VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Shlok Shivaram Iyer (1BM22CS260)

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 **Shlok Shivaram Iyer (1BM22CS260)),** 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.

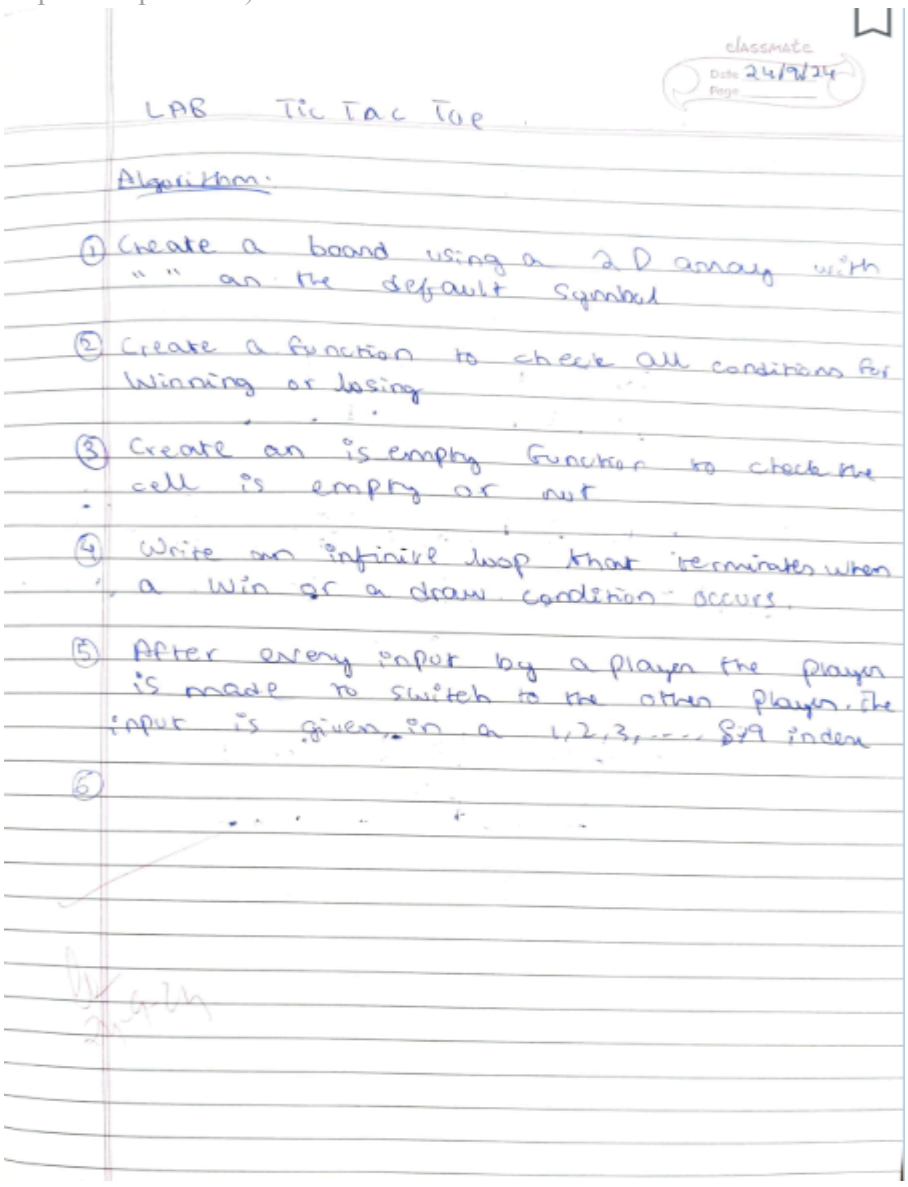
Sonika Sharma D Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/shlokiyer123/AILAB5E/tree/main

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

Algorithm:
(You should provide the screen shot of your observation book of Algorithm/Logic/Solving of respective problem)


LAB Tic Tac Toe

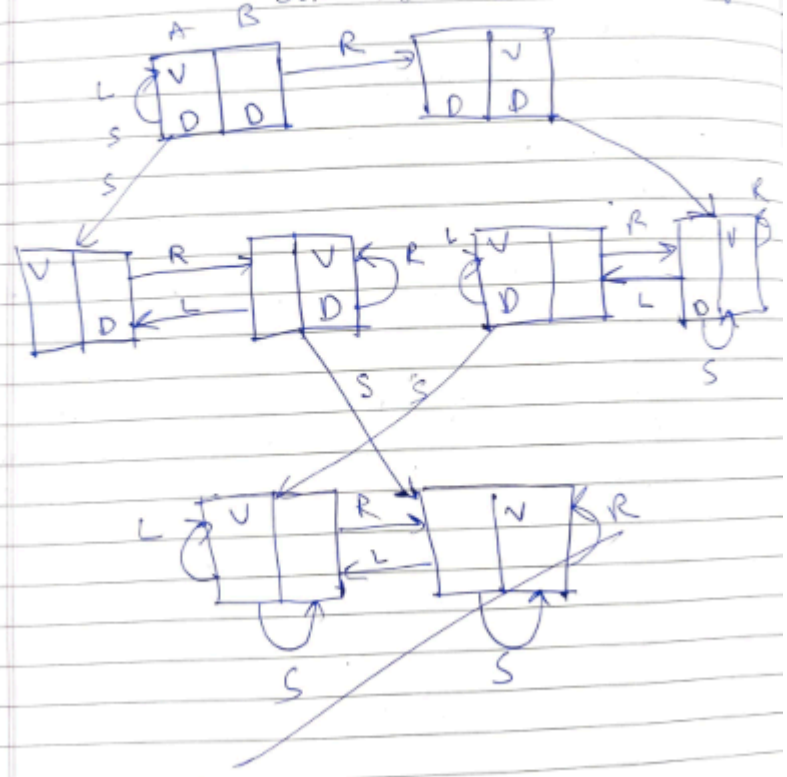
Algorithm:

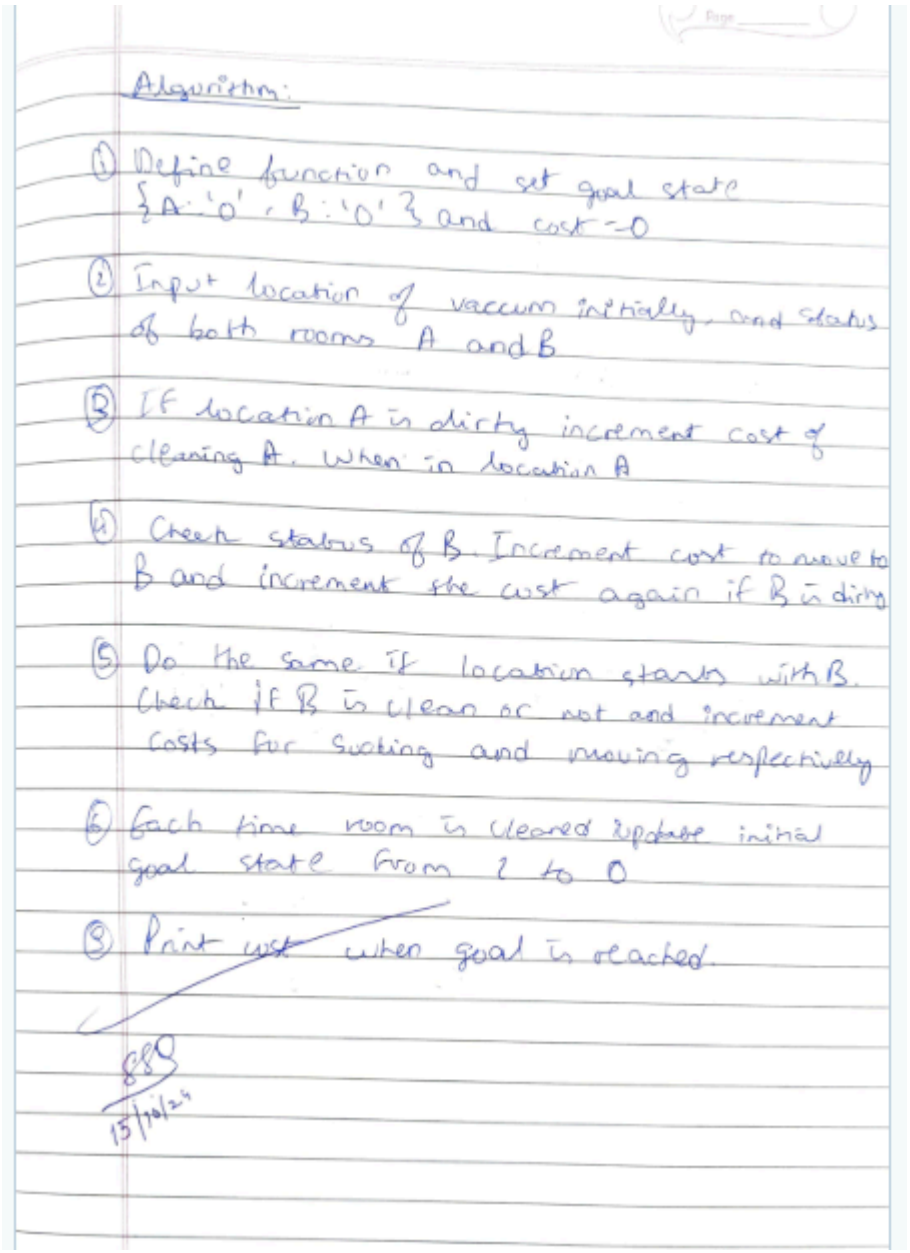
1. Create a board using a 2D array with " " as the default symbol
2. Create a function to check all conditions for winning or losing
3. Create an is empty function to check the cell is empty or not
4. Write an infinite loop that terminates when a win or a draw condition occurs.
5. After every input by a player the player is made to switch to the other player. The input is given in a 1, 2, 3, ... 8, 9 index
6.

LAB-2 - Vacuum-cleaner

function REFLEX_VACCUM-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





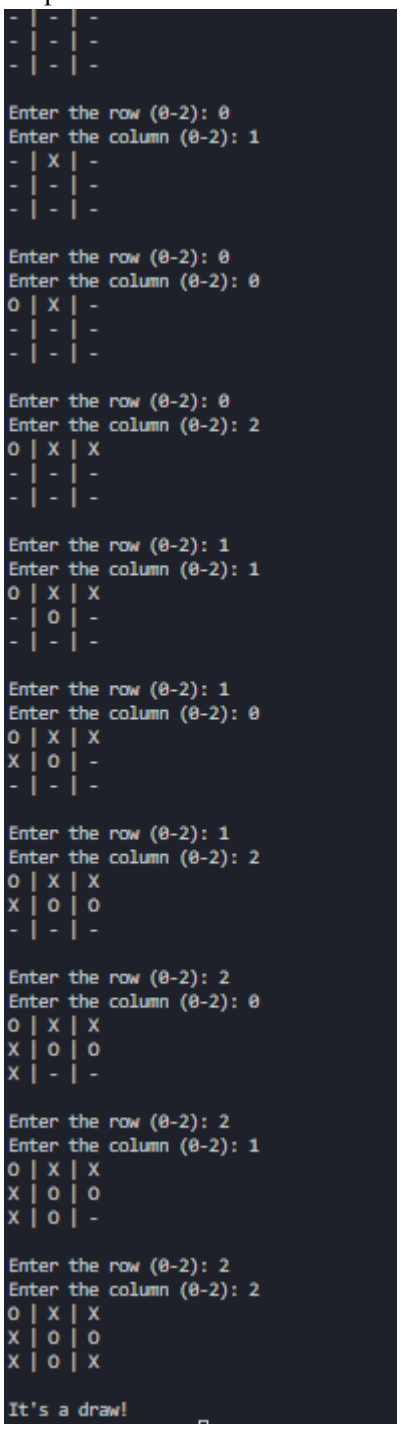
Code: #TIC TAC TOE

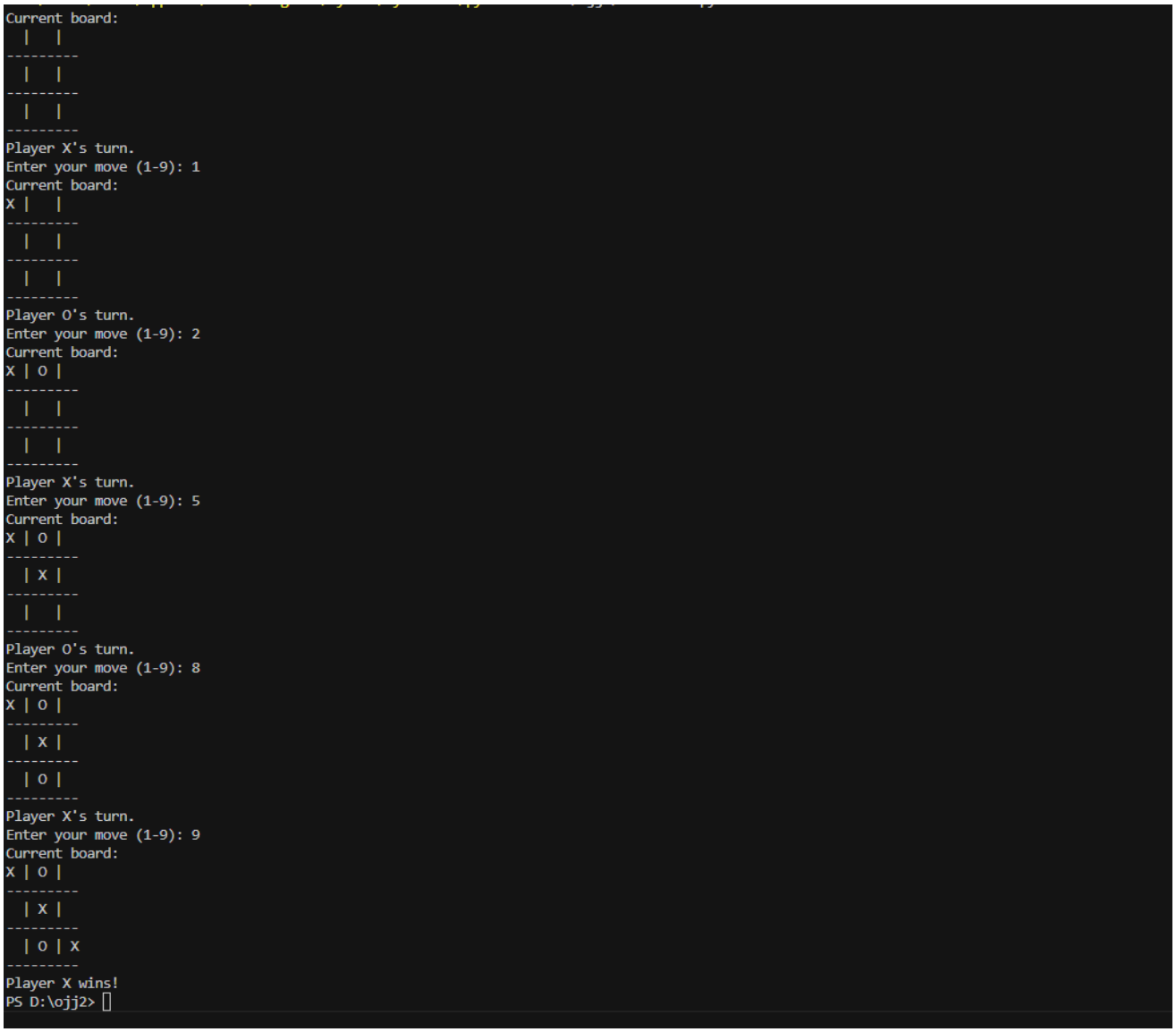
```
def show_arr(arr):
    print("Current board:")
    for row in arr:
        print(" | ".join(cell if cell != "" else " " for cell in row))
        print("-" * 9)

def is_cell_empty(arr, row, col):
    return arr[row][col] == ""

def wincheck(arr):
    # Check rows
```

```
    for row in arr:
        if row[0] == row[1] == row[2] and row[0] != "":
            return "Win"

    # Check columns
    for col in range(3):
        if arr[0][col] == arr[1][col] == arr[2][col] and arr[0][col] != "":
            return "Win"

    # Check diagonals
    if arr[0][0] == arr[1][1] == arr[2][2] and arr[0][0] != "":
        return "Win"
    if arr[0][2] == arr[1][1] == arr[2][0] and arr[0][2] != "":
        return "Win"

    return "Not win"

def is_draw(arr):
    return all(cell != "" for row in arr for cell in row)

def tictactoe():
    arr = [["" for _ in range(3)] for _ in range(3)]
    players = ["X", "O"]
    current_player = 0

    while True:
        show_arr(arr)
        print(f"Player {players[current_player]}'s turn.")

        # Get player input
        move = -1
        while move < 0 or move > 8:
            move_input = input("Enter your move (1-9): ")
            if move_input.isdigit():
                move = int(move_input) - 1
                row, col = divmod(move, 3)
                if not (0 <= row < 3 and 0 <= col < 3):
                    print("Invalid move. Please enter a number between 1 and 9.")
                elif not is_cell_empty(arr, row, col):
                    print("Cell already taken. Try again.")
            else:
                print("Invalid input. Please enter a number between 1 and 9.")

        # Make the move
        arr[row][col] = players[current_player]

        # Check for a win
```

```
        if wincheck(arr) == "Win":
            show_arr(arr)
            print(f"Player {players[current_player]} wins!")
            break

        # Check for a draw
        if is_draw(arr):
            show_arr(arr)
            print("It's a draw!")
            break

        # Switch players
        current_player = 1 - current_player

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

Output:


- | - | -
- | - | -
- | - | -
Enter the row (0-2): 0
Enter the column (0-2): 1
- | X | -
- | - | -
- | - | -
Enter the row (0-2): 0
Enter the column (0-2): 0
O | X | -
- | - | -
- | - | -
Enter the row (0-2): 0
Enter the column (0-2): 2
O | X | X
- | - | -
- | - | -
Enter the row (0-2): 1
Enter the column (0-2): 1
O | X | X
- | O | -
- | - | -
Enter the row (0-2): 1
Enter the column (0-2): 0
O | X | X
X | O | -
- | - | -
Enter the row (0-2): 1
Enter the column (0-2): 2
O | X | X
X | O | O
- | - | -
Enter the row (0-2): 2
Enter the column (0-2): 0
O | X | X
X | O | O
X | - | -
Enter the row (0-2): 2
Enter the column (0-2): 1
O | X | X
X | O | O
X | O | -
Enter the row (0-2): 2
Enter the column (0-2): 2
O | X | X
X | O | O
X | O | X
It's a draw!



Code:#vacuum cleaner

#Enter LOCATION A/B in capital letters

#Enter Status O/1 accordingly where 0 means CLEAN and 1 means DIRTY

```
def vacuum_world():
    # initializing goal_state
    # 0 indicates Clean and 1 indicates Dirty
  goal_state = {'A': '0', 'B': '0'}
  cost = 0

  location_input = input("Enter Location of Vacuum ") #user_input of location vacuum is placed
  status_input = input("Enter status of  "+ location_input+" ") #user_input if location is dirty or clean
  status_input_complement = input("Enter status of other room ")
  print("Initial Location Condition " + str(goal_state))

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

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

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

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

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

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

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

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

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

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


vacuum_world()
```

Output:

```
PS D:\VScodestuff\AI_Lab> python -u "d:\VScodestuff\AI_Lab\vaccum_world.py"
Enter Location of Vacuum A
Enter status of  A 0
Enter status of other room 0
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
PS D:\VScodestuff\AI_Lab>
```

```
PS D:\VScodestuff\AI_Lab> python -u "d:\VScodestuff\AI_Lab\vaccum_world.py"
Enter Location of Vacuum A
Enter status of  A 1
Enter status of other room 0
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
PS D:\VScodestuff\AI_Lab>
```

```
PS D:\VScodestuff\AI_Lab> python -u "d:\VScodestuff\AI_Lab\vaccum_world.py"
Enter Location of Vacuum B
Enter status of  B 1
Enter status of other room 0
Initial Location Condition {'A': '0', 'B': '0'}
Vacuum is placed in location B
Location B is Dirty.
COST for CLEANING 1
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 1
PS D:\VScodestuff\AI_Lab>
```

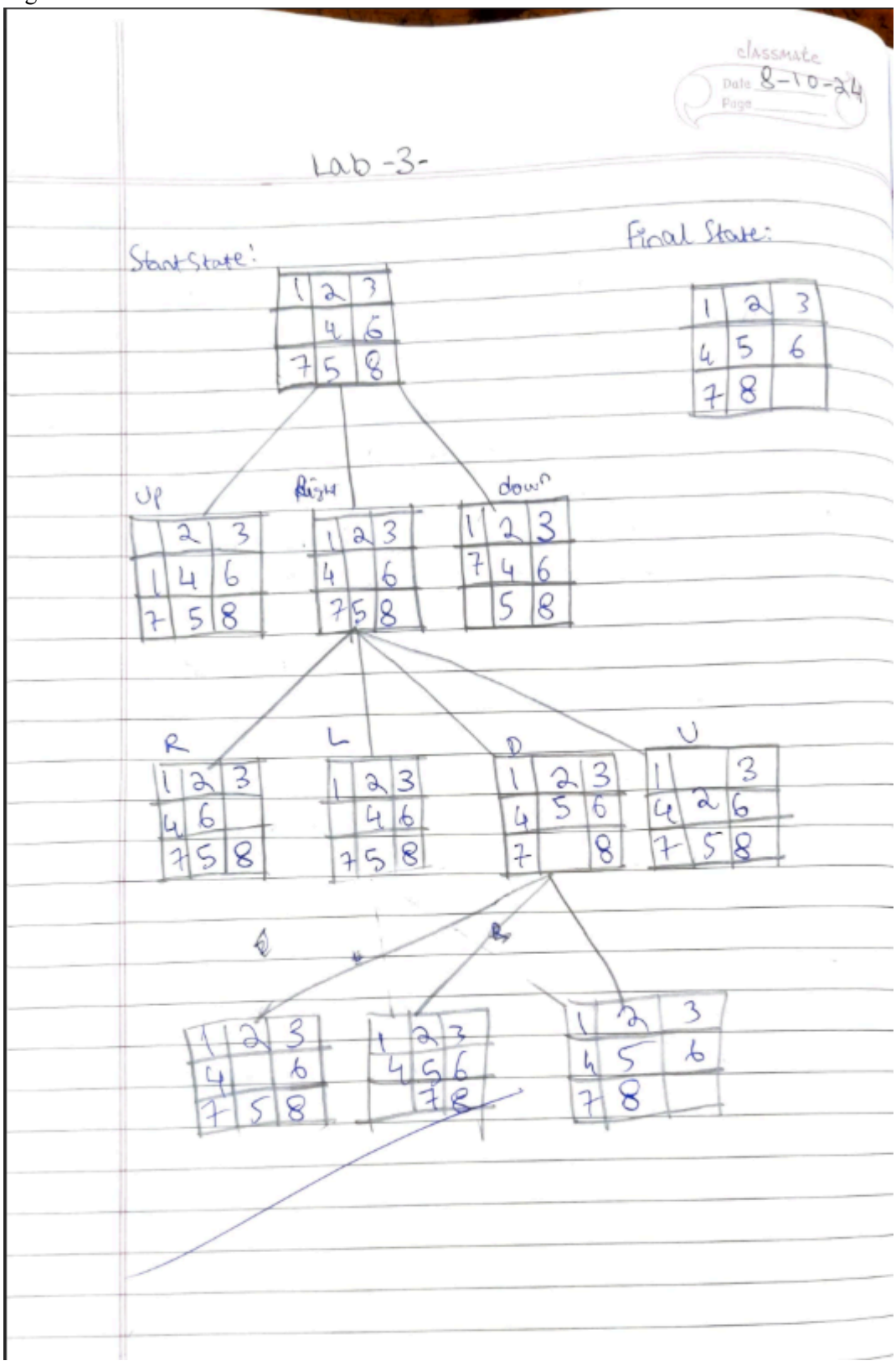
```
PS D:\VScodestuff\AI_Lab> python -u "d:\VScodestuff\AI_Lab\vaccum_world.py"
Enter Location of Vacuum A
Enter status of  A 1
Enter status of other room 1
Initial Location Condition {'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A 1
Location A has been Cleaned.
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
PS D:\VScodestuff\AI_Lab>
```

(You should paste here your code for the respective program)

(You should repeat as above, for all the remaining programs from 2 to 10)

Program 2:

Algorithm:



BFS algorithm:

(1) First we initialise an empty queue and make a BFS function with an empty queue and explored array

(2) While queue is not empty, we append source to explored and print state

(3) Find out possible moves by checking index of empty state. If it is in corners then only up / down available

(4) If direction is possible append to all possible moves

(5) Make a function to perform movements using swapping of values at i^{th} and $(i+3)^{rd}$ index

(6) Keep continuing this process till queue is out of moves to reach goal state

(7) Goal state is checked by comparing both lists

Lab - Iterative deepening DFS

5) Iterative deepening to solve N Queens

Algorithm:

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

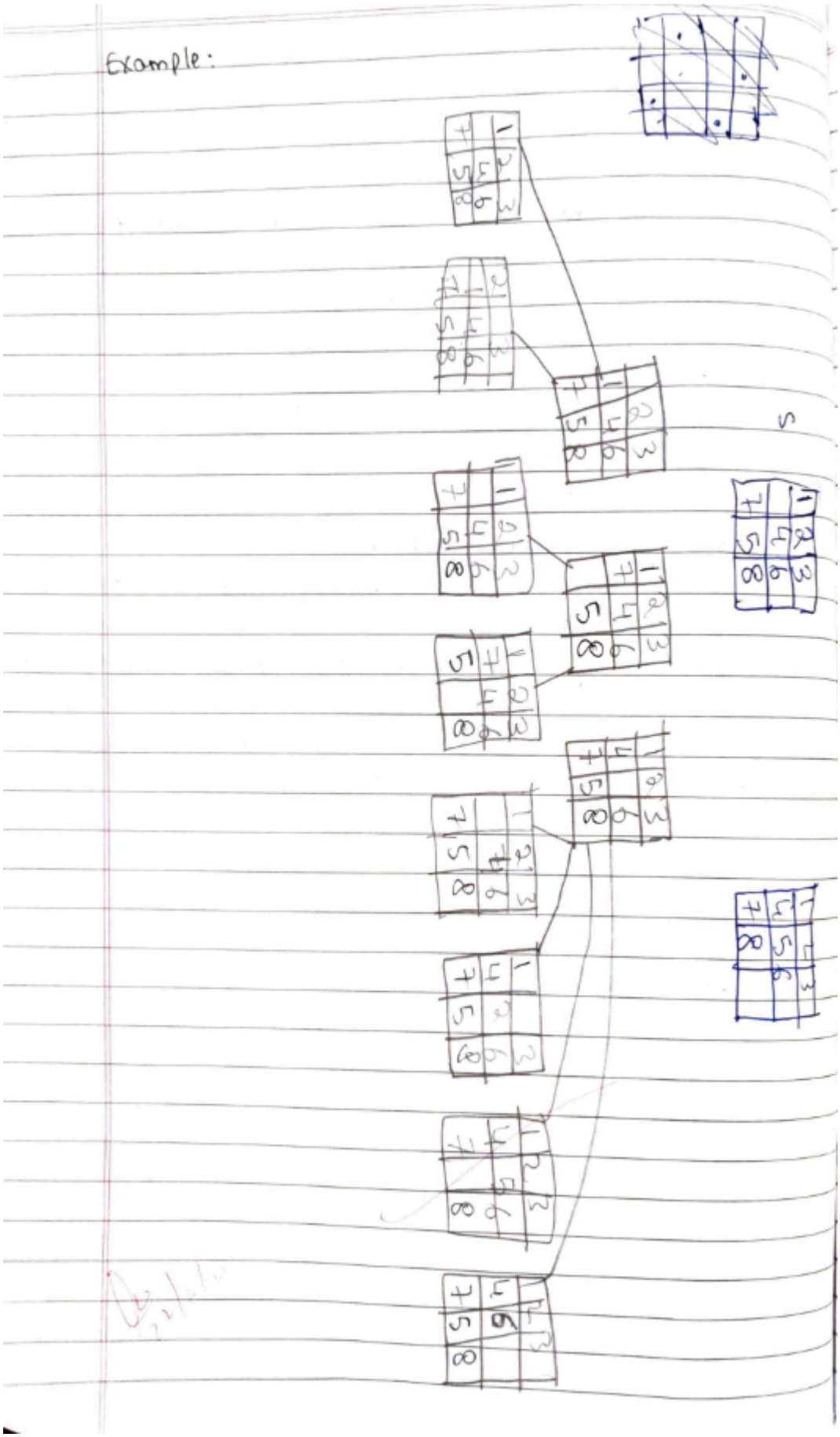
for depth = 0 to ∞ do
result ← DEPTH-LIMITED-SEARCH (problem, depth)
if result ≠ cutoff then return result

1. For each child of the current node
2. If it is the target node return
3. If the current maximum depth is reached. return
4. Set the current node to this node and go back to 1
5. After having gone through all children, go to the next child of the parent (the next sibling)

6. After having gone through all children of the start node, increase the maximum depth and go back to 1

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

Example:

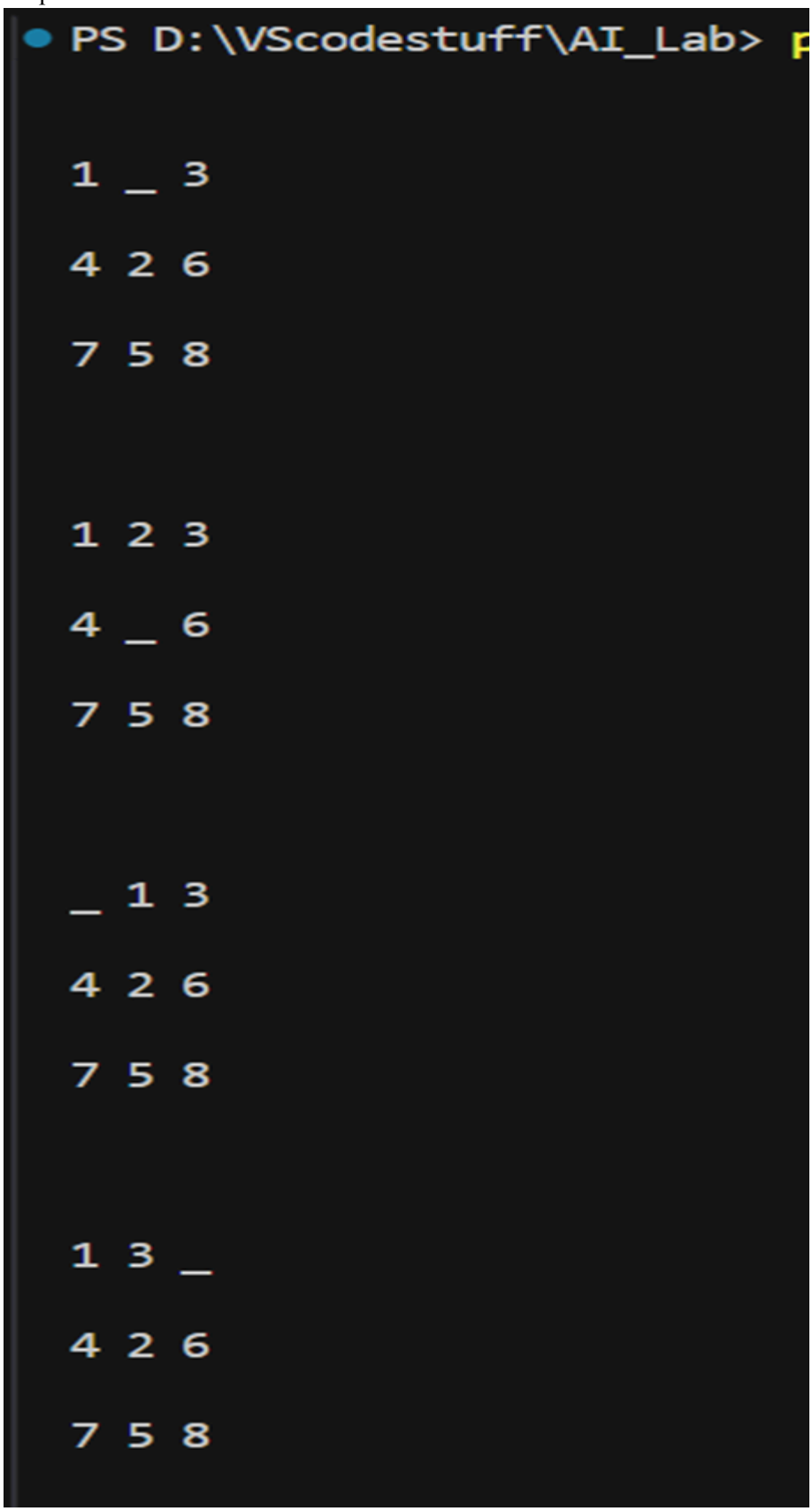


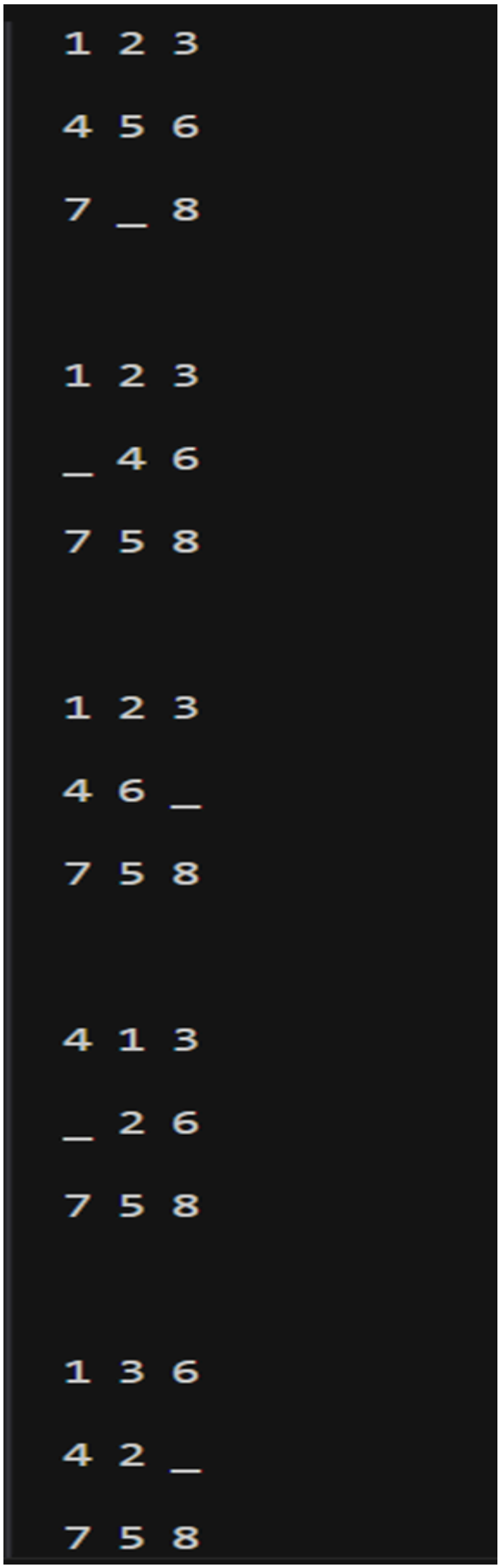
Code: #DFS

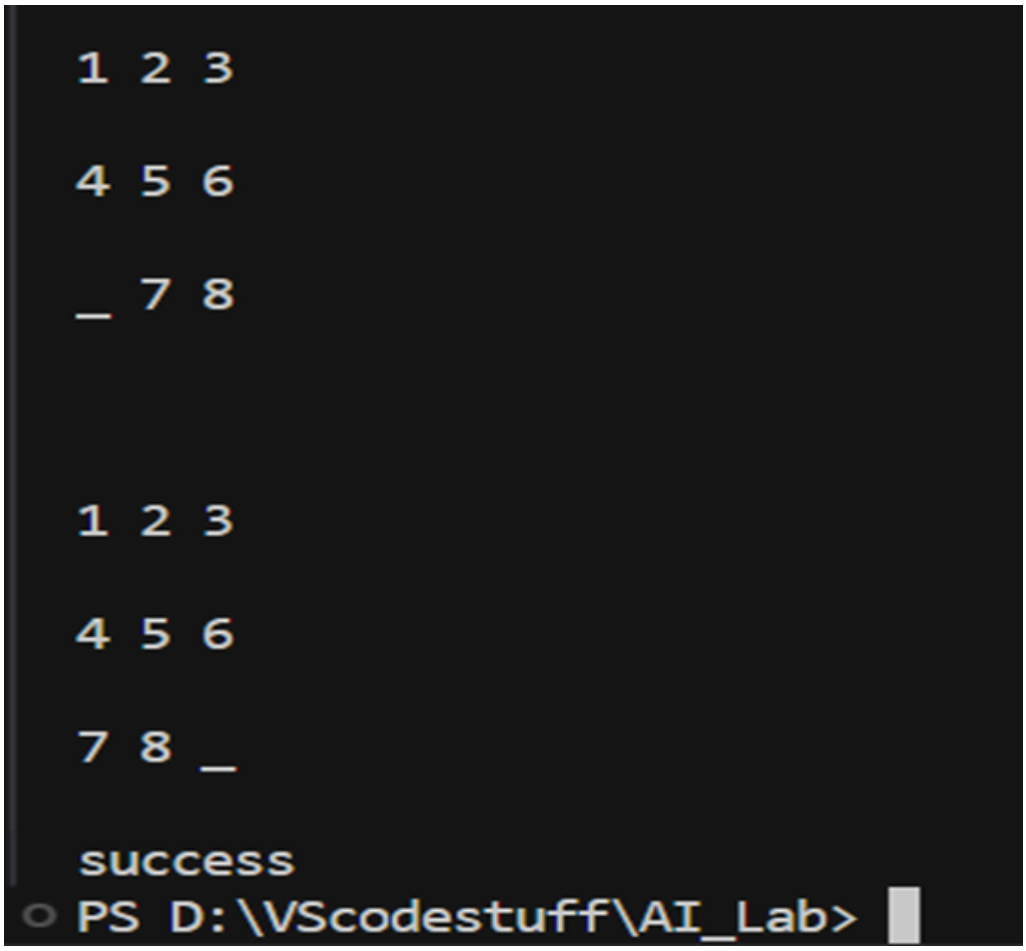
```
def bfs(src,target):
    queue = []
    queue.append(src)

    exp = []

    while len(queue) > 0:
        source = queue.pop(0)
        exp.append(source)

        print_state(source)

        if source==target:
            print("success")
            return

        poss_moves_to_do = []
        poss_moves_to_do = possible_moves(source,exp)

        for move in poss_moves_to_do:

            if move not in exp and move not in queue:
                queue.append(move)

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):
    #index of empty spot
    b = state.index(0)

    #directions array
    d = []
    #Add all the possible directions

    if b not in [0,1,2]:
```

```
        d.append('u')
    if b not in [6,7,8]:
        d.append('d')
    if b not in [0,3,6]:
        d.append('l')
    if b not in [2,5,8]:
        d.append('r')


    # If direction is possible then add state to move
    pos_moves_it_can = []

    # for all possible directions find the state if that move is played
    ### Jump to gen function to generate all possible moves in the given directions

    for i in d:
        pos_moves_it_can.append(gen(state,i,b))

    return [move_it_can for move_it_can in pos_moves_it_can if move_it_can not in visited_states]

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

    if m=='d':
        temp[b+3],temp[b] = temp[b],temp[b+3]

    if m=='u':
        temp[b-3],temp[b] = temp[b],temp[b-3]

    if m=='l':
        temp[b-1],temp[b] = temp[b],temp[b-1]

    if m=='r':
        temp[b+1],temp[b] = temp[b],temp[b+1]

    # return new state with tested move to later check if "src == target"
    return temp

src = [1,0,3,4,2,6,7,5,8]
target = [1,2,3,4,5,6,7,8,0]
bfs(src, target)
```

Output:


PS D:\VScodestuff\AI_Lab>
1 _ 3
4 2 6
7 5 8
1 2 3
4 _ 6
7 5 8
_ 1 3
4 2 6
7 5 8
1 3 _
4 2 6
7 5 8


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



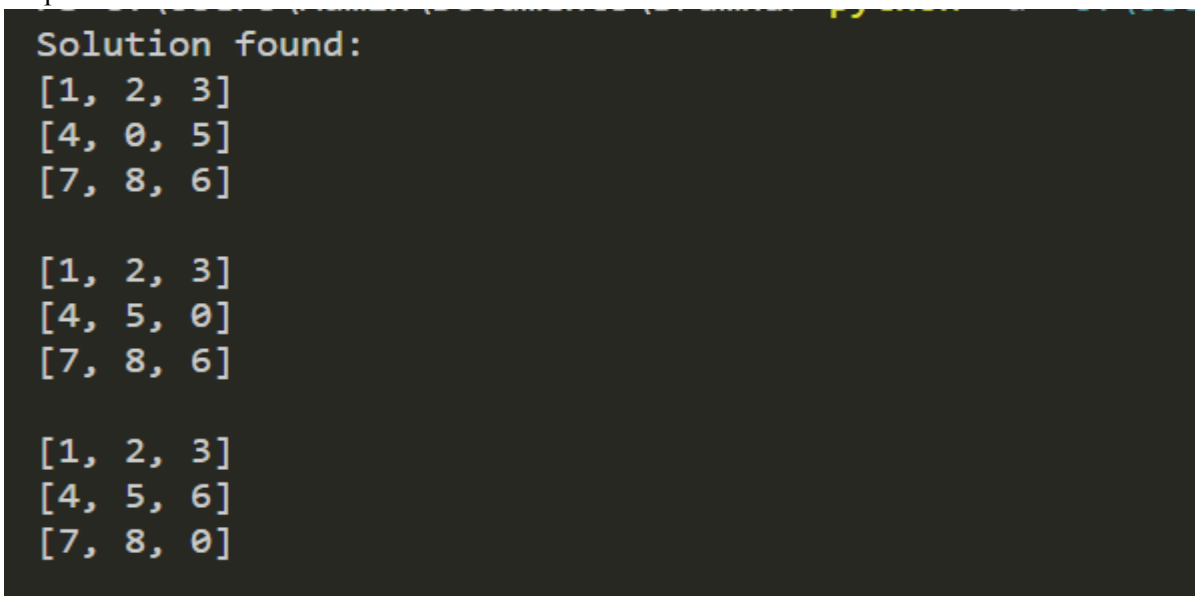
Code: #Iterative deepening

```
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 = 3




# 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:



Program 3:

Algorithm:

Date 19-10-24
Page

a) 8 puzzle algorithm using A* algorithm using ~~A*~~

~~B~~ ~~f(n)~~, a) g(n) - depth of a node

~~b~~ h(n) - heuristic value

↓

No. of misplaced tiles

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

b) g(n) = depth

h(n) = heuristic value

↓

Manhattan distance

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

a) Draw the state space diagram for

2	8	3
1	6	4
7		5

Initial

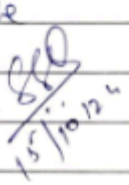
1	2	3
8		4
7	6	5

goal


A*
classmate
Date
Page
g=0
h=4
Uncreated
L
U
R
g(n)=1
h(n)=5
f(n)=6
h(n)=3
f(n)=4
h(n)=5
f(n)=6
g(n)=2
f(n)=3
f(n)=5
h(n)=4
f(n)=6
h(n)=3
h(n)=4
h(n)=2
h(n)=4
g=4
f=5
g=5
f=5
h=0

Algorithm: A*

1. We create an empty list called OPEN which will consist of ~~all possible~~ the starting node
2. If the OPEN list is empty we return false and stop
3. We check the node from the OPEN list which has the smallest value [f(n)]. If it matches goal state then we return success
4. We ~~expand~~ the node n ~~to~~ for all of its successors and we place n in closed list. We make sure that all new expansions dont exist in OPEN or CLOSED
5. We use a 1D array for representation ~~1 node~~



Code: #A*

```
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:

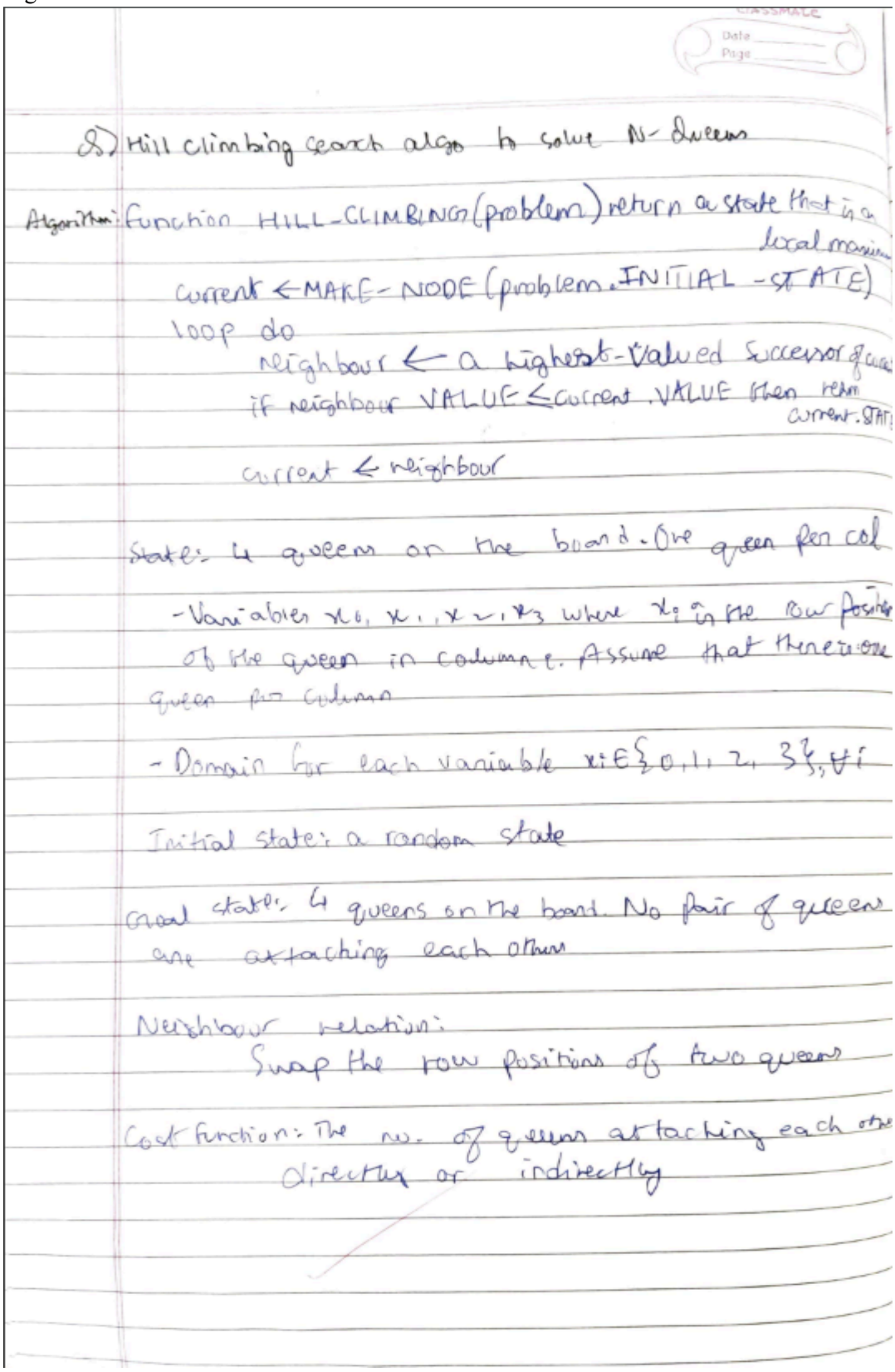
```
PS D:\VScodestuff\AI_Lab> python -u "d:\VScodestuff\AI_Lab\astar.py"
--------------------------------


2 8 3

1 6 4

7 _ 5

Cost: 0
Depth: 0
Moves:
--------------------------------


2 8 3

1 _ 4

7 6 5

Cost: 3
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
Path to target: [(2, 8, 3, 1, 6, 4, 7, 0, 5), (2, 8, 3, 1, 0, 4, 7, 6, 5), (2, 0, 3, 1, 8, 4, 7, 6, 5), (0, 2, 3, 1, 8, 4, 7, 6, 5), (1, 2, 3, 0,
8, 4, 7, 6, 5), (1, 2, 3, 8, 0, 4, 7, 6, 5)]
PS D:\VScodestuff\AI_Lab>
```

Program 4:

Algorithm:



8) Hill climbing search algo to solve N-Queens

Algorithm: function HILL-CLIMBING(problem) return 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 col

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

- Domain for each variable $x_i \in \{0, 1, 2, 3\}, \forall i$

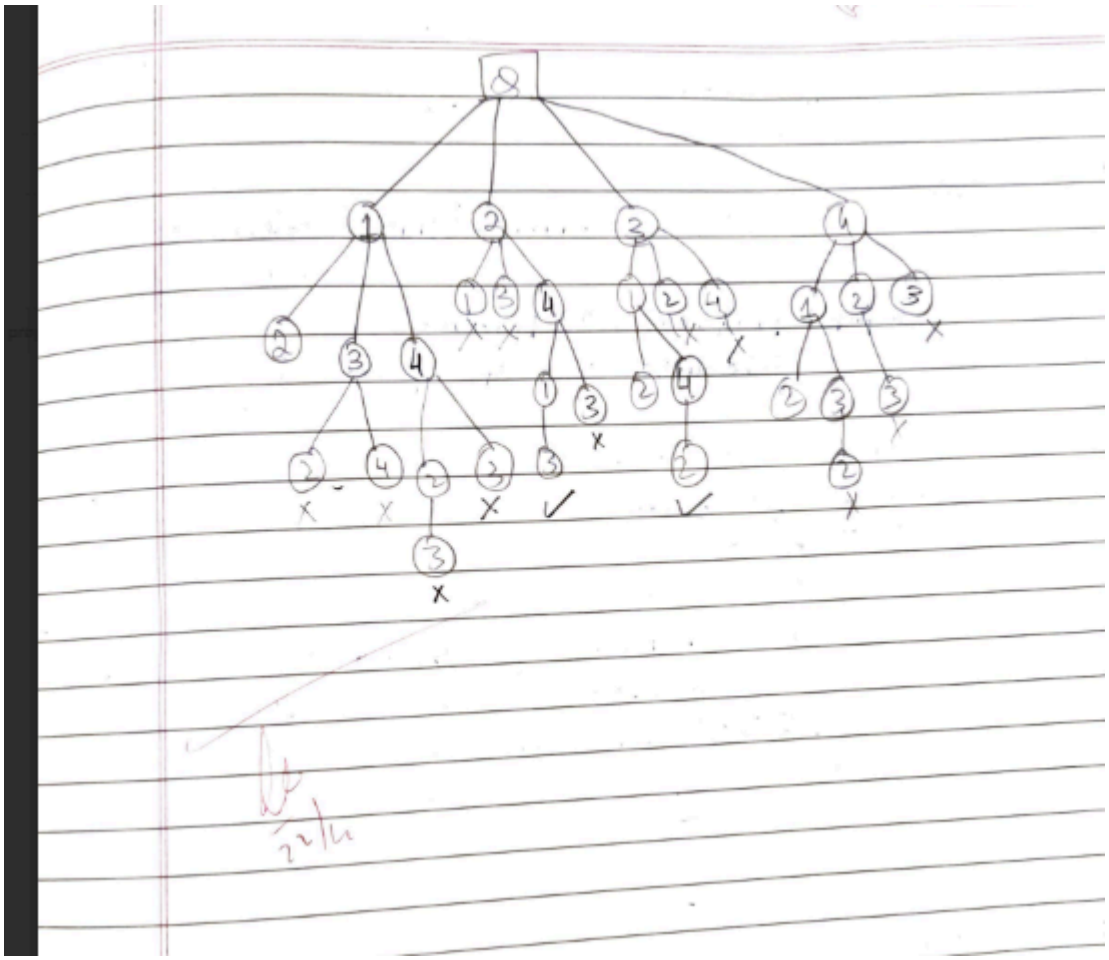
Initial state: a random state

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

Neighbour relation:

Swap the row positions of two queens

Cost function: The no. of queens attacking each other directly or indirectly



Code: #Hill climbing

```
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
```

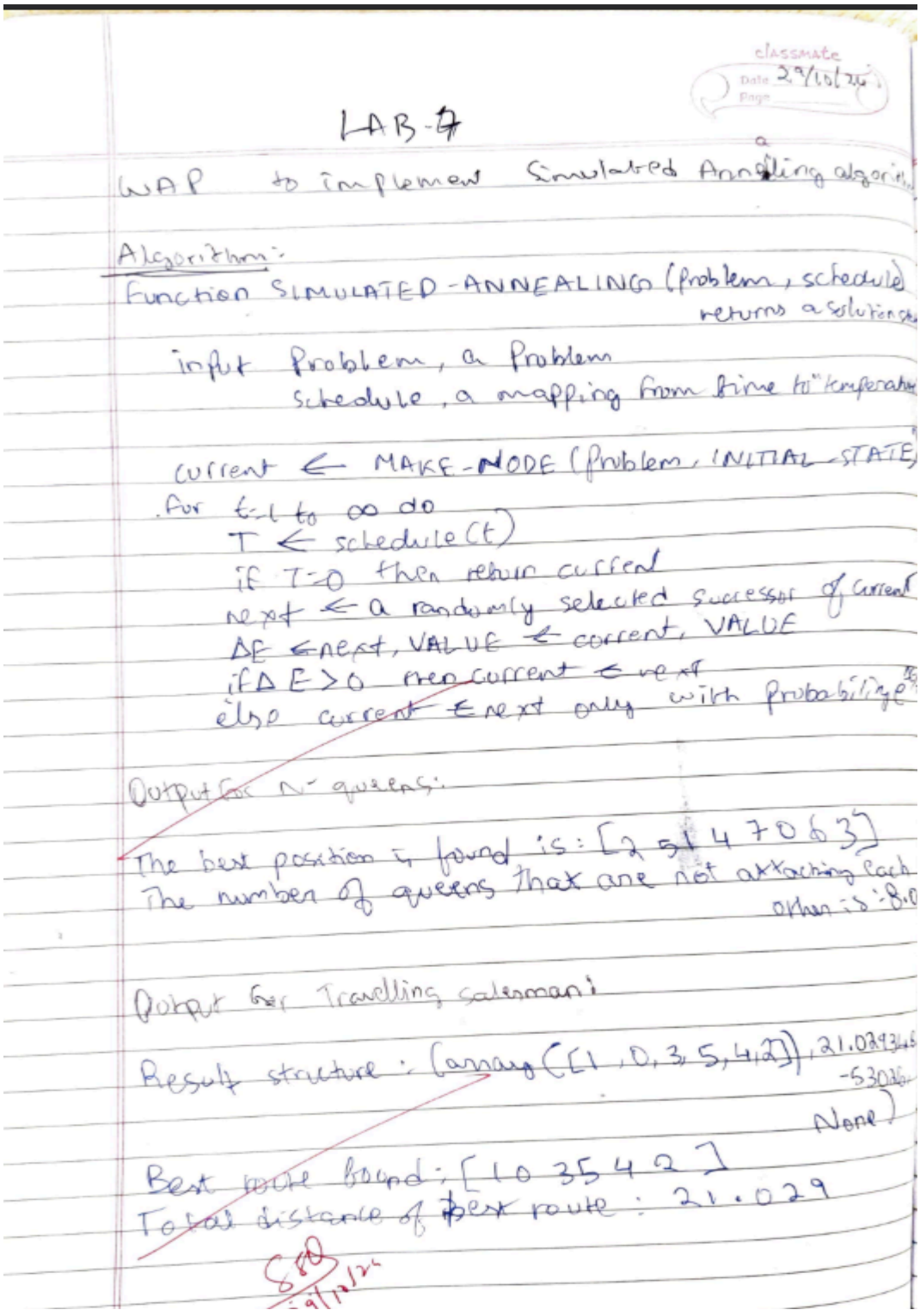
```
initial_state = PuzzleState(initial_board, (1, 1))  # (1, 1) is the position of the empty tile
goal_state = goal_board


# 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:

```
PS C:\Users\Admin\Documents\Bramha> python -u "c:\Users\Ad
Enter the number of queens (N): 4
Iteration 0: Current state: [0, 2, 3, 1], Cost: 1
Local maximum reached at iteration 1. Restarting...
Iteration 0: Current state: [0, 3, 1, 1], Cost: 2
Iteration 1: Current state: [0, 3, 1, 2], Cost: 1
Local maximum reached at iteration 2. Restarting...
Iteration 0: Current state: [0, 0, 3, 1], Cost: 1
Iteration 1: Current state: [2, 0, 3, 1], Cost: 0
Solution found: [2, 0, 3, 1]
```

Program 5:

Algorithm:



LAB-7

WAP to implement Simulated Annealing algorithm

Algorithm:

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

input 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

Output for N-queens:

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

Output for Travelling salesman:

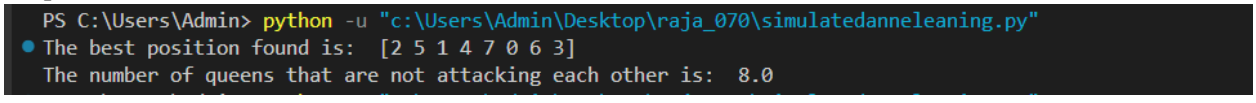
Result structure: (array([1, 0, 3, 5, 4, 2]), 21.029345, -53036. None)

Best route found: [1 0 3 5 4 2]
Total distance of best route: 21.029

Code: #Simulated annealing

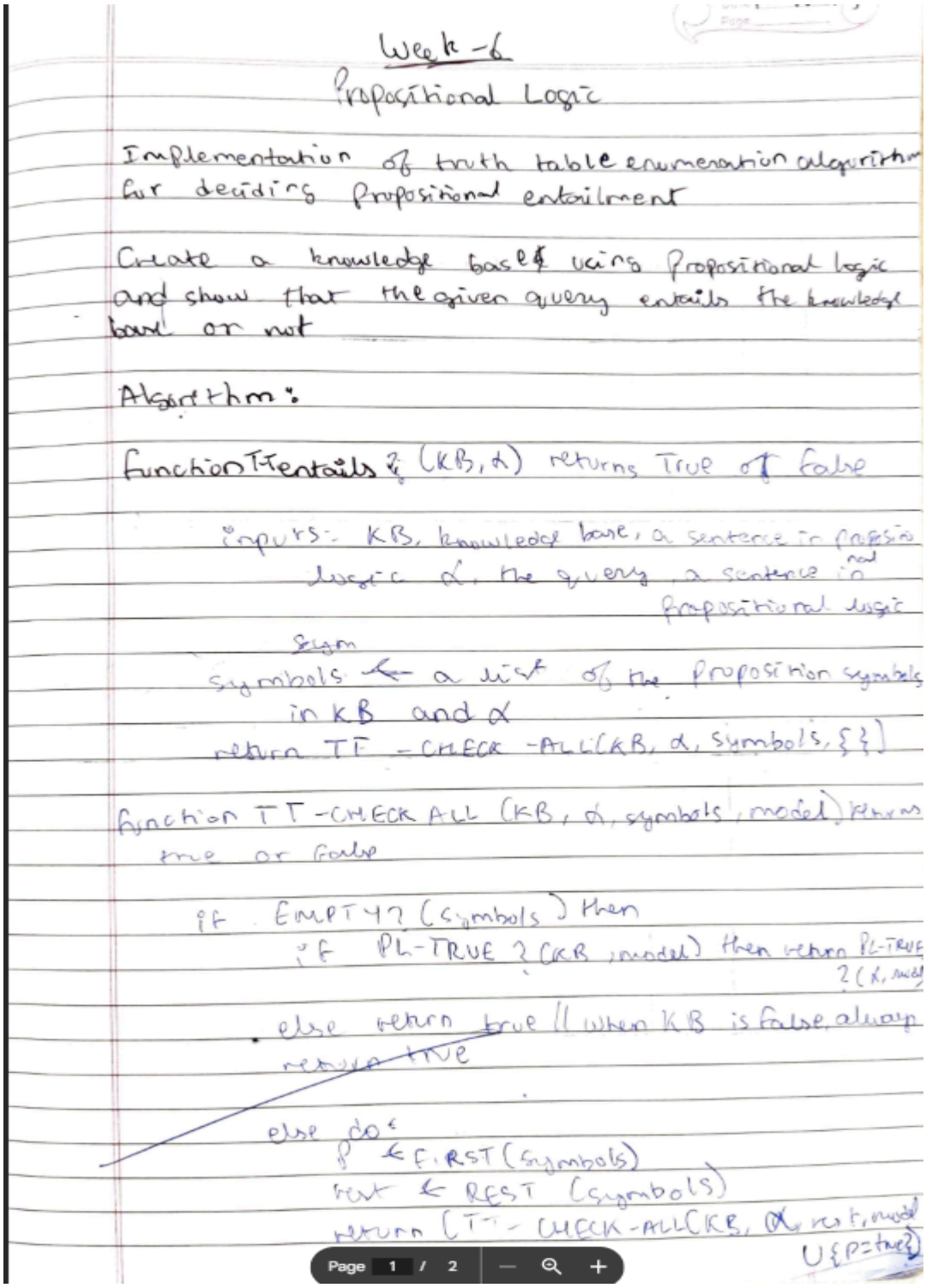
```
import mlrose_hiive as mlrose
import numpy as np


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:



Program 6:

Algorithm:



Week-6

Propositional Logic

Implementation of truth table enumeration algorithm for deciding propositional entailment

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

Algorithm:

function TT-entails? (KB, α) returns True or false

inputs: KB, 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, {})

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

if EMPTY?(symbols) then
if PL-TRUE?(KB, model) then return PL-TRUE?(α, model)
else return true // when KB is false, always return true
else do
P ← FIRST(symbols)
rest ← REST(symbols)
return (TT-CHECK-ALL(KB, α, rest, model ∪ {P=true})

Page 1 / 2

Truth table for connectives:

P	Q	¬P	P∧Q	P∨Q	P⇔Q
false	false	true	false	false	true
false	true	true	false	true	false
true	false	false	false	true	false
true	true	false	true	true	true

Example:

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

Checking $\alpha \models KB$

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

$KB \models \alpha$

12/11/24

Code:#Prepositional logic

```
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)
```

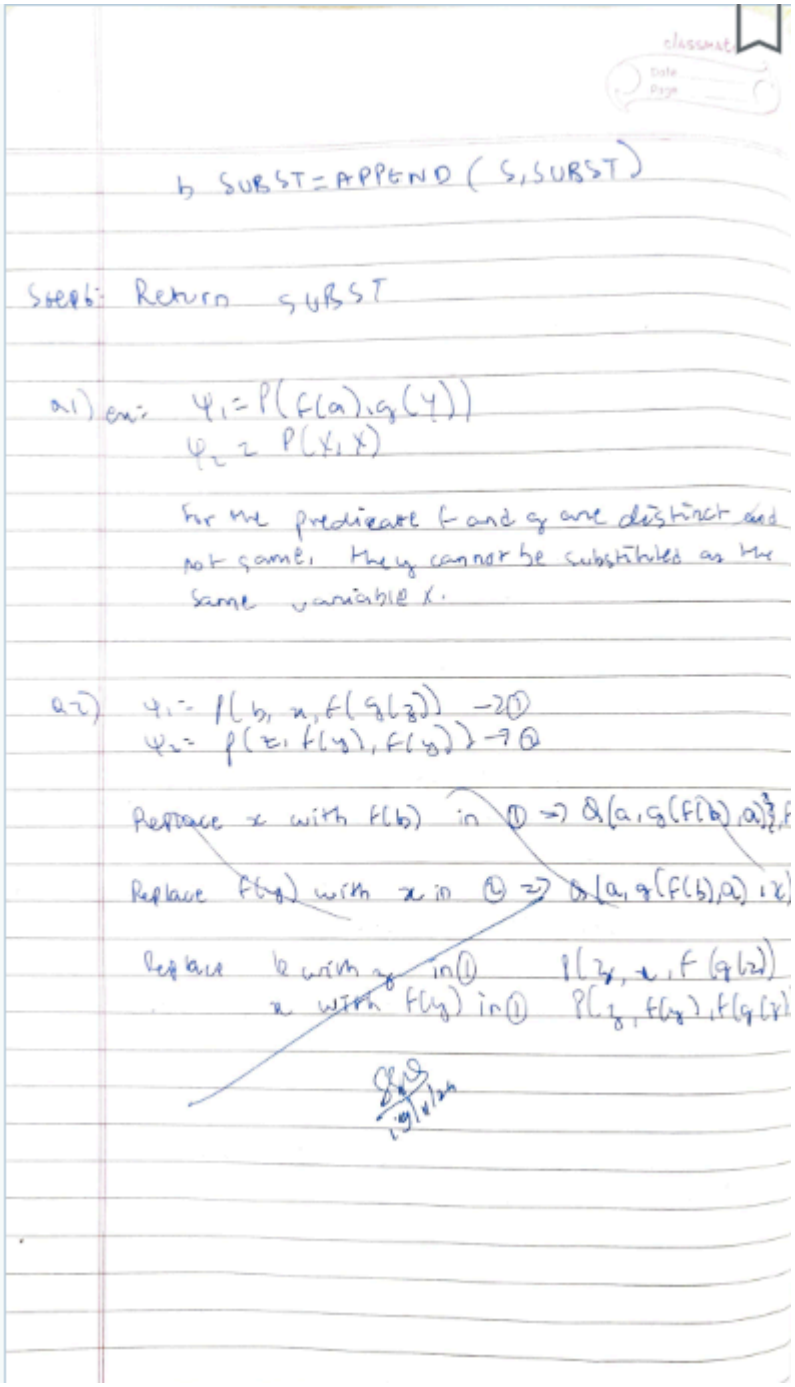
```
        # Highlight rows where both KB and α are true
        if kb_value and alpha_value:
            print(f"\033[92m{row_str}\033[0m")  # Green color for rows where KB and α are true
        else:
            print(row_str)


# 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:

Algorithm:



b SUBST = APPEND (S, SUBST)

Step6: Return SUBST

a1) ex: $\psi_1 = P(f(a), g(y))$
$\psi_2 = P(x, x)$

For the predicate f and g are distinct and not same. They cannot be substituted as the same variable x.

a2) $\psi_1 = P(b, x, f(g(z)))$ →①
$\psi_2 = P(z, f(y), f(y))$ →②

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

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

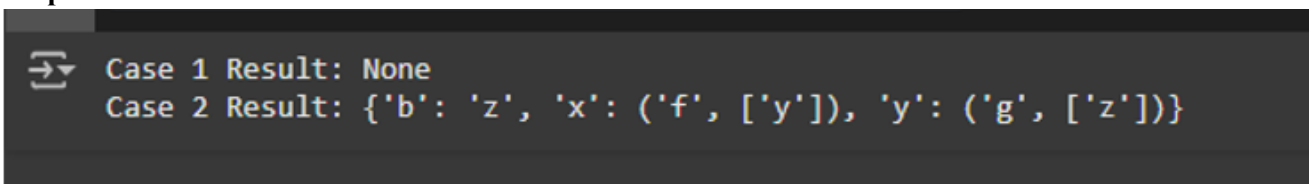
Replace b with y in ① $P(y, x, f(g(z)))$
x with f(y) in ① $P(y, f(y), f(g(z)))$

Code:#Unification First Order logic

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

Output:



Program 8:

Algorithm:

Lab - 8

Forward Reasoning Algorithm:

function FOL_FC_ASK (KB, α) returns a substitution or false

inputs : KB, the knowledge base, a set first order definite clauses.

α, the query, an atomic sentence

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

repeat until new is empty

new ← { }

for each rule in KB do

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

← STANDARDIZE_VARIABLES (rule)

for each θ such that

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

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

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

$q_1' \leftarrow$ SUBST (θ, q)

if q_1' does not unify with some sentence already in KB or new then

add q' to new

$\phi \leftarrow$ UNIFY (q_1', α)

if φ is not fail then

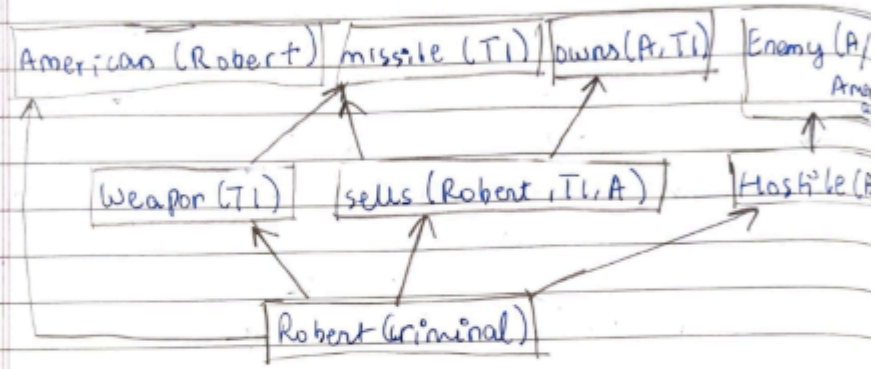
return θ

add new to KB

return false

Example: As per law, it is a crime for an Americ
to sell weapons to hostile nations. Country A, a
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) ∧ selis (p, q, r) ∧ Hostil

∃ x Owns (A, x) ∧ Missiles(x)
Owns (A, T1)
missiles(T1)
∀ x Missile(x) ∧ Owns(A, x) ⇒ sells (Robert, x, A
missile(x) ⇒ weapon (x)
∀ x Enemy (x, American) ⇒ Hostile(x)

26/11/24

Code:#Forward reasoning

```
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
```

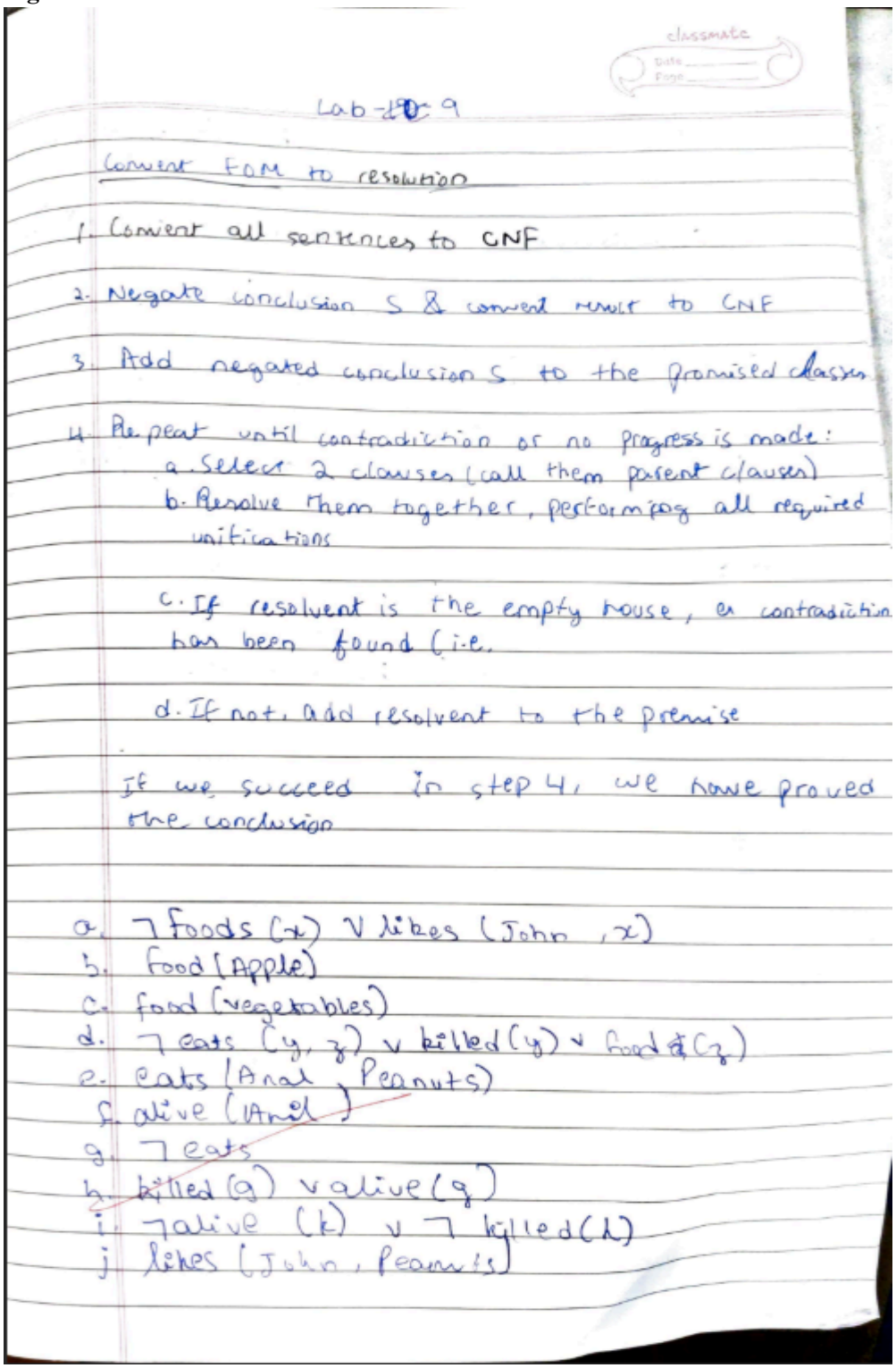
```
# Helper to match a single fact
def match_fact(condition, fact, substitutions):
    if condition["type"] != fact["type"]:
        return False
    for key, value in condition.items():
        if key == "type":
            continue
        if isinstance(value, str) and value.startswith("?"):  # Variable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

Algorithm:



Lab - 9

Convert FOL to resolution

1. Convert all sentences to CNF
2. Negate conclusion S & convert result to CNF
3. Add negated conclusion S to the premised clauses
4. Repeat until contradiction or no progress is made:
   a. Select 2 clauses (call them parent clauses)
   b. Resolve them together, performing all required unifications
   c. If resolvent is the empty clause, a contradiction has been found (i.e.
   d. If not, add resolvent to the premise

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

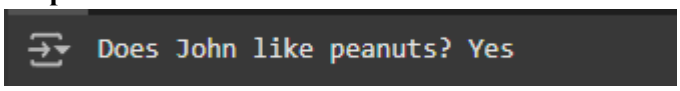
a. ¬ foods (x) V likes (John, x)
b. food (Apple)
c. food (vegetables)
d. ¬ eats (y, z) v killed (y) v food (z)
e. eats (Anil, Peanuts)
f. alive (Anil)
g. ¬ eats
h. killed (g) v alive (g)
i. ¬ alive (k) v ¬ killed (k)
j. likes (John, Peanuts)

Code:#FOM to resolution

```
# 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)")
```

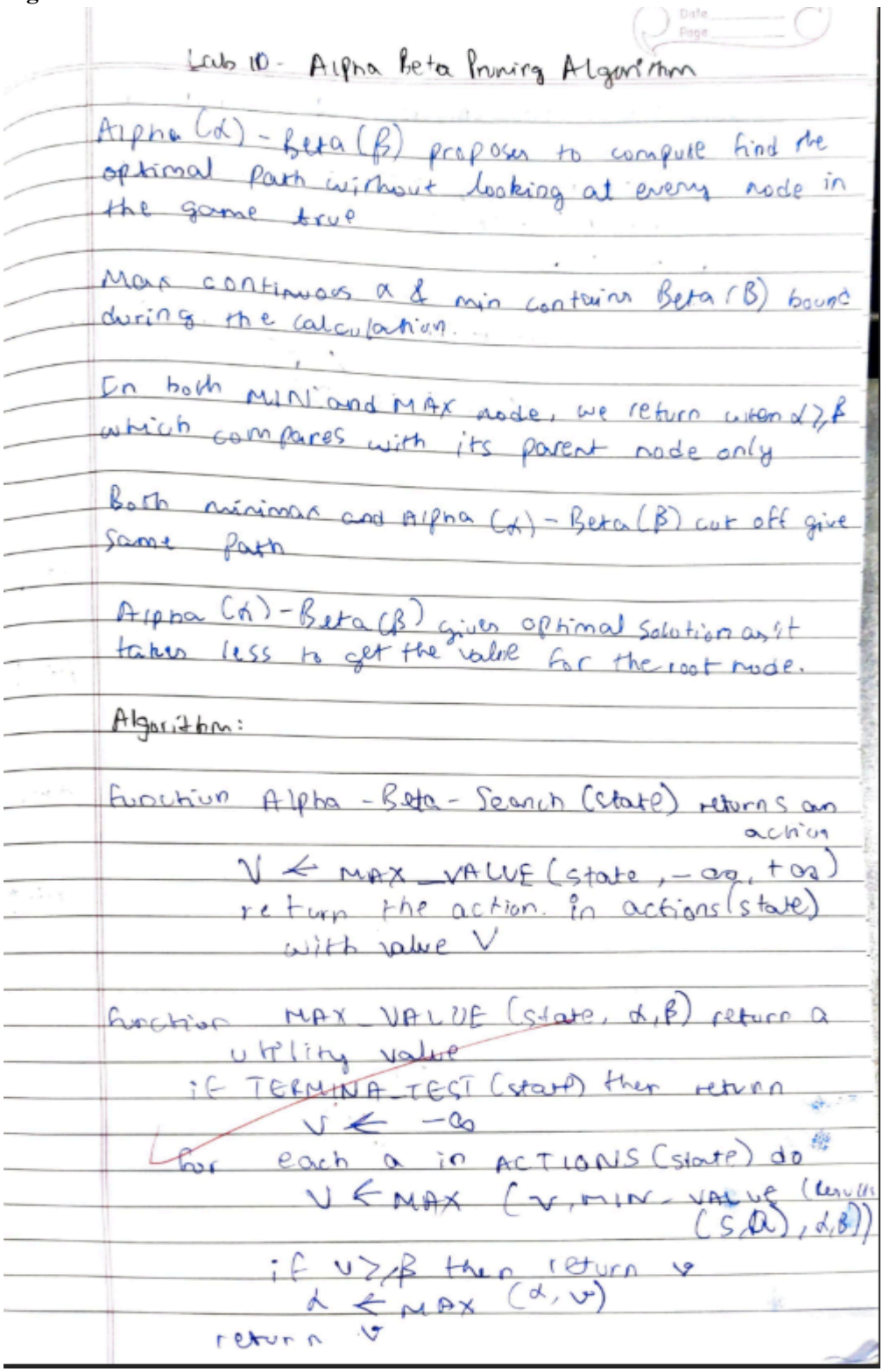
```
    # Default to False if no rule or fact applies
    return False

# 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:



Program 10:

Algorithm:



Lab 10 - 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 continuous α & min contains Beta (β) bound during the calculation.

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

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

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

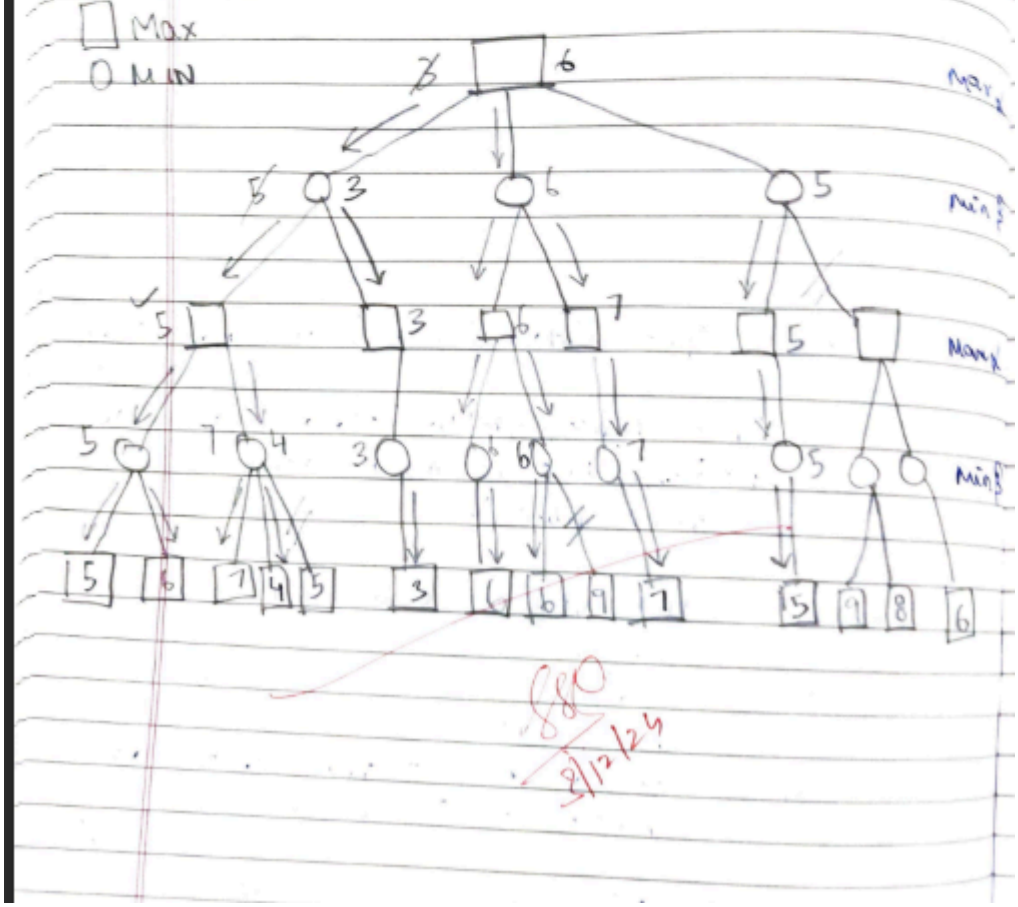
Algorithm:

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) return a utility value
if TERMINAL_TEST (state) then return
V ← −∞
for each a in ACTIONS (state) do
V ← MAX (V, MIN_VALUE (RESULT (s, a), α, β))
if V ≥ β then return V
α ← MAX (α, V)
return V

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



Code:#Minmax algo

```
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: