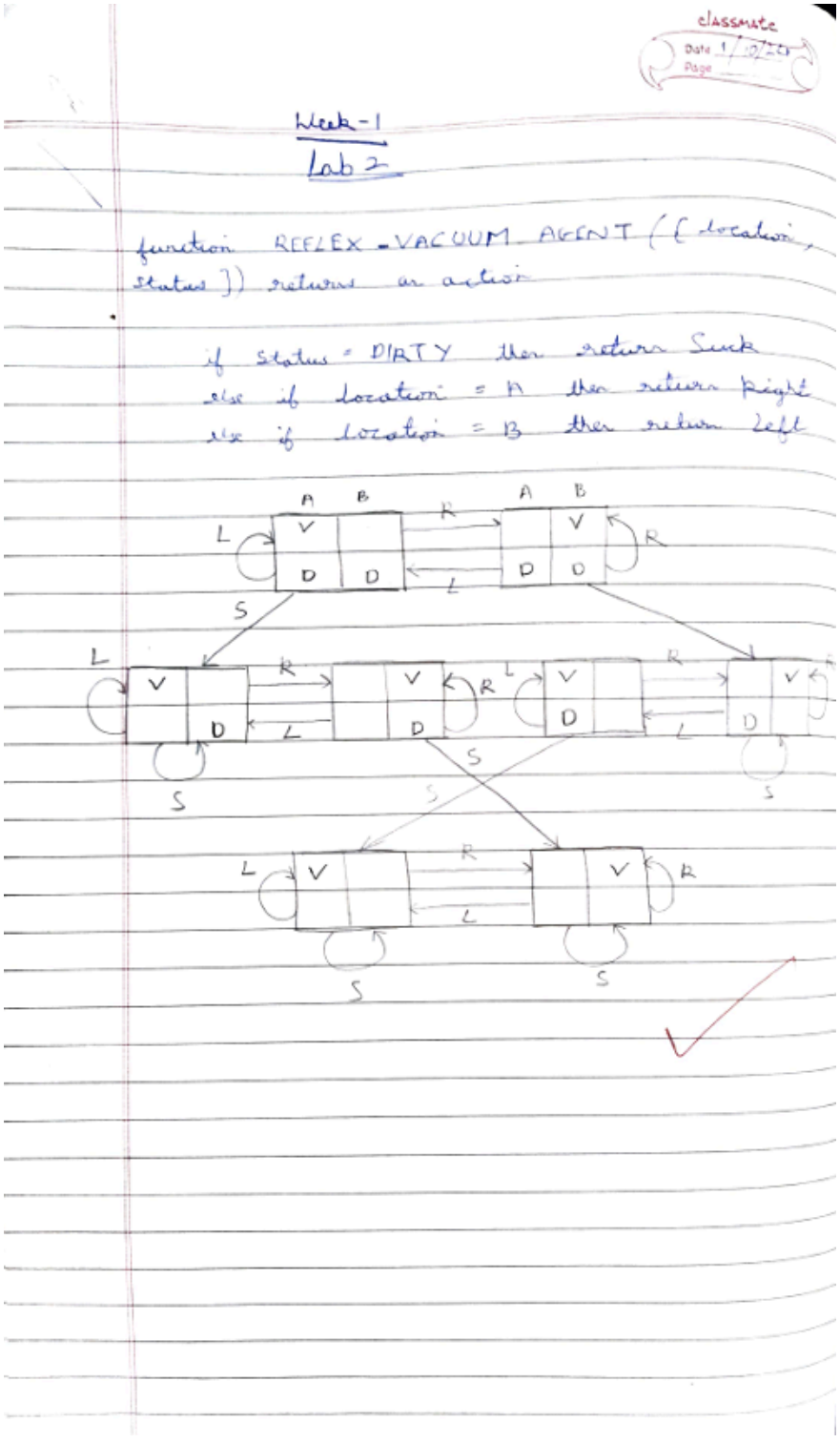
```
- | - | -
- | - | -
- | - | -
Player 1's turn.
Enter your move (row, col): 3,3
Invalid move. Try again.
- | - | -
- | - | -
- | - | -
Player 1's turn.
Enter your move (row, col): 0,1
- | X | -
- | - | -
- | - | -
Player 2's turn.
Enter your move (row, col): 0,1
Invalid move. Try again.
- | X | -
- | - | -
- | - | -
Player 2's turn.
Enter your move (row, col): 0,2
- | X | O
- | - | -
- | - | -
Player 1's turn.
Enter your move (row, col): 1,1
- | X | O
- | X | -
- | - | -
Player 2's turn.
Enter your move (row, col): 0,2
Invalid move. Try again.
- | X | O
- | X | -
- | - | -
Player 2's turn.
Enter your move (row, col): 2,2
- | X | O
- | X | -
- | - | O
Player 1's turn.
Enter your move (row, col): 2,1
- | X | O
- | X | -
- | X | O
1 wins!
```

```
- | - | -
- | - | -
- | - | -
Player 1's turn.
Enter your move (row, col): 0,0
X | - | -
- | - | -
- | - | -
Player 2's turn.
Enter your move (row, col): 0,1
X | O | -
- | - | -
- | - | -
Player 1's turn.
Enter your move (row, col): 0,2
X | O | X
- | - | -
- | - | -
Player 2's turn.
Enter your move (row, col): 1,1
X | O | X
- | O | -
- | - | -
Player 1's turn.
Enter your move (row, col): 1,0
X | O | X
X | O | -
- | - | -
Player 2's turn.
Enter your move (row, col): 1,2
X | O | X
X | O | O
- | - | -
Player 1's turn.
Enter your move (row, col): 2,2
X | O | X
X | O | O
- | - | X
Player 2's turn.
Enter your move (row, col): 2,0
X | O | X
X | O | O
O | - | X
Player 1's turn.
Enter your move (row, col): 2,1
It's a tie!
```

```
- | - | -
- | - | -
- | - | -
Player 1's turn.
Enter your move (row, col): 3,3
Invalid move. Try again.
- | - | -
- | - | -
- | - | -
Player 1's turn.
Enter your move (row, col): 0,1
- | X | -
- | - | -
- | - | -
Player 2's turn.
Enter your move (row, col): 0,1
Invalid move. Try again.
- | X | -
- | - | -
- | - | -
```

Implement vacuum cleaner agent

Algorithm:



Algorithm

1. Define function and set goal state {'A': '0', 'B': '0'} and cost = 0
2. Input location of vacuum initially, and status of both rooms A and B
3. If location is A, check if status of A is dirty. Increment cost by 1 to clear A
4. Check status of B and increment cost to move to B, then again increment cost to clear B, only if B is dirty
5. If location is B, check status of B and increment cost if dirty
6. Check status of A and increment cost to move to A and clear A, only if A is dirty
7. Each time room is cleared, update goal state from 1 to 0
8. Print cost when goal state is reached

11/10/2025

Code:

```
def vacuum_world():

    goal_state = {'A': '0', 'B': '0'}
    cost = 0

    location_input = input("Enter Location of Vacuum")
    status_input = input("Enter status of " + location_input)
    status_input_complement = input("Enter status of other room")


    if location_input == 'A':
        print("Initial Location Condition" + str({"A":status_input, "B":status_input_complement}))
        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")

            goal_state['A'] = '0'
            cost += 1
            print("Cost for CLEANING A " + str(cost))
            print("Location A has been Cleaned.")

            if status_input_complement == '1':

                print("Location B is Dirty.")
                print("Moving right to the Location B. ")
                cost += 1
                print("COST for moving RIGHT " + str(cost))

                goal_state['B'] = '0'
                cost += 1
                print("COST for SUCK " + str(cost))
                print("Location B has been Cleaned. ")
            else:
                print("No action" + str(cost))

                print("Location B is already clean.")

        if status_input == '0':
            print("Location A is already clean ")
            if status_input_complement == '1':
                print("Location B is Dirty.")
                print("Moving RIGHT to the Location B. ")
                cost += 1
                print("COST for moving RIGHT " + str(cost))

                goal_state['B'] = '0'
                cost += 1
```

```
            print("Cost for SUCK " + str(cost))
            print("Location B has been Cleaned. ")
        else:
            print("No action " + str(cost))
            print(cost)

            print("Location B is already clean.")

else:
    print("Initial Location Condition " + str({"A":status_input_complement, "B":status_input}))
    print("Vacuum is placed in location B")

    if status_input == '1':
        print("Location B is Dirty.")

        goal_state['B'] = '0'
        cost += 1
        print("COST for CLEANING " + str(cost))
        print("Location B has been Cleaned.")

        if status_input_complement == '1':

            print("Location A is Dirty.")
            print("Moving LEFT to the Location A. ")
            cost += 1
            print("COST for moving LEFT" + str(cost))

            goal_state['A'] = '0'
            cost += 1
            print("COST for SUCK " + str(cost))
            print("Location A has been Cleaned.")

    else:
        print(cost)

        print("Location B is already clean.")

        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("Moving LEFT to the Location A. ")
            cost += 1
            print("COST for moving LEFT " + str(cost))

            goal_state['A'] = '0'
            cost += 1
            print("Cost for SUCK " + str(cost))
            print("Location A has been Cleaned. ")
```

```
        else:
            print("No action " + str(cost))

            print("Location A is already clean.")

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


vacuum_world()
```

Output:

```
Enter Location of VacuumA
Enter status of A1
Enter status of other room1
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A 1
Location A has been Cleaned.
Location B is Dirty.
Moving right to the Location B.
COST for moving RIGHT2
COST for SUCK 3
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
```

```
Enter Location of VacuumA
Enter status of A0
Enter status of other room0
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is already clean
No action 0
0
Location B is already clean.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 0
```

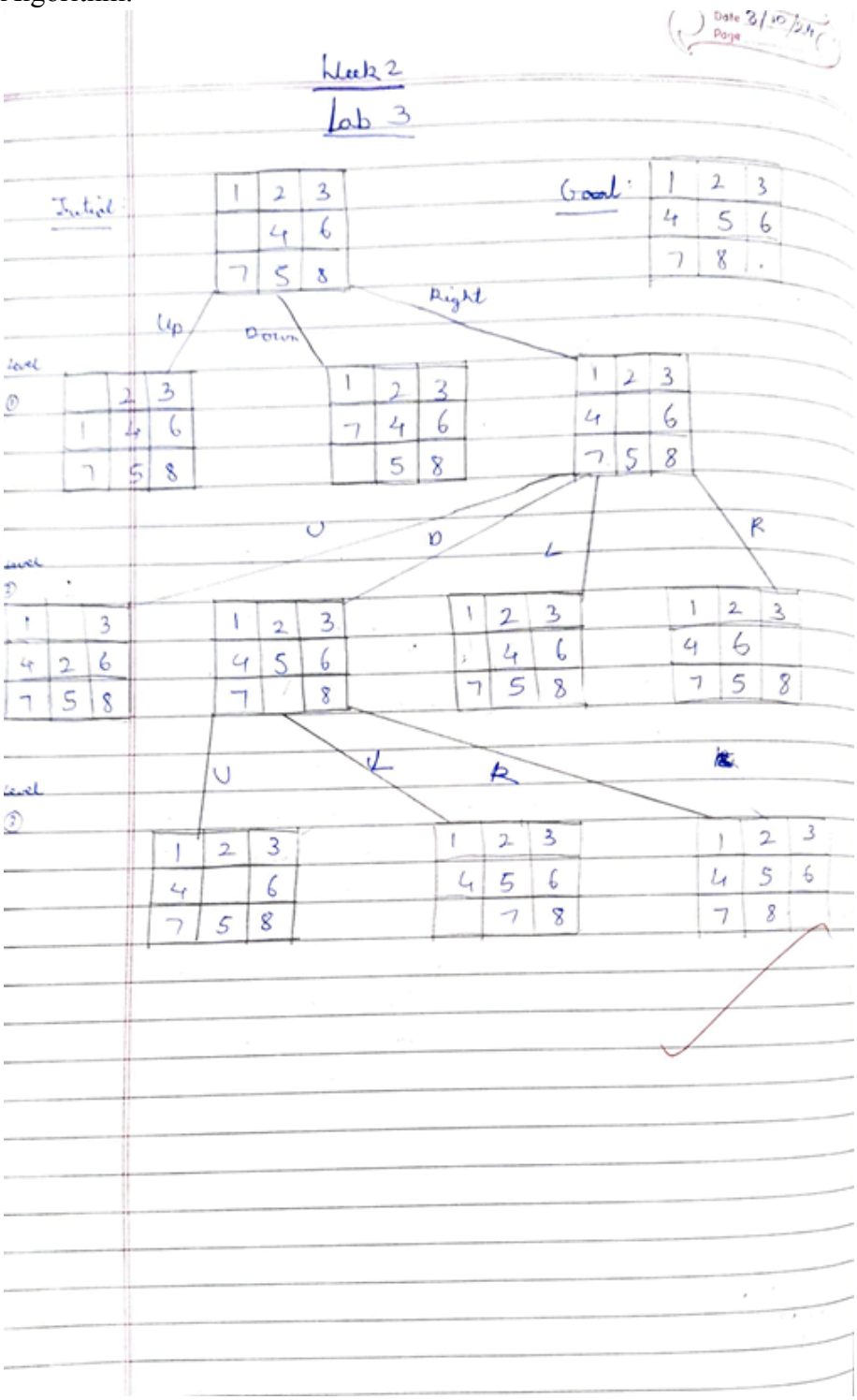
```
Enter Location of VacuumA
Enter status of A0
Enter status of other room1
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is already clean
Location B is Dirty.
Moving RIGHT to the Location B.
COST for moving RIGHT 1
Cost for SUCK2
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 2
```

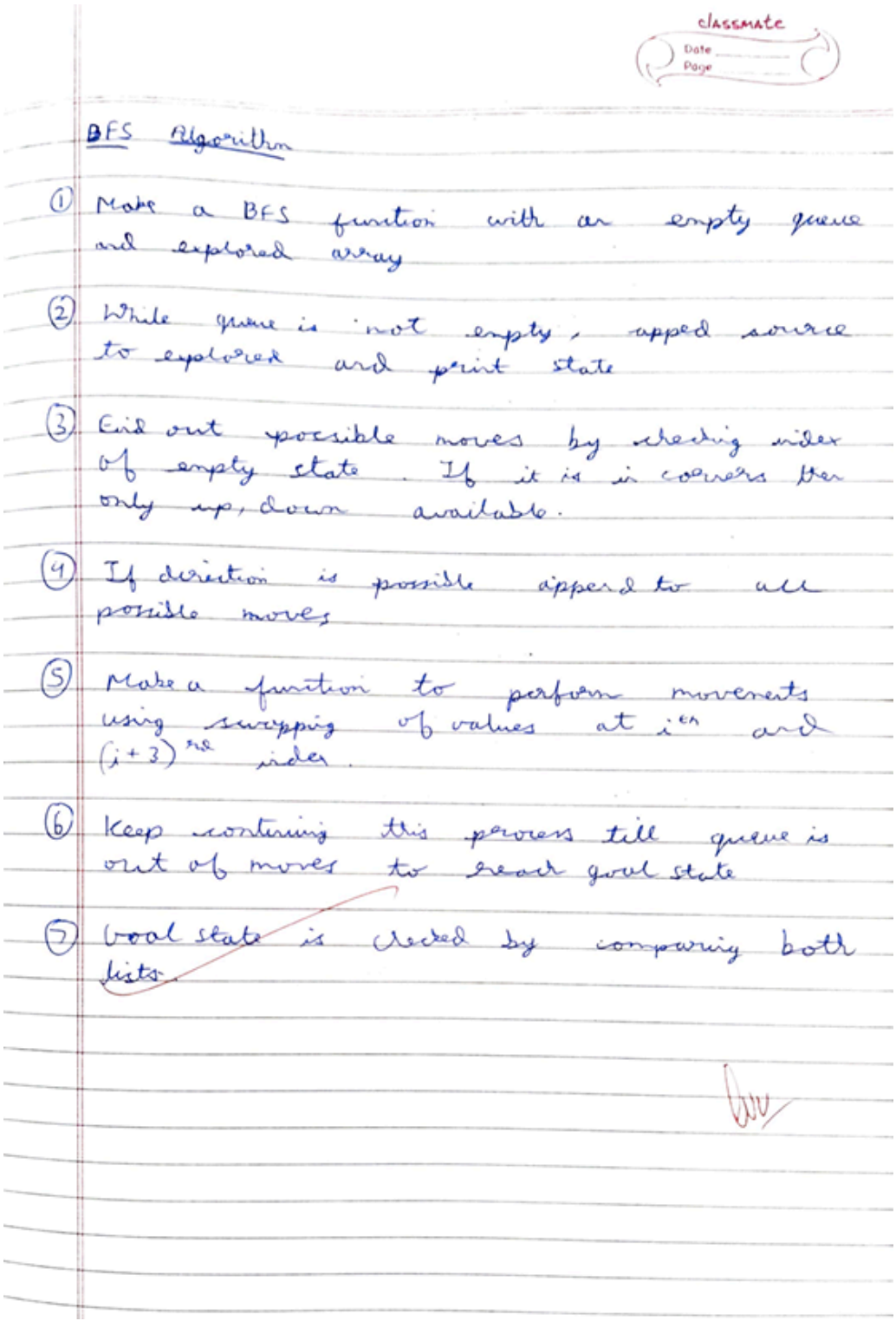
```
Enter Location of VacuumA
Enter status of A1
Enter status of other room0
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A 1
Location A has been Cleaned.
No action1
Location B is already clean.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 1
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)

Algorithm:





Code:

```
cnt = 0;
def print_state(in_array):
```

```
    global cnt
    cnt += 1
    for row in in_array:
        print(' '.join(str(num) for num in row))
    print()  # Print a blank line for better readability


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


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


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


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


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


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


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

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


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


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

Output:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

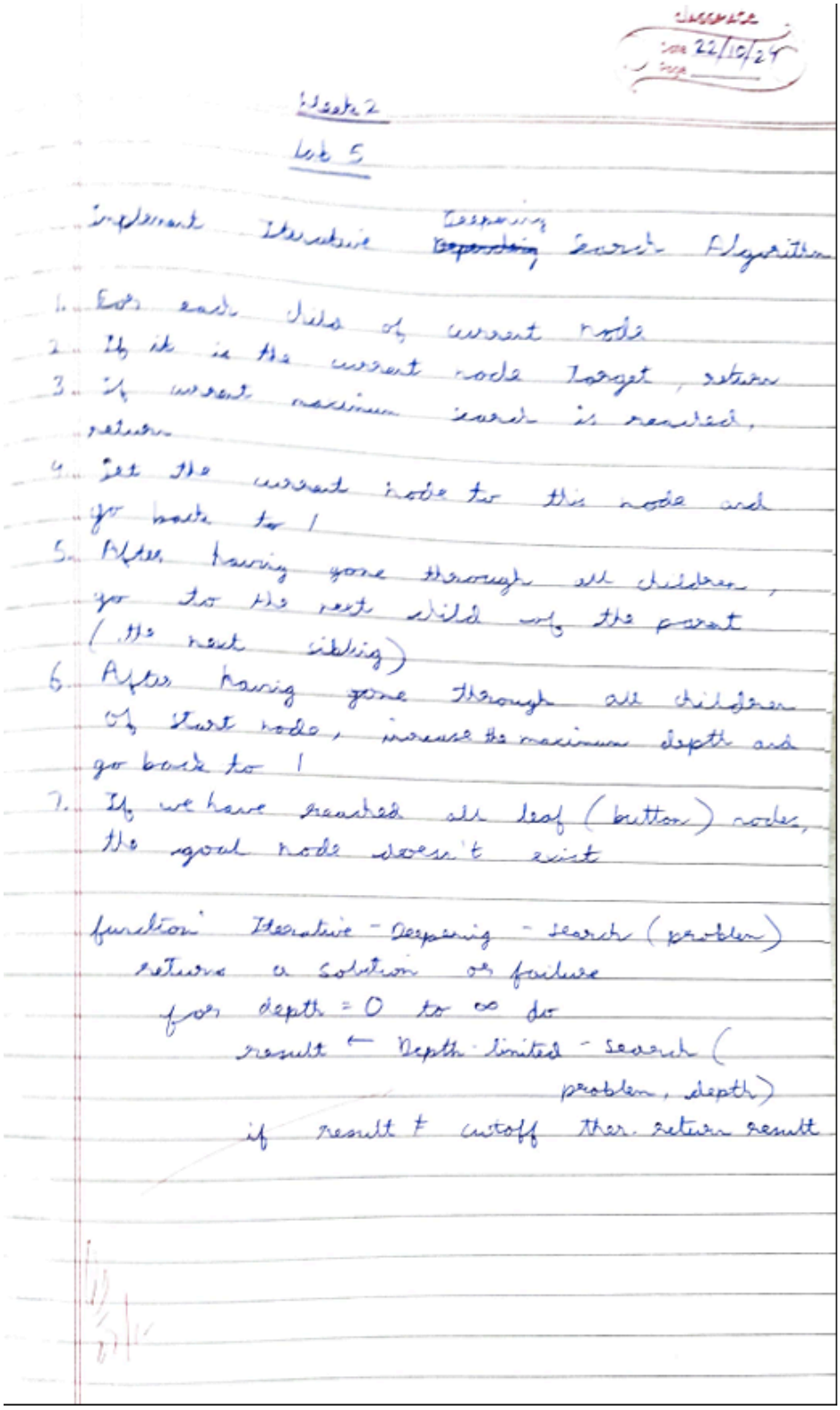
```
Took a L move
Current state:
2 4 3
1 6 8
0 7 5

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1 2 3
4 5 6
7 8 0

Number of states : 42
Solution found: True
```

Implement Iterative deepening search algorithm

Algorithm:



Date 22/10/24

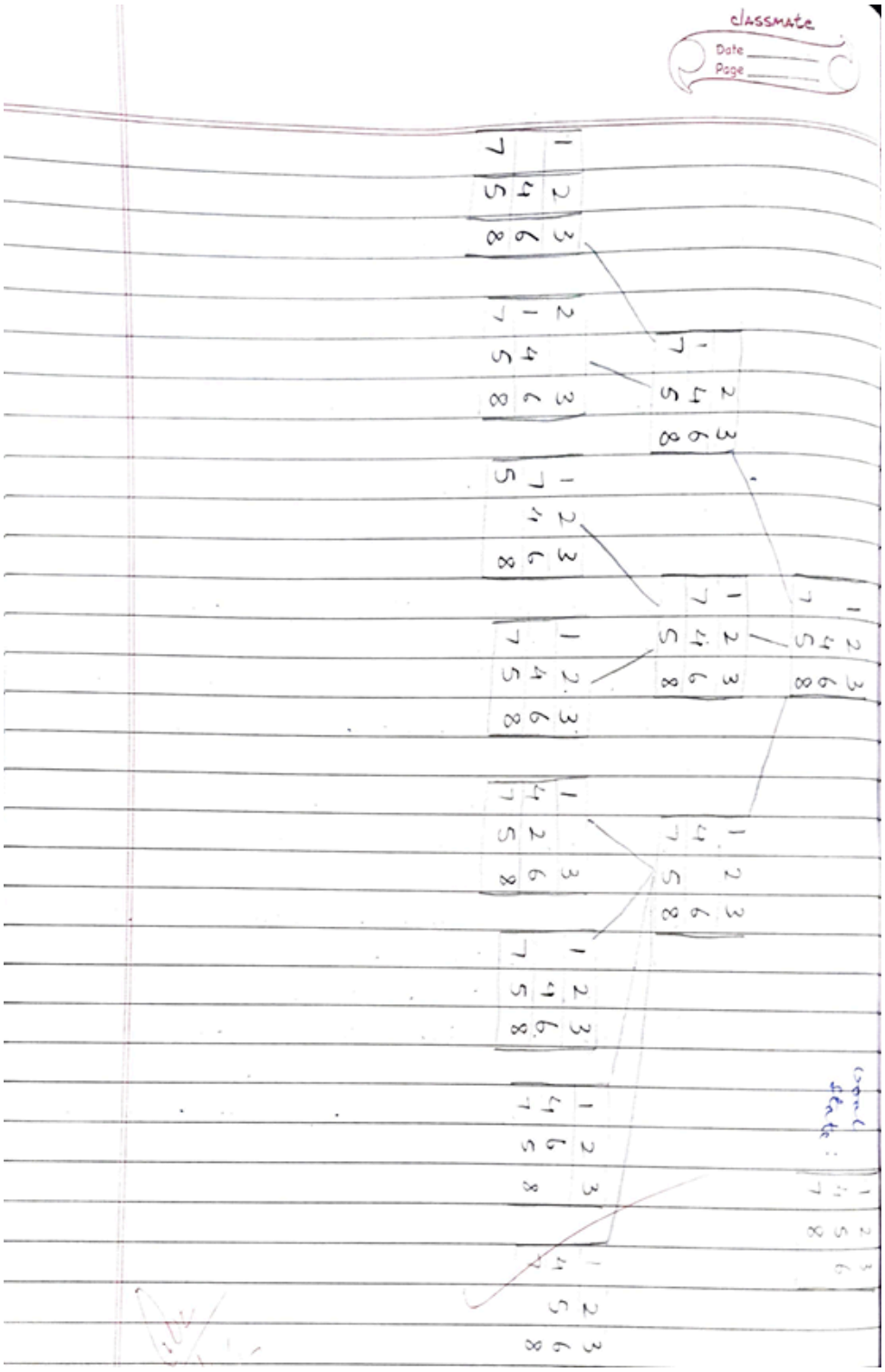
Week 2

Lab 5

Implement Iterative Deepening Search Algorithm

1. Explore each child of current node
2. If it is the current node target, return
3. If current maximum level 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 that node, increase the maximum depth and go back to 1
7. If we have reached all leaf (button) nodes, the goal node doesn't exist

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



Code:

```
class PuzzleState:
    def __init__(self, board, empty_tile_pos, moves=0, previous=None):
        self.board = board
        self.empty_tile_pos = empty_tile_pos
        self.moves = moves
        self.previous = previous


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


    def get_possible_moves(self):
        possible_moves = []
        x, y = self.empty_tile_pos
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]  # Down, Up, Right, Left


        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 = [list(row) for row in self.board]  # Create a copy
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
                possible_moves.append(PuzzleState(new_board, (new_x, new_y), self.moves + 1, self))
        return possible_moves


def iddfs(initial_state, goal_state, depth_limit):
    def dls(state, depth):
        if state.is_goal(goal_state):
            return state
        if depth == 0:
            return None


        for move in state.get_possible_moves():
            result = dls(move, depth - 1)
            if result is not None:
                return result
        return None


    for depth in range(depth_limit):
        result = dls(initial_state, depth)
```

```
        if result is not None:
            return result
    return None


# Example initial and goal states
initial_board = [
    [1, 2, 3],
    [4, 0, 5],
    [7, 8, 6]
]
goal_board = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]


# Initial state and goal state setup
initial_state = PuzzleState(initial_board, (1, 1))  # (1, 1) is the position of the empty tile
goal_state = goal_board


# Set a depth limit
depth_limit = 1


# Run IDDFS
solution = iddfs(initial_state, goal_state, depth_limit)


# Function to print the solution path
def print_solution(solution):
    path = []
    while solution:
        path.append(solution.board)
        solution = solution.previous
    for step in reversed(path):
        for row in step:
            print(row)
        print()


if solution:
    print("Solution found:")
    print_solution(solution)
else:
```

```
    print("No solution found within the depth limit.")
```

Output:

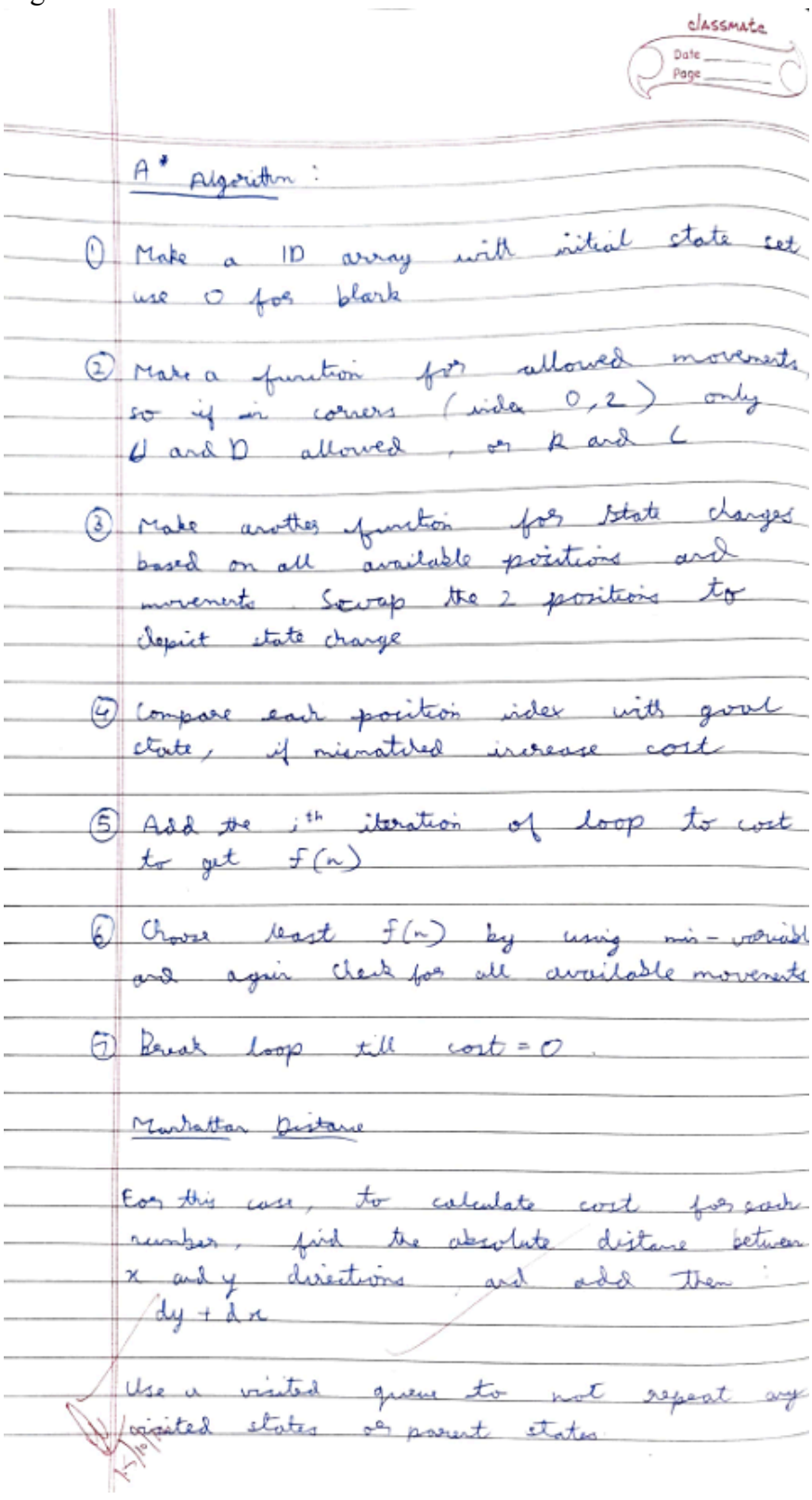
```
Depth:  1
No solution found within the depth limit.
```

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

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

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

Program 3

Implement A* search algorithm with Manhattan Distance

Algorithm:



A* Algorithm :

1. Make a 1D array with initial state set use 0 for blank
2. Make a function for allowed movements so if in corners (idx 0, 2) only U and D allowed, or R and L
3. Make another function for state changes based on all available positions and movements. Swap the 2 positions to depict state change
4. Compare each position index with goal state, if mismatched increase cost
5. Add the i^{th} iteration of loop to cost to get $f(n)$
6. Choose least $f(n)$ by using min - variable and again check for all available movements
7. Break loop till cost = 0

Manhattan Distance

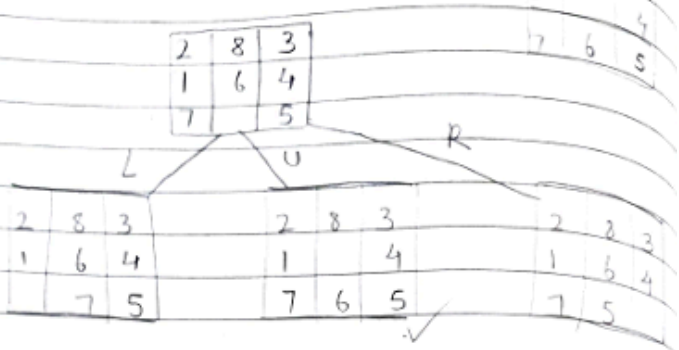
For this case, to calculate cost for each number, find the absolute distance between x and y directions and add them : $dy + dx$

Use a visited queue to not repeat any visited states or parent states

b) Manhattan Distance

goal:

1	2	3
8		4
7	6	5



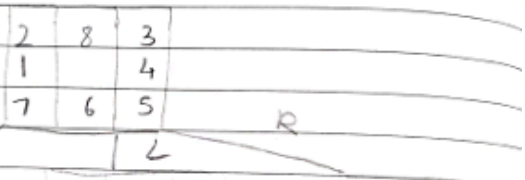
g = 1

h(n) = 1 2 3 4 5 6 7 8

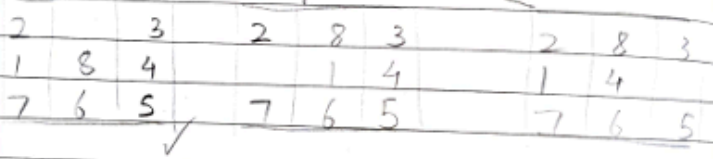
for U 1 1 0 0 0 0 0 2 = 4

for L 1 1 0 0 0 1 1 2 = 6

for R 1 1 0 0 1 1 0 2 = 6



g = 2



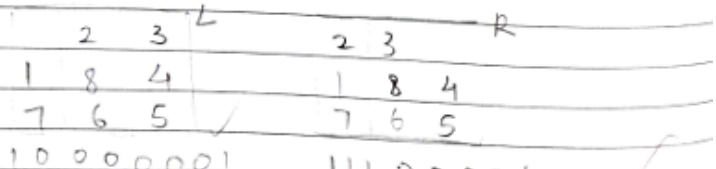
h(n) = 1 2 3 4 5 6 7 8

U : 1 1 0 0 0 0 0 1 = 3

L : 2 1 0 0 0 0 0 2 = 4

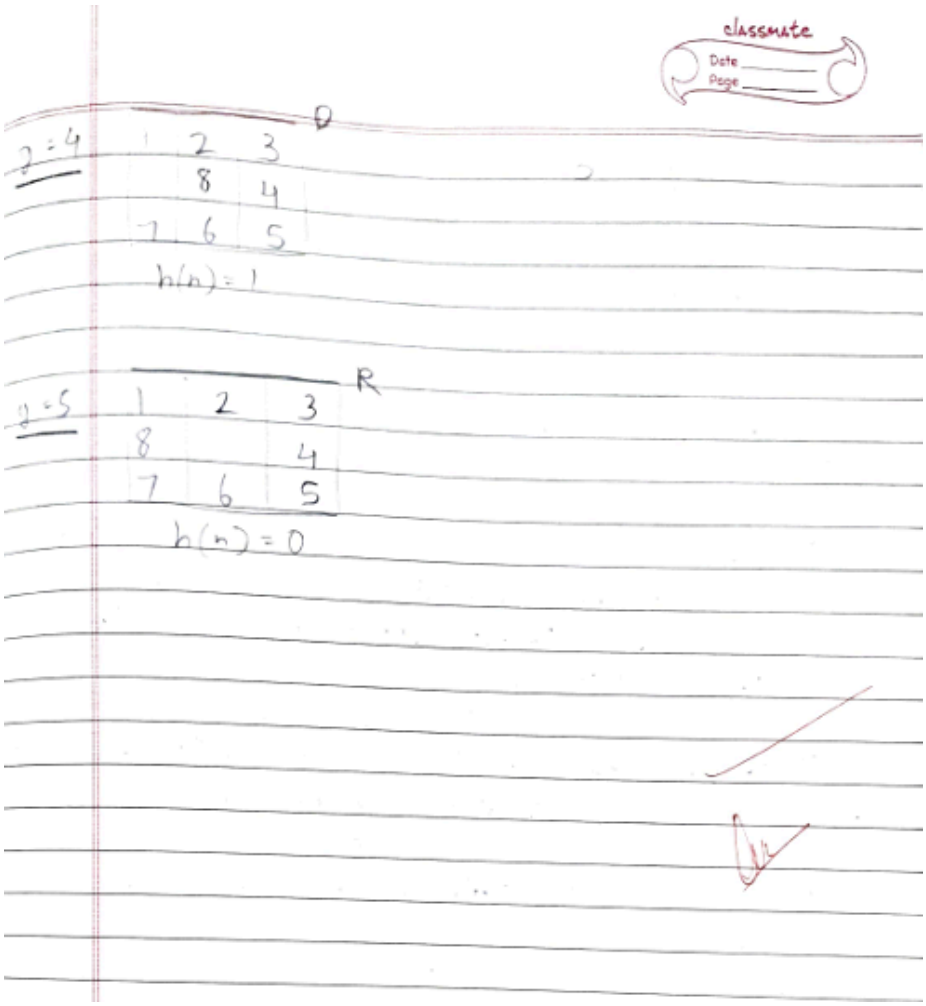
R : 1 1 0 1 0 0 0 2 = 5

g = 3



1 0 0 0 0 0 0 1 = 5

1 1 1 0 0 0 0 1 = 7



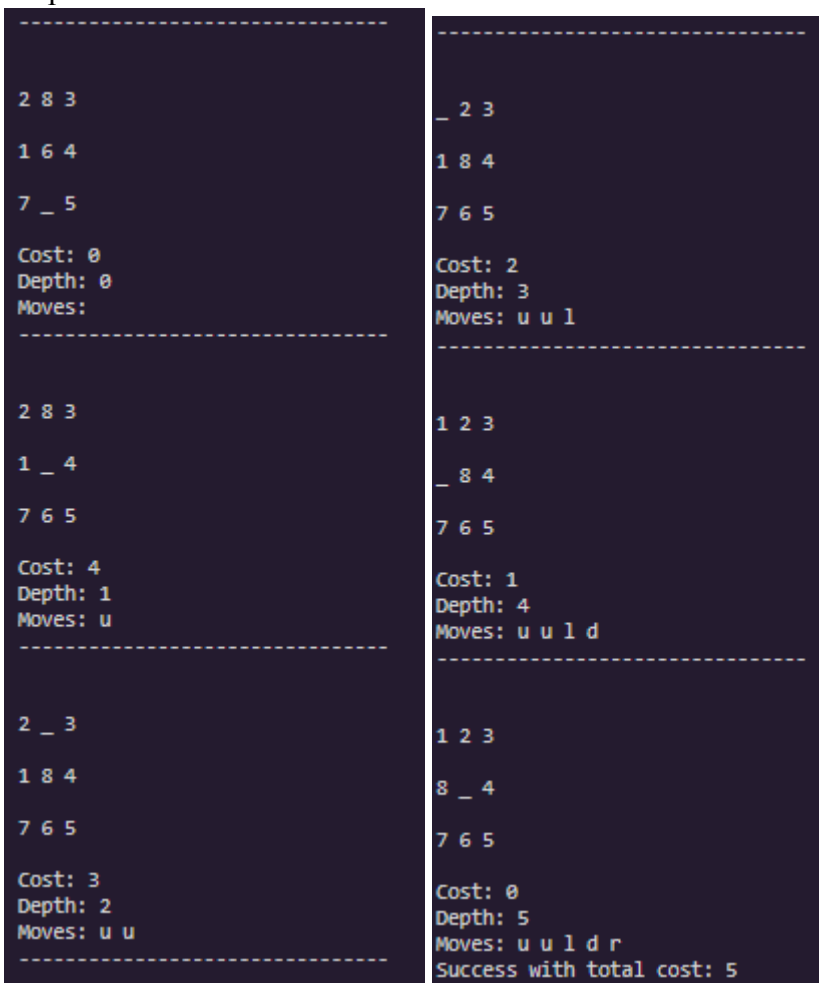
Code:

```
import heapq

def solve(src, target):
    queue = []
    heapq.heappush(queue, (0, src, 0, []))  # (cost, state, depth, path of moves)
    visited = {}
    visited[tuple(src)] = None  # Store the parent of the initial state as None

    while len(queue) > 0:
        cost, source, depth, moves = heapq.heappop(queue)
        print("--------------------------------")
        print_state(source)
        print("Cost:", cost)
        print("Depth:", depth)
        print("Moves:", " ".join(moves))

        if source == target:
```

```
            total_cost = cost + depth
            print("Success with total cost:", total_cost)
            print("Path to target:", reconstruct_path(visited, source))
            return

        poss_moves_to_do = possible_moves(source, visited)
        for move, direction in poss_moves_to_do:
            move_tuple = tuple(move)
            if move_tuple not in visited:
                move_cost = calculate_cost(move, target)
                heapq.heappush(queue, (move_cost, move, depth + 1, moves + [direction]))
                visited[move_tuple] = tuple(source)  # Record the parent state

def print_state(state):
    for i in range(9):
        if i % 3 == 0:
            print("\n")
        if state[i] == 0:
            print("_ ", end="")
        else:
            print(str(state[i]) + " ", end="")
    print("\n")

def possible_moves(state, visited_states):
    b = state.index(0)  # Index of empty spot
    directions = []
    # Add all the possible directions with corresponding moves
    if b not in [0, 1, 2]:  # Up
        new_state = gen(state, 'u', b)
        directions.append((new_state, 'u'))
    if b not in [6, 7, 8]:  # Down
        new_state = gen(state, 'd', b)
        directions.append((new_state, 'd'))
    if b not in [0, 3, 6]:  # Left
        new_state = gen(state, 'l', b)
        directions.append((new_state, 'l'))
    if b not in [2, 5, 8]:  # Right
        new_state = gen(state, 'r', b)
        directions.append((new_state, 'r'))


    # Filter out visited states
    return [(move, direction) for move, direction in directions if tuple(move) not in visited_states]


def gen(state, move, b):
    temp = state.copy()
```

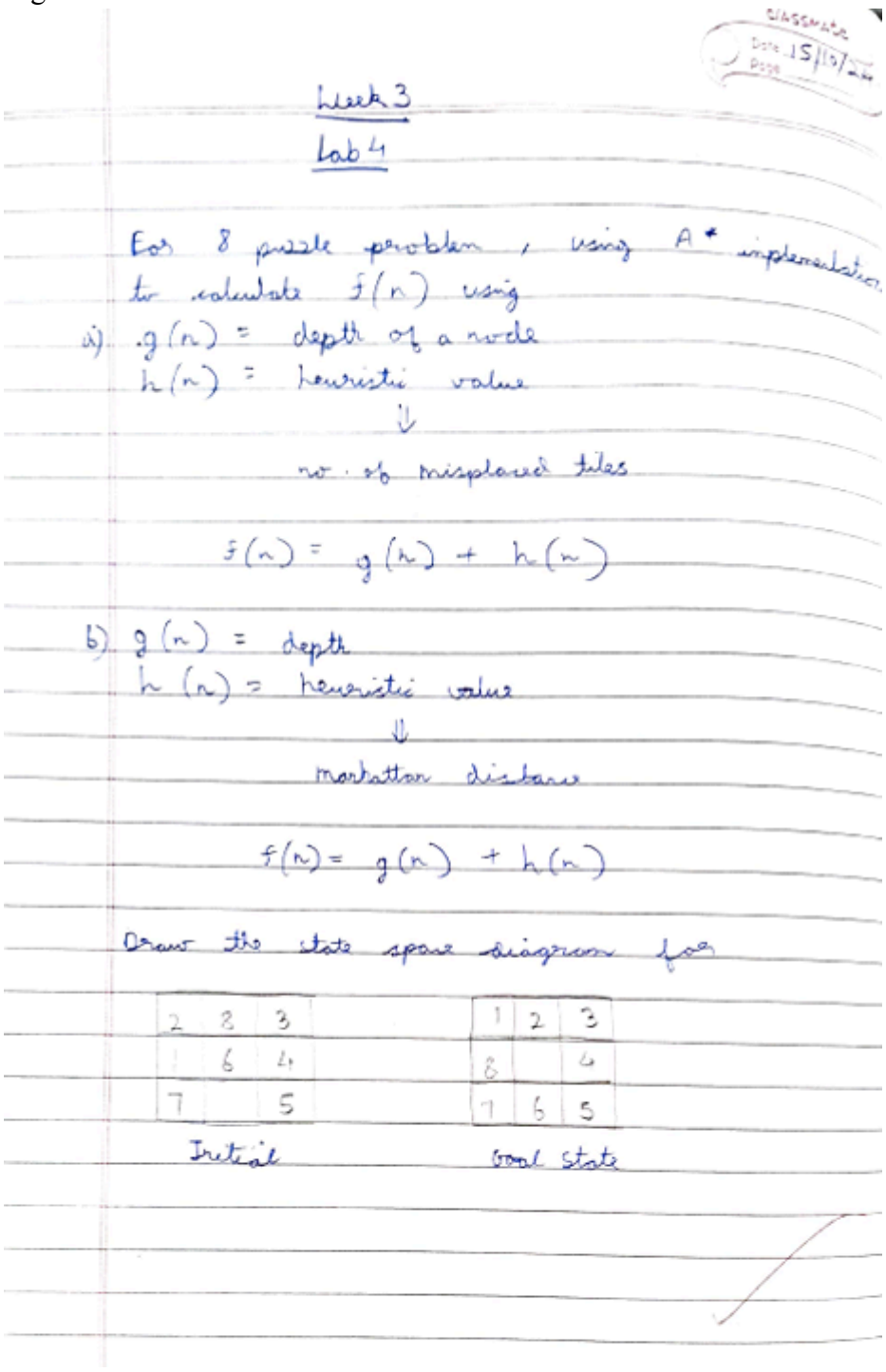
```
    if move == 'd':
        temp[b], temp[b + 3] = temp[b + 3], temp[b]
    elif move == 'u':
        temp[b], temp[b - 3] = temp[b - 3], temp[b]
    elif move == 'l':
        temp[b], temp[b - 1] = temp[b - 1], temp[b]
    elif move == 'r':
        temp[b], temp[b + 1] = temp[b + 1], temp[b]
    return temp


def calculate_cost(state, target):
    cost = 0
    for i in range(len(state)):
        if state[i] != 0:  # Exclude the empty space
            target_index = target.index(state[i])
            cost += abs(target_index // 3 - i // 3) + abs(target_index % 3 - i % 3)
    return cost


def reconstruct_path(visited, target):
    path = []
    current = tuple(target)
    while current is not None:
        path.append(current)
        current = visited.get(current)  # Use get to avoid KeyError
    return path[::-1]  # Return reversed path


# Example usage
src = [2, 8, 3, 1, 6, 4, 7, 0, 5]
target = [1, 2, 3, 8, 0, 4, 7, 6, 5]
solve(src, target)
```

Output:


2 8 3
1 6 4
7 _ 5
Cost: 0
Depth: 0
Moves:
2 8 3
1 _ 4
7 6 5
Cost: 4
Depth: 1
Moves: u
2 _ 3
1 8 4
7 6 5
Cost: 3
Depth: 2
Moves: u u
_ 2 3
1 8 4
7 6 5
Cost: 2
Depth: 3
Moves: u u l
1 2 3
_ 8 4
7 6 5
Cost: 1
Depth: 4
Moves: u u l d
1 2 3
8 _ 4
7 6 5
Cost: 0
Depth: 5
Moves: u u l d r
Success with total cost: 5

Implement A* search algorithm with misplaced tiles

Algorithm:



Week 3

Lab 4

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

a). $g(n)$ = depth of a node

$h(n)$ = heuristic value

⇓

no. of misplaced tiles

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

b) $g(n)$ = depth

$h(n)$ = heuristic value

⇓

manhattan distance

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

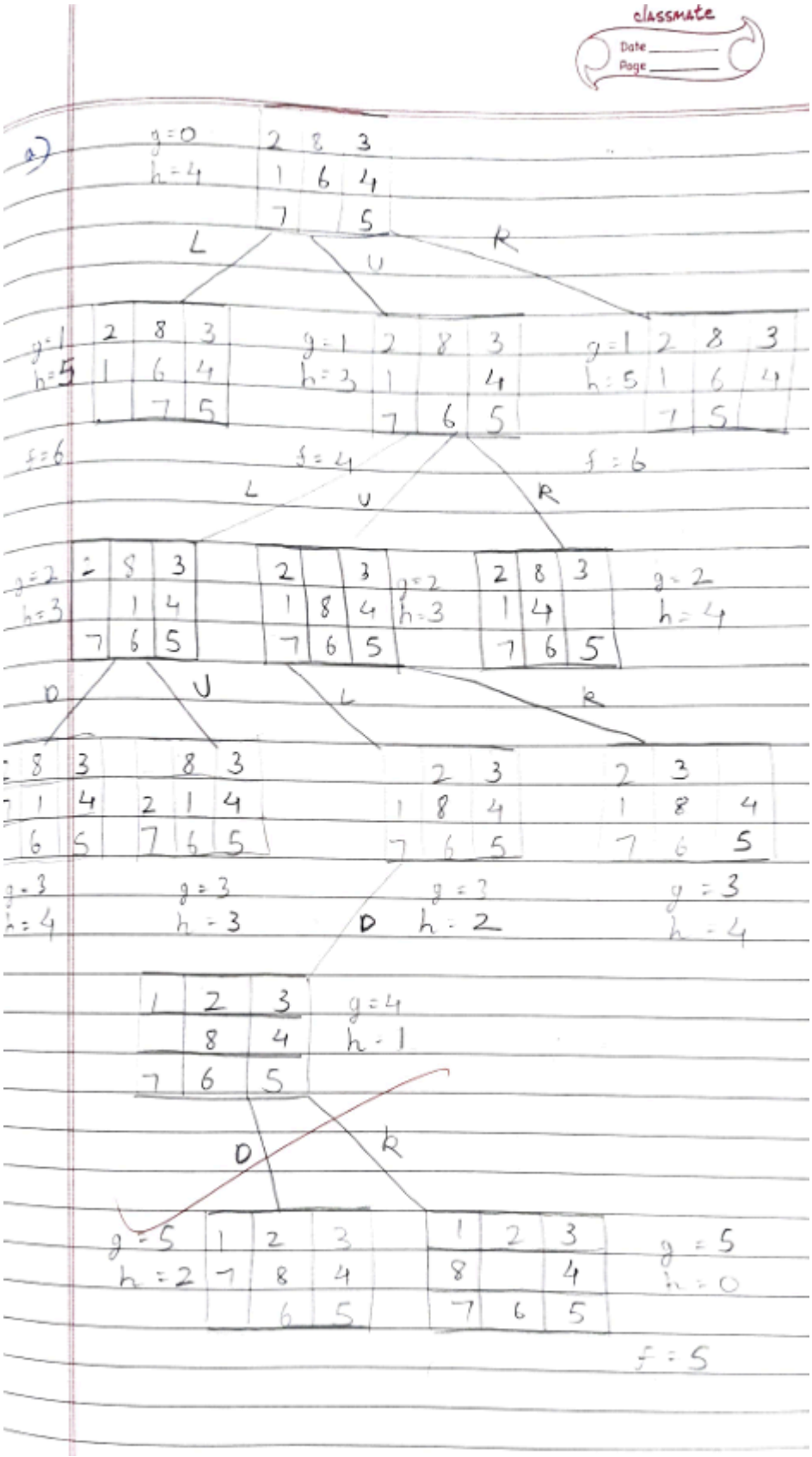
Draw the state space diagram for

2	8	3
1	6	4
7		5

Initial

1	2	3
8		4
7	6	5

Goal state


classmate
Date
Page
a)
g=0
h=4
2 8 3
1 6 4
7 5
L
U
R
g=1
h=5
2 8 3
1 6 4
7 5
f=6
g=1
h=3
2 8 3
1 4
7 6 5
f=4
g=1
h=5
2 8 3
1 6 4
7 5
f=6
L
U
R
g=2
h=3
8 3
1 4
7 6 5
2 3
1 8 4
7 6 5
g=2
h=3
2 8 3
1 4
7 6 5
g=2
h=4
D
U
L
R
8 3
1 4
6 5
8 3
2 1 4
7 6 5
2 3
1 8 4
7 6 5
2 3
1 8 4
7 6 5
g=3
h=4
g=3
h=3
g=3
D h=2
g=3
h=4
1 2 3
8 4
7 6 5
g=4
h=1
D
R
g=5
h=2
1 2 3
7 8 4
6 5
1 2 3
8 4
7 6 5
g=5
h=0
f=5

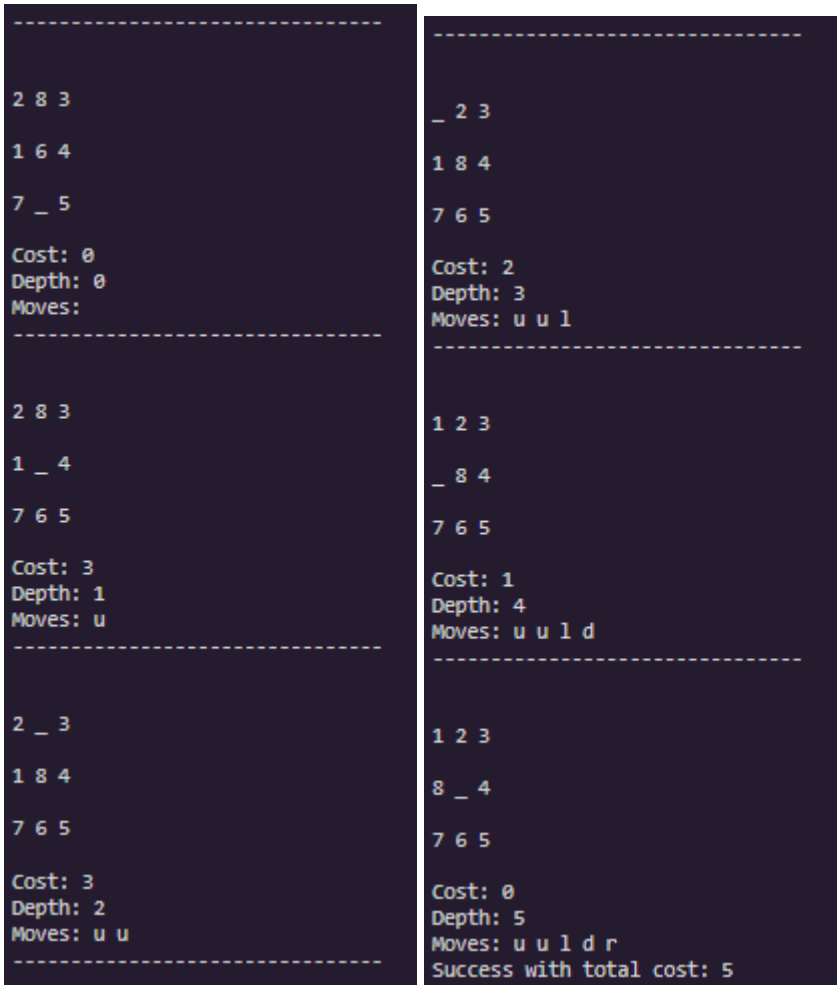
Code:

```
import heapq


def solve(src, target):
    queue = []
    heapq.heappush(queue, (0, src, 0, []))  # (cost, state, depth, path of moves)
    visited = {}
    visited[tuple(src)] = None  # Store the parent of the initial state as None

    while len(queue) > 0:
        cost, source, depth, moves = heapq.heappop(queue)
        print("------------------------------")
        print_state(source)
        print("Cost:", cost)
        print("Depth:", depth)
        print("Moves:", " ".join(moves))


        if source == target:
            total_cost = cost + depth
            print("Success with total cost:", total_cost)
            print("Path to target:", reconstruct_path(visited, source))
            return


        poss_moves_to_do = possible_moves(source, visited)
        for move, direction in poss_moves_to_do:
            move_tuple = tuple(move)
            if move_tuple not in visited:
                move_cost = calculate_cost(move, target)
                heapq.heappush(queue, (move_cost, move, depth + 1, moves + [direction]))
                visited[move_tuple] = tuple(source)  # Record the parent state


def print_state(state):
    for i in range(9):
        if i % 3 == 0:
            print("\n")
        if state[i] == 0:
            print("_ ", end="")
        else:
            print(str(state[i]) + " ", end="")
    print("\n")
```

```
def possible_moves(state, visited_states):
    b = state.index(0)  # Index of empty spot
    directions = []


    # Add all the possible directions with corresponding moves
    if b not in [0, 1, 2]:  # Up
        new_state = gen(state, 'u', b)
        directions.append((new_state, 'u'))
    if b not in [6, 7, 8]:  # Down
        new_state = gen(state, 'd', b)
        directions.append((new_state, 'd'))
    if b not in [0, 3, 6]:  # Left
        new_state = gen(state, 'l', b)
        directions.append((new_state, 'l'))
    if b not in [2, 5, 8]:  # Right
        new_state = gen(state, 'r', b)
        directions.append((new_state, 'r'))


    # Filter out visited states
    return [(move, direction) for move, direction in directions if tuple(move) not in visited_states]


def gen(state, move, b):
    temp = state.copy()
    if move == 'd':
        temp[b], temp[b + 3] = temp[b + 3], temp[b]
    elif move == 'u':
        temp[b], temp[b - 3] = temp[b - 3], temp[b]
    elif move == 'l':
        temp[b], temp[b - 1] = temp[b - 1], temp[b]
    elif move == 'r':
        temp[b], temp[b + 1] = temp[b + 1], temp[b]
    return temp


def calculate_cost(state, target):
    # Count the number of misplaced tiles
    cost = sum(1 for i in range(len(state)) if state[i] != target[i] and state[i] != 0)
    return cost


def reconstruct_path(visited, target):
```

```
    path = []
    current = tuple(target)
    while current is not None:
        path.append(current)
        current = visited.get(current)  # Use get to avoid KeyError
    return path[::-1]  # Return reversed path


# Example usage
src = [2, 8, 3, 1, 6, 4, 7, 0, 5]
target = [1, 2, 3, 8, 0, 4, 7, 6, 5]
solve(src, target)
```

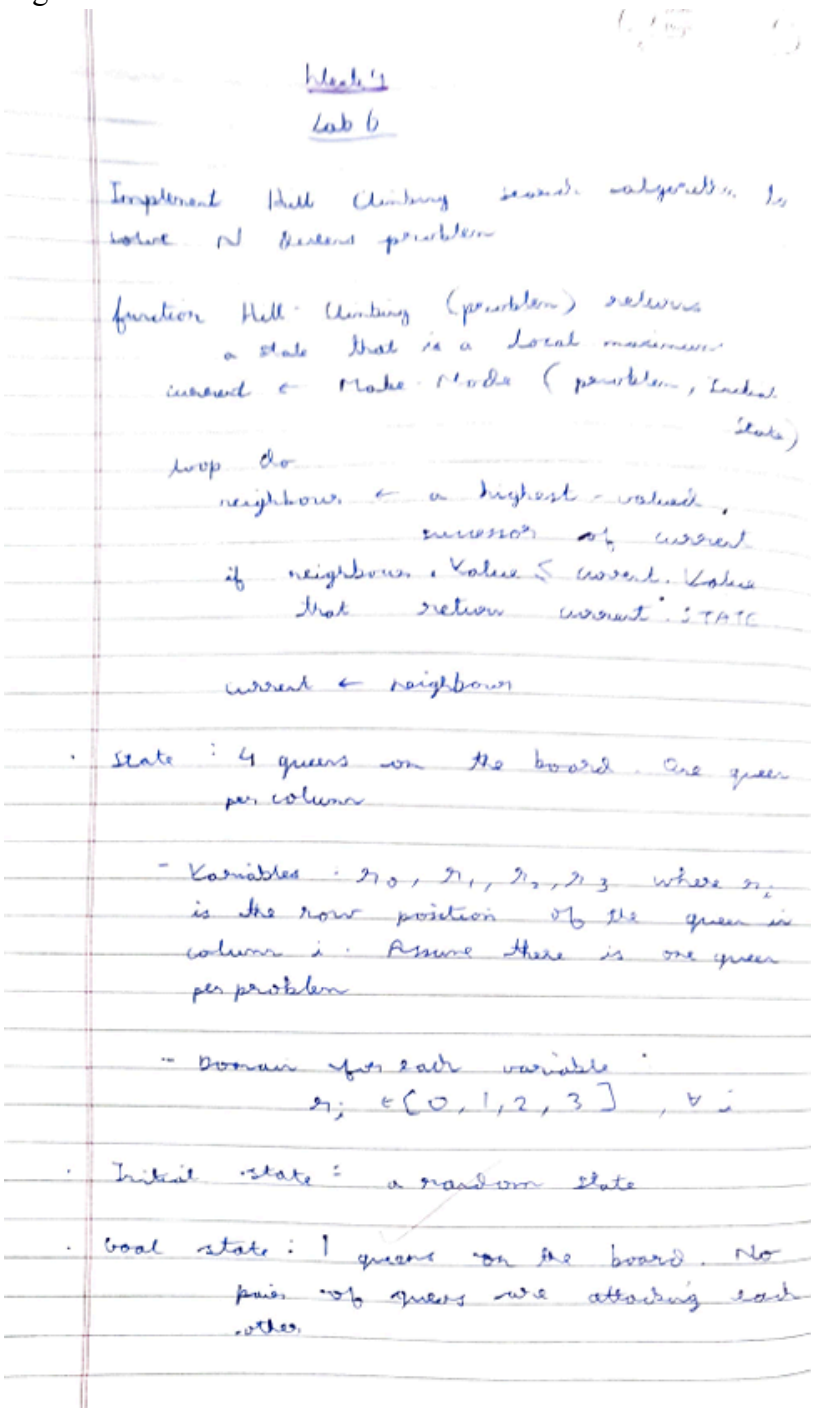
Output:



Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



Hill's

Lab 6

Implement Hill Climbing search algorithm to solve N Queens problem

function Hill Climbing (problem) returns a state that is a local maximum

current ← Make Node (problem, Initial State)

loop do

neighbour ← a highest-valued successor of current

if neighbour.Value ≤ current.Value then return current.STATE

current ← neighbour

- State : 4 queens on the board. One queen per column

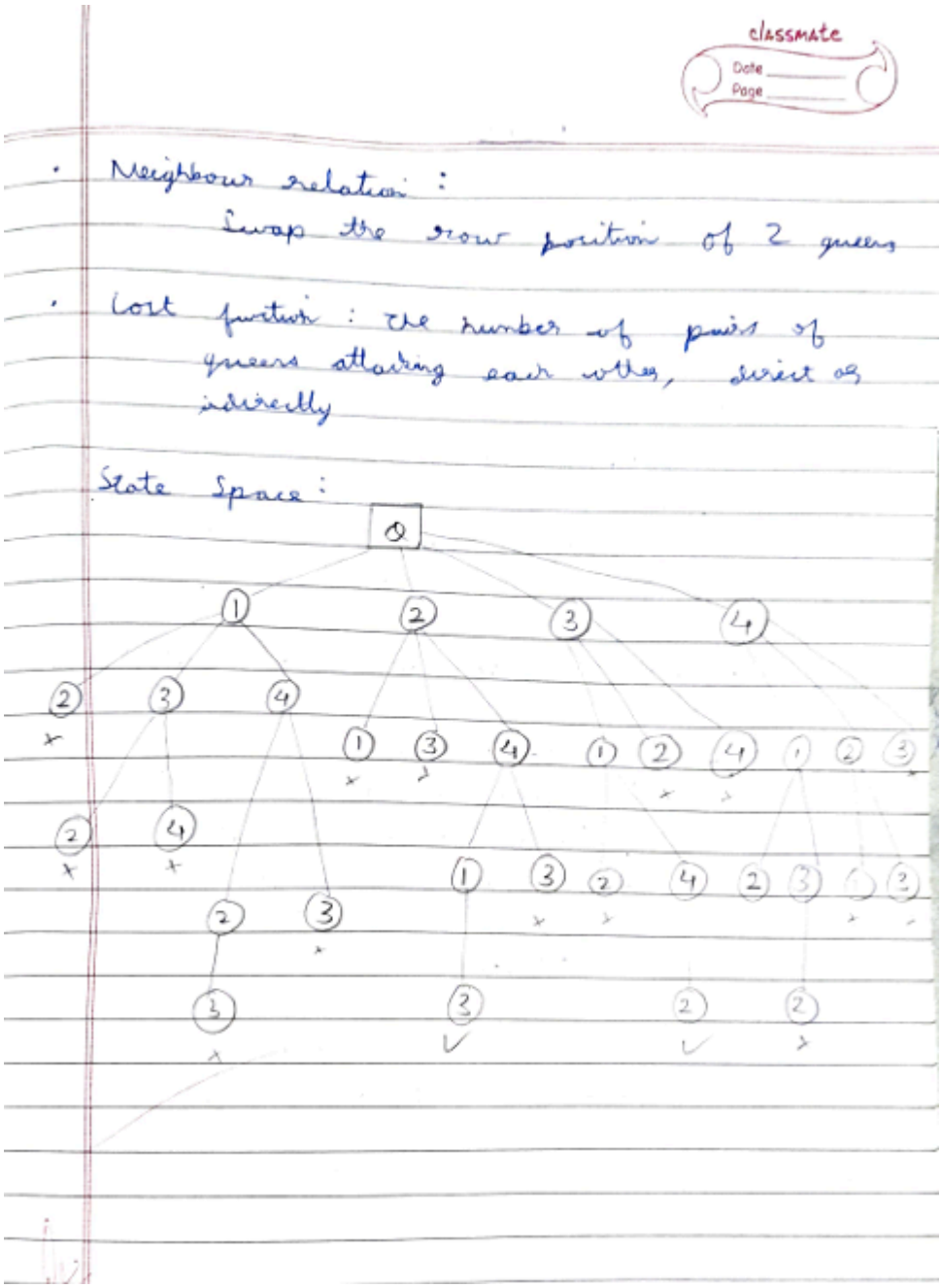
- Variables : x_0, x_1, x_2, x_3 where x_i is the row position of the queen in column i. Assume there is one queen per problem

- 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.



Code:

```
import random

def calculate_cost(state):
    """Calculate the number of conflicts in the current state."""
    cost = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
```

```
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                cost += 1
    return cost


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


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


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


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


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


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


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

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


solution = None


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


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

Output:

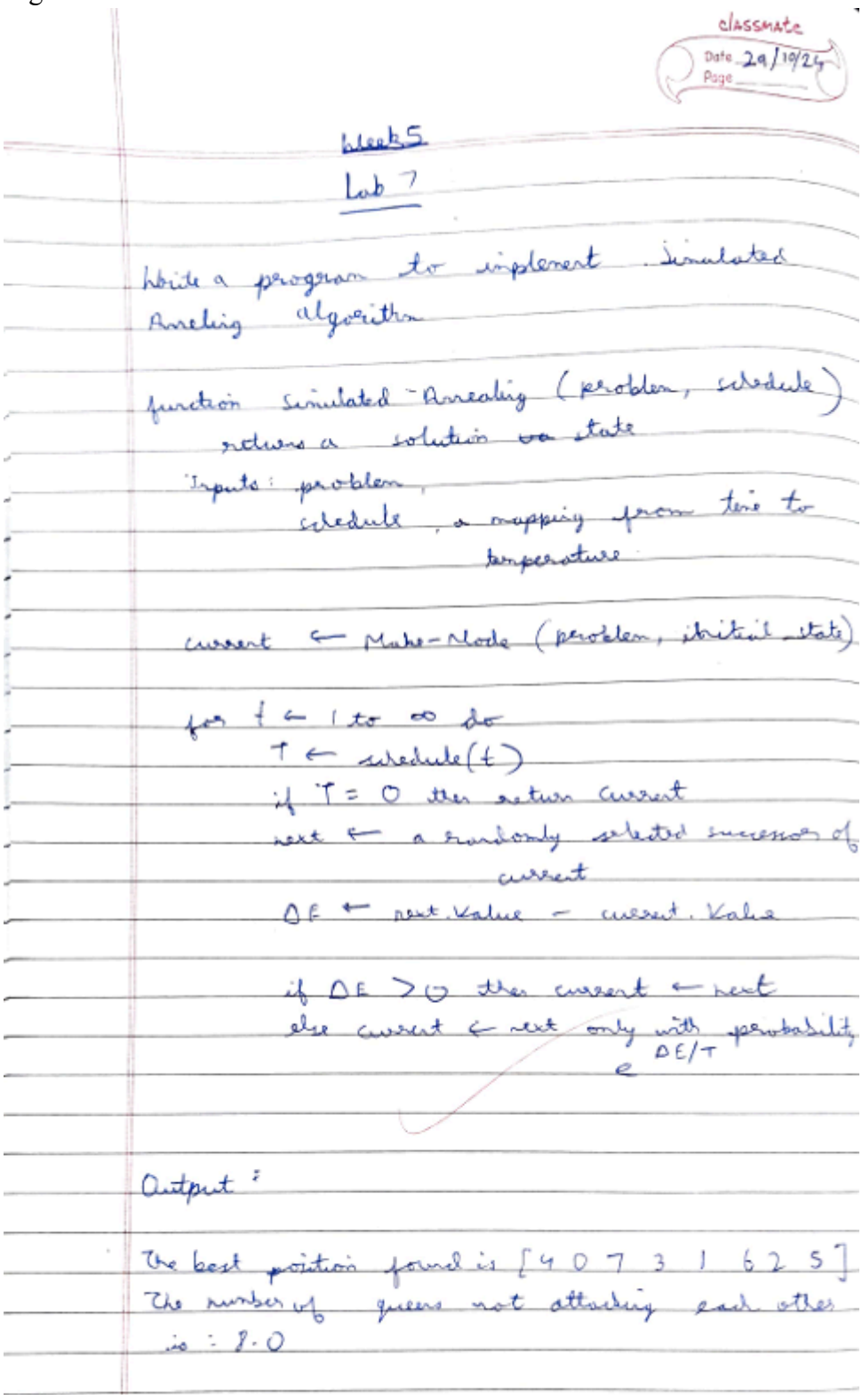
```
Enter the number of queens (N): 4
Iteration 0: Current state: [3, 0, 2, 0], Cost: 2
Iteration 1: Current state: [3, 0, 2, 1], Cost: 1
Local maximum reached at iteration 2. Restarting...
Iteration 0: Current state: [2, 2, 0, 3], Cost: 2
Iteration 1: Current state: [1, 2, 0, 3], Cost: 1
Local maximum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

```
Enter the number of queens (N): 8
Iteration 0: Current state: [1, 3, 0, 6, 1, 7, 2, 4], Cost: 3
Iteration 1: Current state: [1, 5, 0, 6, 1, 7, 2, 4], Cost: 2
Iteration 2: Current state: [1, 5, 0, 6, 3, 7, 2, 4], Cost: 0
Solution found: [1, 5, 0, 6, 3, 7, 2, 4]
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



classmate

Date 29/10/24

Page

Week 5

Lab 7

Write a program to implement Simulated Annealing algorithm

function simulated-Annealing (problem, schedule)
returns a solution on state

Inputs: problem,
schedule, a mapping from time to temperature

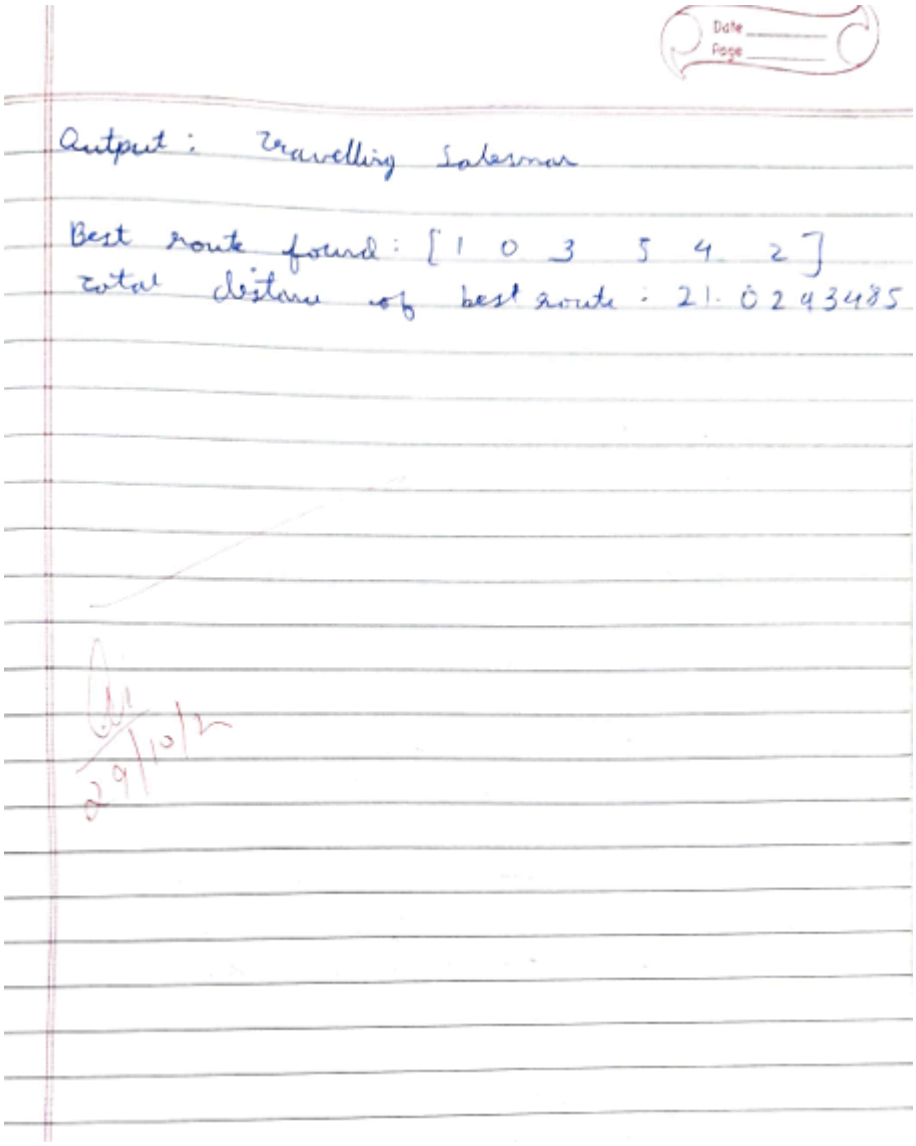
current ← Make-Node (problem, initial state)

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 $e^{\Delta E/T}$

Output :

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



Code:

```
import mlrose_hiive as mlrose
import numpy as np

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

```
            queen_not_attacking += 1
    if (queen_not_attacking == 7):
        queen_not_attacking += 1
    return queen_not_attacking


objective = mlrose.CustomFitness(queens_max)


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


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


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


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

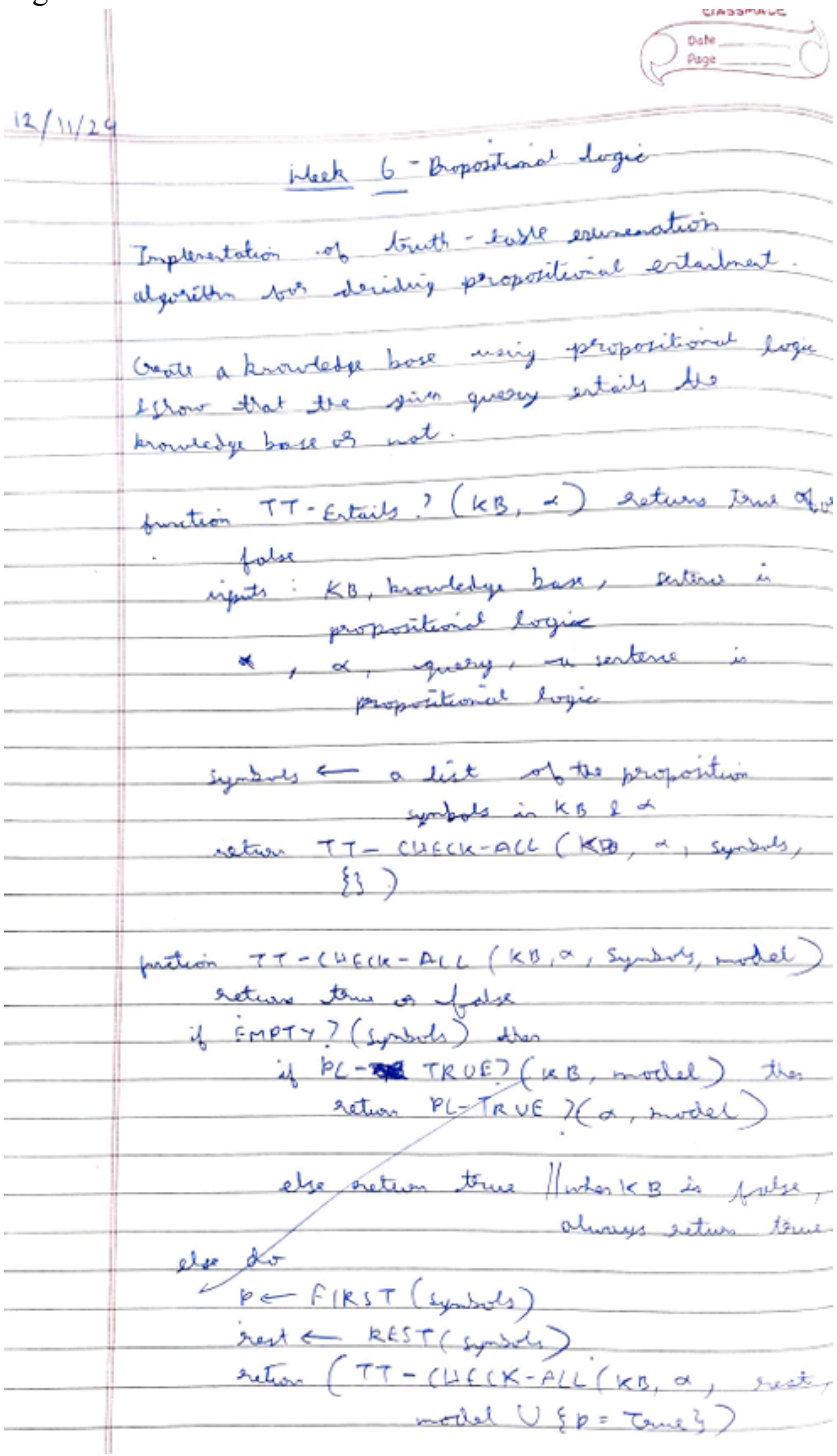
Output:

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

Program 6

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

Algorithm:



12/11/24

Week 6 - Propositional logic

Implementation of truth-table enumeration algorithm for deciding propositional entailment.

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

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

inputs: KB, knowledge base, sentence in propositional logic

α, query, a sentence in propositional logic

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

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

function TT-CHECK-ALL (KB, α, symbols, model)

return true or false

if EMPTY? (symbols) then

if PL-TRUE? (KB, model) then return PL-TRUE? (α, model)

else return true // 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:

α = A ∨ B KB = (A ∨ C) ∧ (B ∨ ¬C

Checking that KB ⊨ α

A	B	C	A ∨ C	A ∨ ¬C	KB	α
false	F	F	F	T	F	F
false	F	T	T	F	F	F
F	T	F	F	T	F	T
F	T	T	T	T	T	T
T	F	F	T	T	T	T
T	F	T	T	F	F	T
T	T	F	T	T	T	T
T	T	T	T	T	T	T

Explanation:

P	Q	¬P	P ∧ Q	P ∨ Q	P ⇔ Q	P ⇒
F	F	T	F	F	T	T
F	T	T	F	T	F	T
T	F	F	F	T	F	F
T	T	F	T	T	T	T

O/P seen

12/11/24

Code:

```
from itertools import product

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


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


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

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

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

        # Highlight rows where both KB and α are true
```

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


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


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

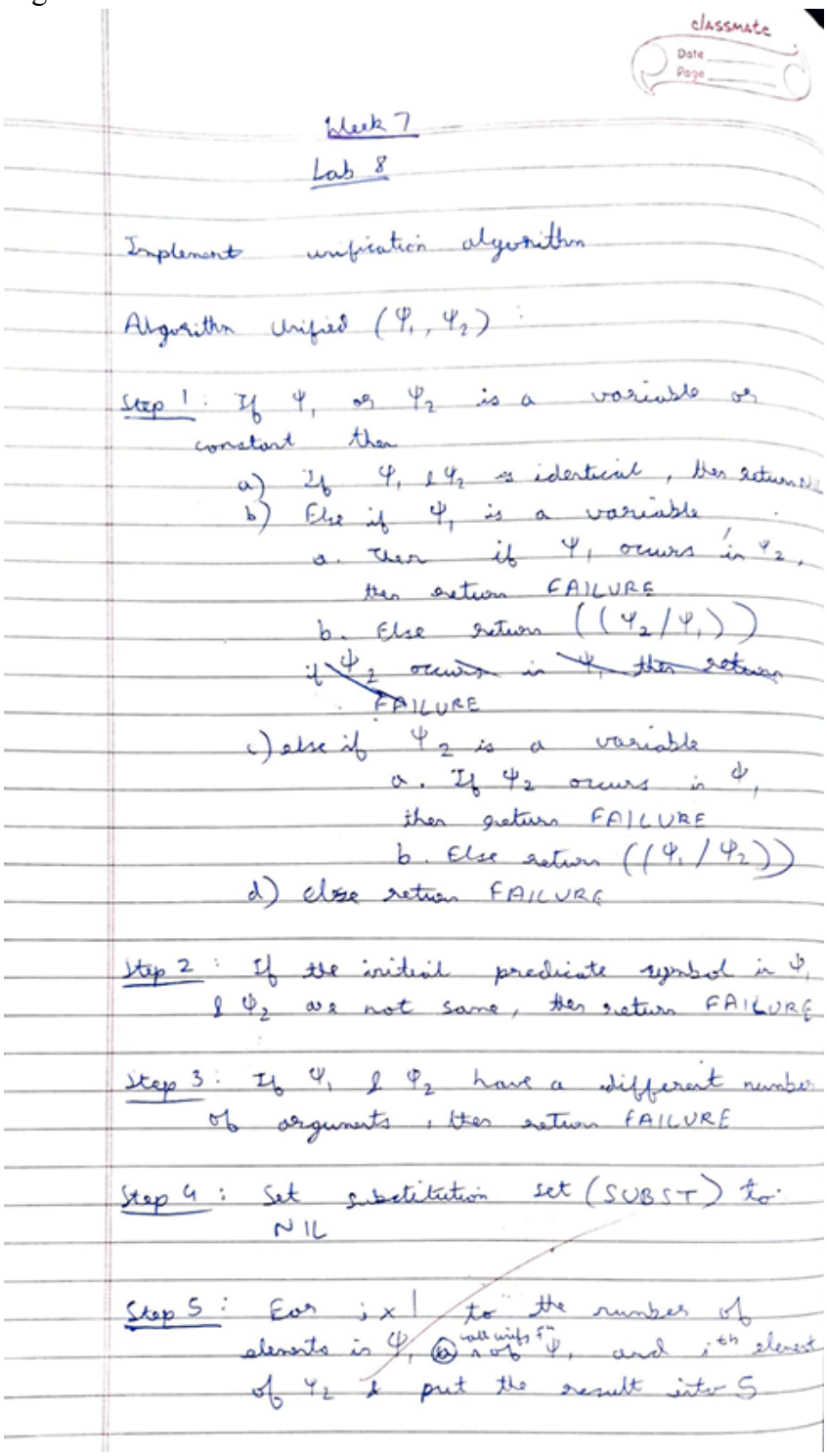
Output:

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

Program 7

Implement unification in first order logic

Algorithm:



Week 7

Lab 8

Implement unification algorithm

Algorithm Unified (Ψ_1, Ψ_2) :

Step 1 : If Ψ_1 or Ψ_2 is a variable or constant then

a) If Ψ_1 & 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)

if ~~Ψ_2 occurs in Ψ_1 then return~~ FAILURE

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 the number of elements in Ψ_1, call Unify function with the ith element of Ψ_1 & ith element of Ψ_2 & 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 & L2
b. Subst = APPEND(S, SUBST)

Step 6: Return SUBST

Q1) $\psi_1 = P(f(a), g(y))$
$\psi_2 = P(x, x)$

Predicates f & g are distinct and not same, they cannot be substituted at the same variable x.

Unification Fail

Q2) $\psi_1 = P(b, x, f(g(z)))$
$\psi_2 = P(z, f(y), f(y))$

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

24

Code:

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


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


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


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


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


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


    # If they don't unify
    return None


def unify_variable(var, expr, subst):
    """Handle variable unification."""
    if var in subst:  # Variable already substituted
        return unify(subst[var], expr, subst)
    if occurs_check(var, expr, subst):  # Occurs-check
```

```
        return None
    subst[var] = expr
    return subst

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

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

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


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

# Testing the algorithm with the given cases
if __name__ == "__main__":
    # Case 1: p(f(a), g(b)) and p(x, x)
    expr1 = ("p", [("f", ["a"]), ("g", ["b"])])
    expr2 = ("p", ["x", "x"])
    result = unify(expr1, expr2)
    print("Case 1 Result:", result)


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

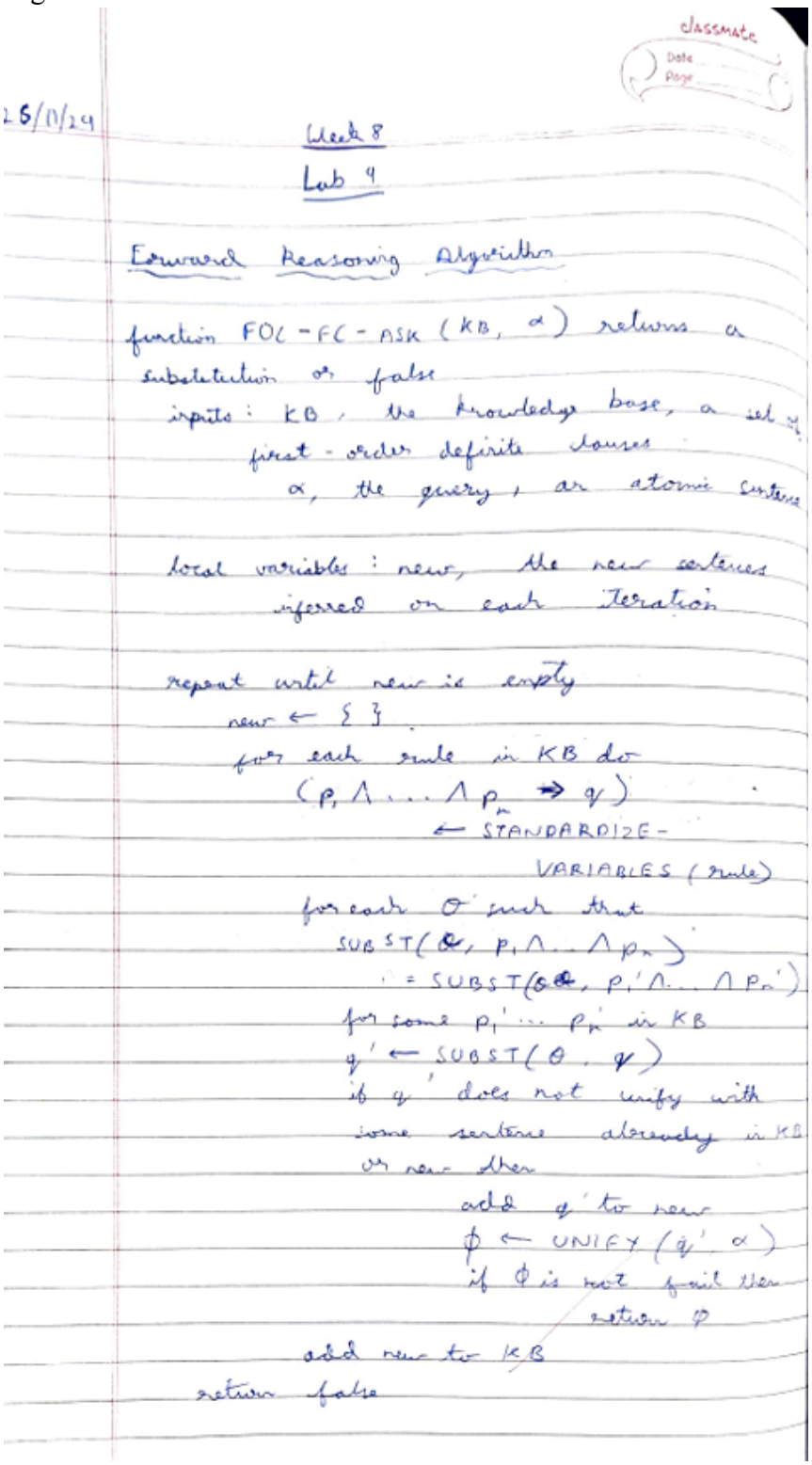
Output:

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

Program 8

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

Algorithm:



26/11/24

Week 8

Lab 9

Forward Reasoning Algorithm

function 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 $\leftarrow$ { }

for each rule in KB do

$(p_1 \wedge \ldots \wedge p_n \Rightarrow q)$

$\leftarrow$ STANDARDIZE-VARIABLES (rule)

for each θ such that

SUBST(θ, $p_1 \wedge \ldots \wedge p_n$)

= SUBST(θ, $p_1' \wedge \ldots \wedge p_n'$)

for some $p_1' \ldots p_n'$ 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)

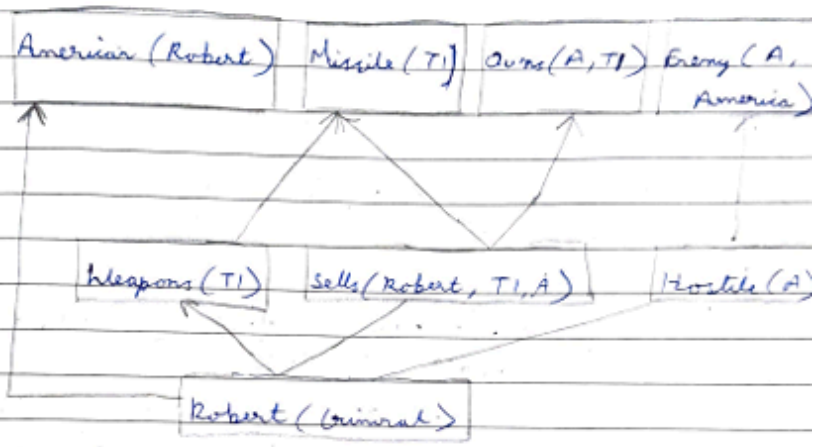
if ϕ is not fail then return ϕ

add new to KB

return false

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

Prove that Robert is criminal



American (p) ∧ {Weapon (q) ∧ Sells (p, q, r)
∧ Hostile (r)
⇒ Criminal (p)
∃ x Owns (A, x) ∧ Missiles (x)
Owns (A, T1)
Missile (T1)
∀ x Missile (x) ∧ Owns (A, x) ⇒ Sells (Robert, x, A)
Missile (x) ⇒ Weapon (x)
∀ x Enemy (x, America) ⇒ Hostile (x)
American (robert)
Enemy (A, America)
To prove Criminal (Robert)

Code:

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


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


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

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


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


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


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

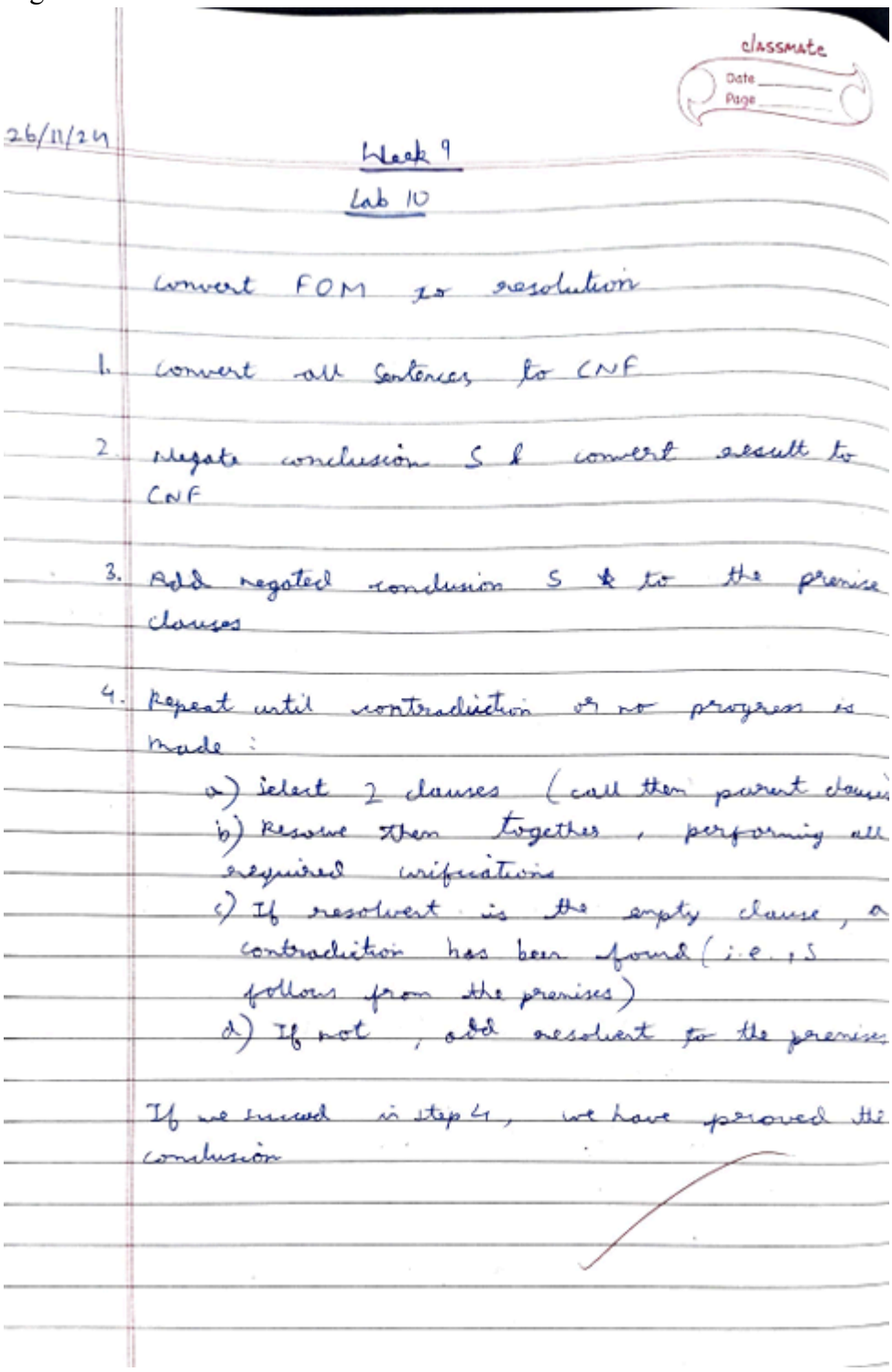
Output:

```
Robert is a criminal.
```

Program 9

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

Algorithm:



26/11/24

Week 9

Lab 10

Convert FOM to resolution

1. Convert all sentences to CNF
2. Negate conclusion S & convert result to CNF
3. Add negated conclusion S & to the premise clauses
4. Repeat until contradiction or no progress is made :
   a) Select 2 clauses (call them parent clauses)
   b) Resolve them together, performing all required unifications
   c) If resolvent is the empty clause, a contradiction has been found (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

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. $likes(John, Peanuts)$

$\neg likes(John, Peanuts)$ — $\neg food(x) \vee likes(John, x)$ $\{Peanuts/x\}$

$\neg food(Peanuts)$ — $\neg eats(y, z) \vee killed(y) \vee food(z)$ $\{Peanuts/z\}$

$\neg eats(y, Peanuts) \vee killed(y)$ — $eats(Anil, Peanuts)$ $\{Anil/y\}$

$killed(Anil)$ — $\neg alive(k) \vee \neg killed(k)$ $\{Anil/k\}$

$\neg alive(Anil)$ — $alive(Anil)$

$\{\}$ Hence Proved

3/12/14

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'}")
```

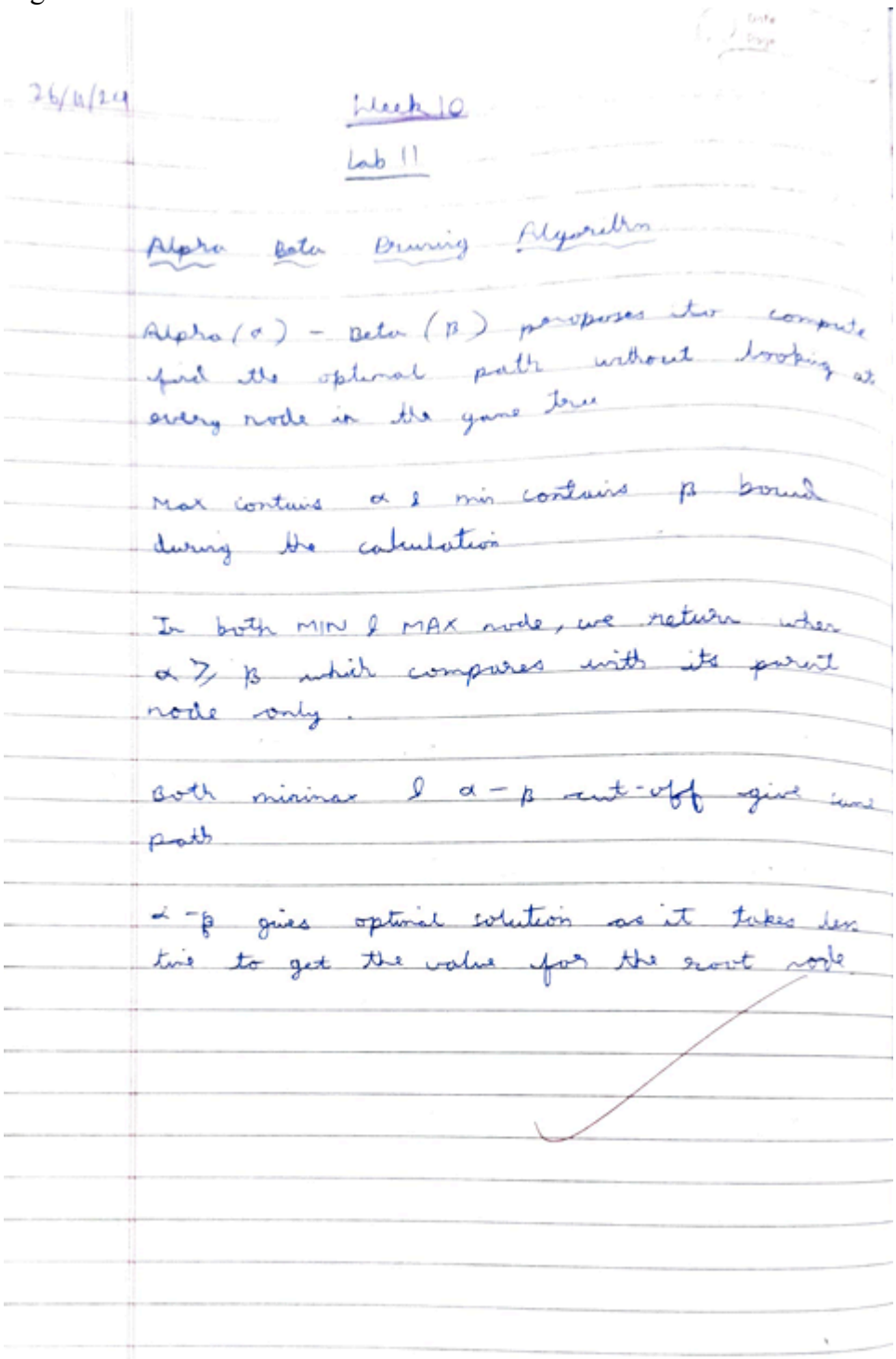
Output:

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

Program 10

Implement Alpha-Beta Pruning.

Algorithm:



26/11/24

Week 10

Lab 11

Alpha Beta Pruning Algorithm

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

Max contains α & min contains β bound during the calculation

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

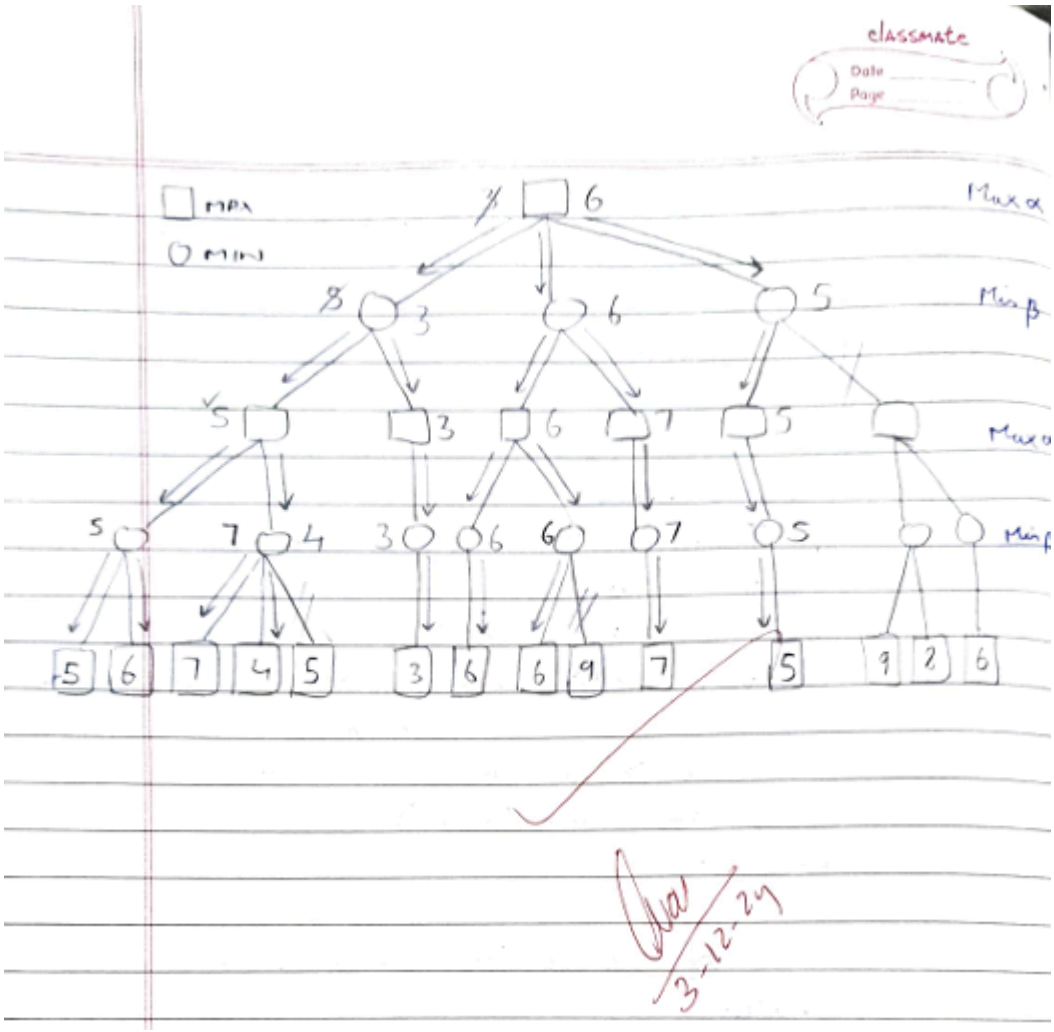
Both minimax & α - β cut-off give same path

α - β gives optimal solution as it takes less time to get the value for the root node

function Alpha-Beta-Search (state) returns an action
    v ← MAX-VALUE (state, −∞, +∞)
    return the action in actions (state)
        with value v

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

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



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
```