Artificial Intelligence Lab Report

Submitted by

SWAPNIL SAHIL(1BM22CS300)
Batch: 1

Course: Artificial Intelligence
Course Code: 23CS5PCAIP
Sem & Section: 5F

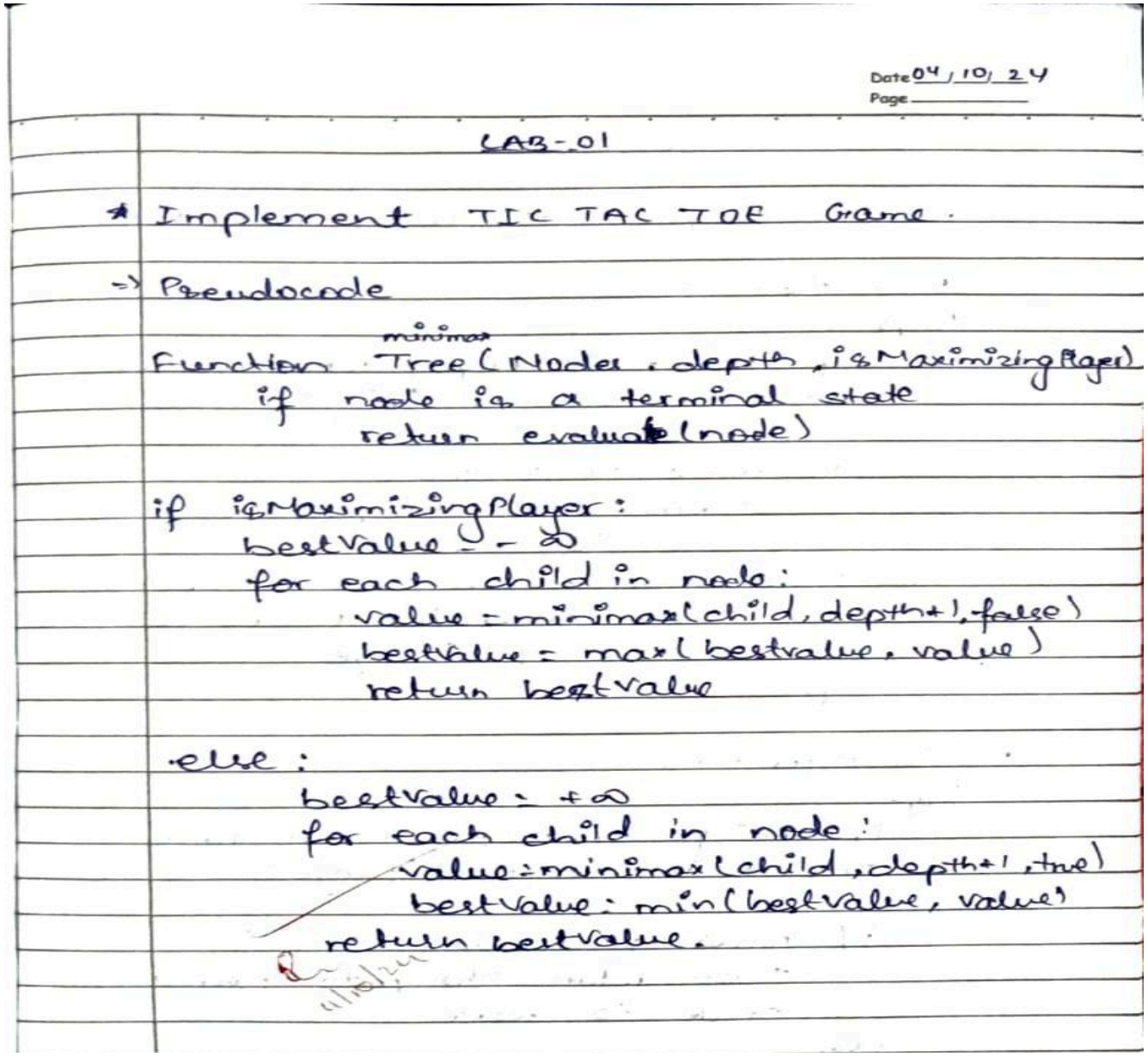
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B. M. S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
2023-2024

Table of contents

Program 1 - Tic Tac toe

Algorithm



Date 04/10/24
Page

LAB-01

* Implement TIC TAC TOE Game.

=> Pseudocode

```
Function minimax Tree(Node, depth, isMaximizingPlayer)
    if node is a terminal state
        return evaluate(node)

    if isMaximizingPlayer:
        bestValue = -∞
        for each child in node:
            value = minimax(child, depth+1, false)
            bestValue = max(bestvalue, value)
            return bestValue

    else:
        bestValue = +∞
        for each child in node:
            value = minimax(child, depth+1, true)
            bestValue = min(bestValue, value)
        return bestValue.
```

Code :

```
board={1:' ',2:' ',3:' ',
    4:' ',5:' ',6:' ',
    7:' ',8:' ',9:' '
}
```

```
def printBoard(board):
    print(board[1]+'|'+board[2]+'|'+board[3])
    print('-+-+-')
    print(board[4] + '|' + board[5] + '|' + board[6])
    print('-+-+-')
    print(board[7] + '|' + board[8] + '|' + board[9])
    print('\n')

def spaceFree(pos):
    if(board[pos]==' '):
        return True
    else:
        return False

def checkWin():
    if(board[1]==board[2] and board[1]==board[3] and board[1]!=' '):
        return True
    elif(board[4]==board[5] and board[4]==board[6] and board[4]!=' '):
        return True
    elif(board[7]==board[8] and board[7]==board[9] and board[7]!=' '):
        return True
    elif (board[1] == board[5] and board[1] == board[9] and board[1] != ' '):
        return True
    elif (board[3] == board[5] and board[3] == board[7] and board[3] != ' '):
        return True
    elif (board[1] == board[4] and board[1] == board[7] and board[1] != ' '):
        return True
    elif (board[2] == board[5] and board[2] == board[8] and board[2] != ' '):
        return True
    elif (board[3] == board[6] and board[3] == board[9] and board[3] != ' '):
        return True
    else:
        return False

def checkMoveForWin(move):
    if (board[1]==board[2] and board[1]==board[3] and board[1] ==move):
        return True
    elif (board[4]==board[5] and board[4]==board[6] and board[4] ==move):
        return True
    elif (board[7]==board[8] and board[7]==board[9] and board[7] ==move):
        return True
    elif (board[1]==board[5] and board[1]==board[9] and board[1] ==move):
        return True
    elif (board[3]==board[5] and board[3]==board[7] and board[3] ==move):
        return True
```

```
  elif (board[1]==board[4] and board[1]==board[7] and board[1] ==move):
    return True
  elif (board[2]==board[5] and board[2]==board[8] and board[2] ==move):
    return True
  elif (board[3]==board[6] and board[3]==board[9] and board[3] ==move):
    return True
  else:
    return False

def checkDraw():
  for key in board.keys():
    if (board[key]==' '):
      return False
  return True
def insertLetter(letter, position):
  if (spaceFree(position)):
    board[position] = letter
    printBoard(board)

    if (checkDraw()):
      print('Draw!')
    elif (checkWin()):
      if (letter == 'X'):
        print('Bot wins!')
      else:
        print('You win!')
    return

  else:
    print('Position taken, please pick a different position.')
    position = int(input('Enter new position: '))
    insertLetter(letter, position)
    return

player = 'O'
bot ='X'

def playerMove():
  position=int(input('Enter position for O:'))
  insertLetter(player, position)
  return

def compMove():
  bestScore=-1000
  bestMove=0
```

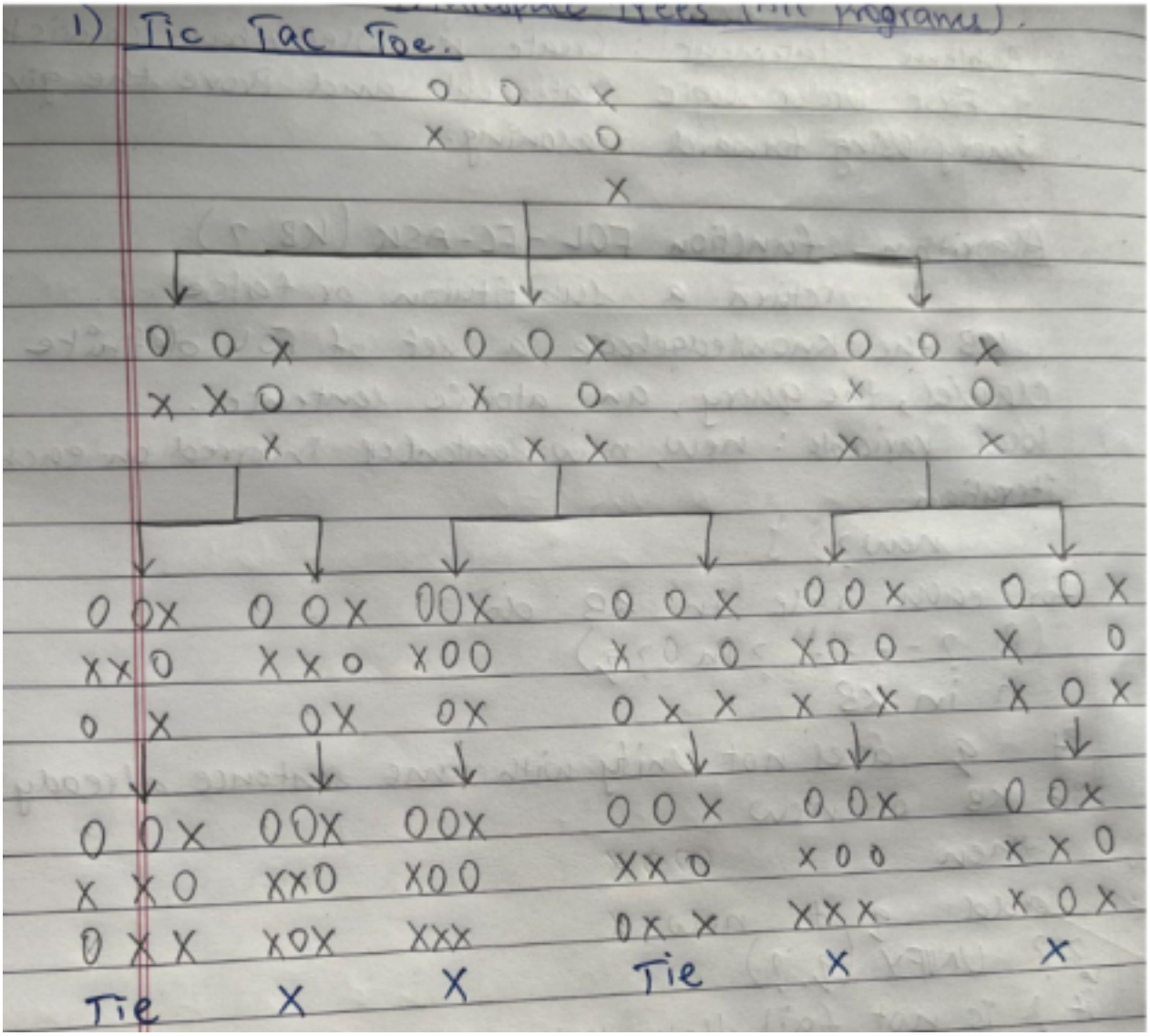
```
    for key in board.keys():
        if (board[key]==' '):
            board[key]=bot
            score = minimax(board, False)
            board[key] = ' '
            if (score > bestScore):
                bestScore = score
                bestMove = key

    insertLetter(bot, bestMove)
    return
def minimax(board, isMaximizing):
    if (checkMoveForWin(bot)):
        return 1
    elif (checkMoveForWin(player)):
        return -1
    elif (checkDraw()):
        return 0

    if isMaximizing:
        bestScore = -1000
        for key in board.keys():
            if board[key] == ' ':
                board[key] = bot
                score = minimax(board, False)
                board[key] = ' '
                if (score > bestScore):
                    bestScore = score
        return bestScore
    else:
        bestScore = 1000

        for key in board.keys():
            if board[key] == ' ':
                board[key] = player
                score = minimax(board, True)
                board[key] = ' '
                if (score < bestScore):
                    bestScore = score
        return bestScore

while not checkWin():
    compMove()
    playerMove()
```

Output :

```
OUTPUT:
SWAPNIL SAHIL (1BM22CS300)
X| |
-+-+-
 | |
-+-+-
 | |


Enter position for O:5
X| |
-+-+-
 |O|
-+-+-
 | |


X|X|
-+-+-
 |O|
-+-+-
 | |


Enter position for O:3
X|X|O
-+-+-
 |O|
-+-+-
 | |


X|X|O
-+-+-
 |O|
-+-+-
X| |


Enter position for O:4
X|X|O
-+-+-
O|O|
-+-+-
X| |


X|X|O
-+-+-
O|O|X
-+-+-
X| |
```

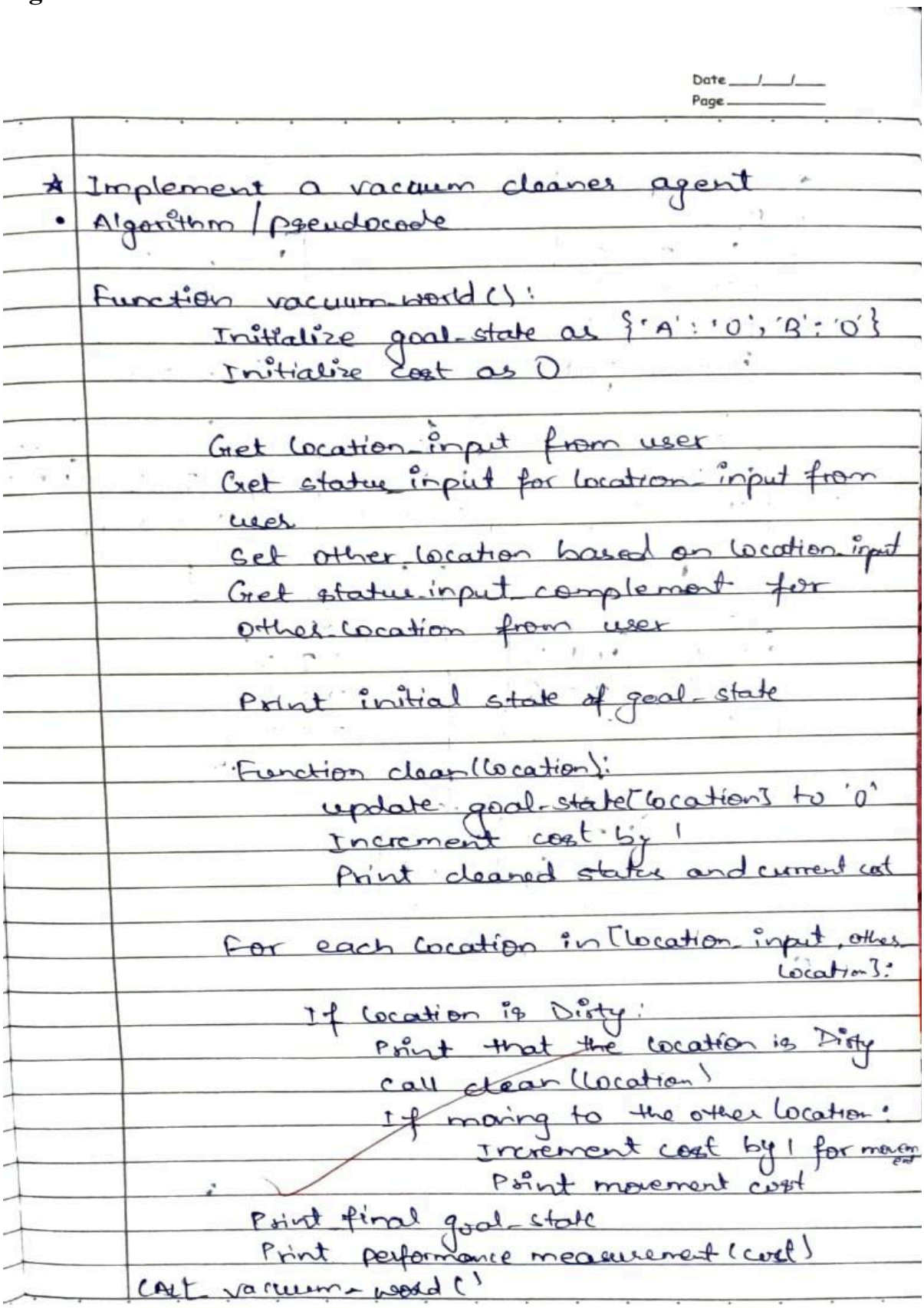
```
Enter new position: 8
X|X|O
-+-+-
O|O|X
-+-+-
X|O|


X|X|O
-+-+-
O|O|X
-+-+-
X|O|X


Draw!
Draw!
```

State Space Tree



Program 2:Vacuum Cleaner

Algorithm



Date __/__/__
Page _____

★ Implement a vacuum cleaner agent
• Algorithm / pseudocode

Function vacuum_world():
    Initialize goal_state as {'A': '0', 'B': '0'}
    Initialize cost as 0

    Get location_input from user
    Get status_input for location_input from user
    Set other_location based on location_input
    Get status_input_complement for other_location from user

    Print initial state of goal_state

    Function clean(location):
        update goal_state[location] to '0'
        Increment cost by 1
        Print cleaned status and current cost

    For each location in [location_input, other_location]:

        If location is Dirty:
            Print that the location is Dirty
            call clean(location)
            If moving to the other location:
                Increment cost by 1 for movement
                Print movement cost

    Print final goal_state
    Print performance measurement (cost)
Call vacuum_world()

Code:

```
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 (A or B): ").strip().upper()  # User input for vacuum location
    status_input = input(f"Enter status of {location_input} (0 for Clean, 1 for Dirty): ").strip()  # Status of the current location
    other_location = 'B' if location_input == 'A' else 'A'
    status_input_complement = input(f"Enter status of {other_location} (0 for Clean, 1 for Dirty): ").strip()  # Status of the other room

    print("Initial Location Condition: " + str(goal_state))

    # Helper function to clean a location
    def clean(location):
        nonlocal cost
        goal_state[location] = '0'
        cost += 1  # Cost for sucking dirt
        print(f"Location {location} has been Cleaned. Cost: {cost}")

    # Main logic
    if location_input == 'A':
        print("Vacuum is placed in Location A.")
        if status_input == '1':
            print("Location A is Dirty.")
            clean('A')
            if status_input_complement == '1':
                print("Location B is Dirty.")
                print("Moving right to Location B.")
                cost += 1  # Cost for moving right
                print(f"COST for moving RIGHT: {cost}")
```

```
            clean('B')
        else:
            print("Location B is already clean.")
    else:
        print("Location A is already clean.")
        if status_input_complement == '1':
            print("Location B is Dirty.")
            print("Moving right to Location B.")
            cost += 1  # Cost for moving right
            print(f"COST for moving RIGHT: {cost}")
            clean('B')
        else:
            print("Location B is already clean.")

else:  # Vacuum is placed in Location B
    print("Vacuum is placed in Location B.")
    if status_input == '1':
        print("Location B is Dirty.")
        clean('B')
        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("Moving left to Location A.")
            cost += 1  # Cost for moving left
            print(f"COST for moving LEFT: {cost}")
            clean('A')
        else:
            print("Location A is already clean.")
    else:
        print("Location B is already clean.")
        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("Moving left to Location A.")
            cost += 1  # Cost for moving left
            print(f"COST for moving LEFT: {cost}")
            clean('A')
```

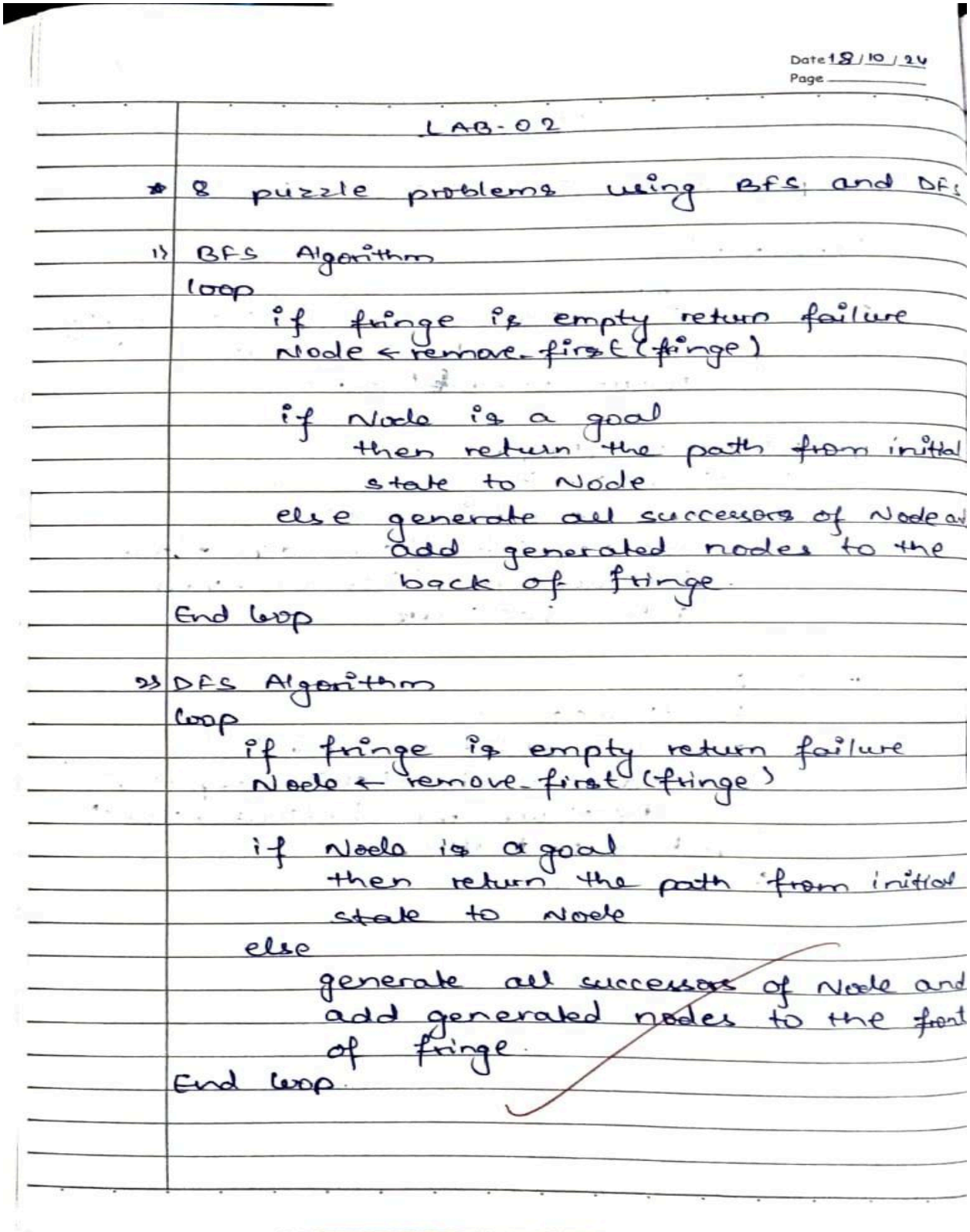
```
        else:
            print("Location A is already clean.")

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

# Output
print('OUTPUT:')
print('SWAPNIL SAHIL (1BM22CS300)')
vacuum_world()
```

OUTPUT:

```
SWAPNIL SAHIL (1BM22CS300)
Enter Location of Vacuum (A or B): A
Enter status of A (0 for Clean, 1 for Dirty): 1
Enter status of B (0 for Clean, 1 for Dirty): 1
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in Location A.
Location A is Dirty.
Location A has been Cleaned. Cost: 1
Location B is Dirty.
Moving right to Location B.
COST for moving RIGHT: 2
Location B has been Cleaned. Cost: 3
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
```

Program 3 - 8 Puzzle Using BFS and DFS

Algorithm



Date 18 / 10 / 24
Page

LAB-02

* 8 puzzle problems using BFS and DFS

1) BFS Algorithm

loop
if fringe is empty return failure
Node ← remove_first(fringe)

if Node is a goal
then return the path from initial state to Node
else generate all successors of Node and add generated nodes to the back of fringe
End loop

2) DFS Algorithm

loop
if fringe is empty return failure
Node ← remove_first(fringe)

if Node is a goal
then return the path from initial state to Node
else
generate all successors of Node and add generated nodes to the front of fringe
End loop

Code(BFS) :

```
# 8 puzzle problem using BFS technique
# prompt: solve 8-puzzle problem using BFS

from collections import deque

def solve_8puzzle_bfs(initial_state):

    def find_blank(state):
        """Finds the row and column of the blank tile."""
        for row in range(3):
            for col in range(3):
                if state[row][col] == 0:
                    return row, col

    def get_neighbors(state):
        """Generates possible neighbor states by moving the blank tile."""
        row, col = find_blank(state)
        neighbors = []
        if row > 0:
            new_state = [row[:] for row in state]
            new_state[row][col], new_state[row - 1][col] = new_state[row - 1][col],
new_state[row][col]
            neighbors.append(new_state)
        if row < 2:
            new_state = [row[:] for row in state]
            new_state[row][col], new_state[row + 1][col] = new_state[row + 1][col],
new_state[row][col]
            neighbors.append(new_state)
        if col > 0:
            new_state = [row[:] for row in state]
            new_state[row][col], new_state[row][col - 1] = new_state[row][col - 1],
new_state[row][col]
```

```
            neighbors.append(new_state)
        if col < 2:
            new_state = [row[:] for row in state]
            new_state[row][col], new_state[row][col + 1] = new_state[row][col + 1],
new_state[row][col]
            neighbors.append(new_state)
        return neighbors

    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    queue = deque([(initial_state, [])])
    visited = set()

    while queue:
        current_state, path = queue.popleft()
        if current_state == goal_state:
            return path + [current_state]

        visited.add(tuple(map(tuple, current_state)))
        for neighbor in get_neighbors(current_state):
            if tuple(map(tuple, neighbor)) not in visited:
                queue.append((neighbor, path + [current_state]))

    return None  # No solution found
# Example usage:
initial_state = [[1, 2, 3], [4, 0, 6], [7, 5, 8]]
solution = solve_8puzzle_bfs(initial_state)

if solution:
    print("Solution found:")
    for state in solution:
        for row in state:
            print(row)
        print()
else:
    print("No solution found.")
```

Output Snapshot

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

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

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

Code(DFS):

```
from collections import deque


def solve_8puzzle_dfs(initial_state):



    def find_blank(state):
        """Finds the row and column of the blank tile."""
        for row in range(3):
            for col in range(3):
                if state[row][col] == 0:
                    return row, col

    def get_neighbors(state):
        """Generates possible neighbor states by moving the blank tile."""
        row, col = find_blank(state)
        neighbors = []
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_state = [r[:] for r in state]
                new_state[row][col], new_state[new_row][new_col] =
new_state[new_row][new_col], new_state[row][col]
```

```
                neighbors.append(new_state)
        return neighbors

    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    stack = [(initial_state, [])]
    visited = set()

    while stack:
        current_state, path = stack.pop()
        state_tuple = tuple(map(tuple, current_state))  # Convert to tuple for set
        if state_tuple in visited:
            continue
        visited.add(state_tuple)

        if current_state == goal_state:
            return path + [current_state]

        for neighbor in get_neighbors(current_state):
            stack.append((neighbor, path + [current_state]))

    return None  # No solution found

# Example usage:
initial_state = [[1, 2, 3], [4, 5, 6], [0, 7, 8]]
solution = solve_8puzzle_dfs(initial_state)

if solution:
    print("Solution found:")
    for state in solution:
        for row in state:
            print(row)
        print()
else:
    print("No solution found.")
```

OUTPUT:

Solution found:

[1, 2, 3]

[4, 5, 6]

[0, 7, 8]

[1, 2, 3]

[4, 5, 6]

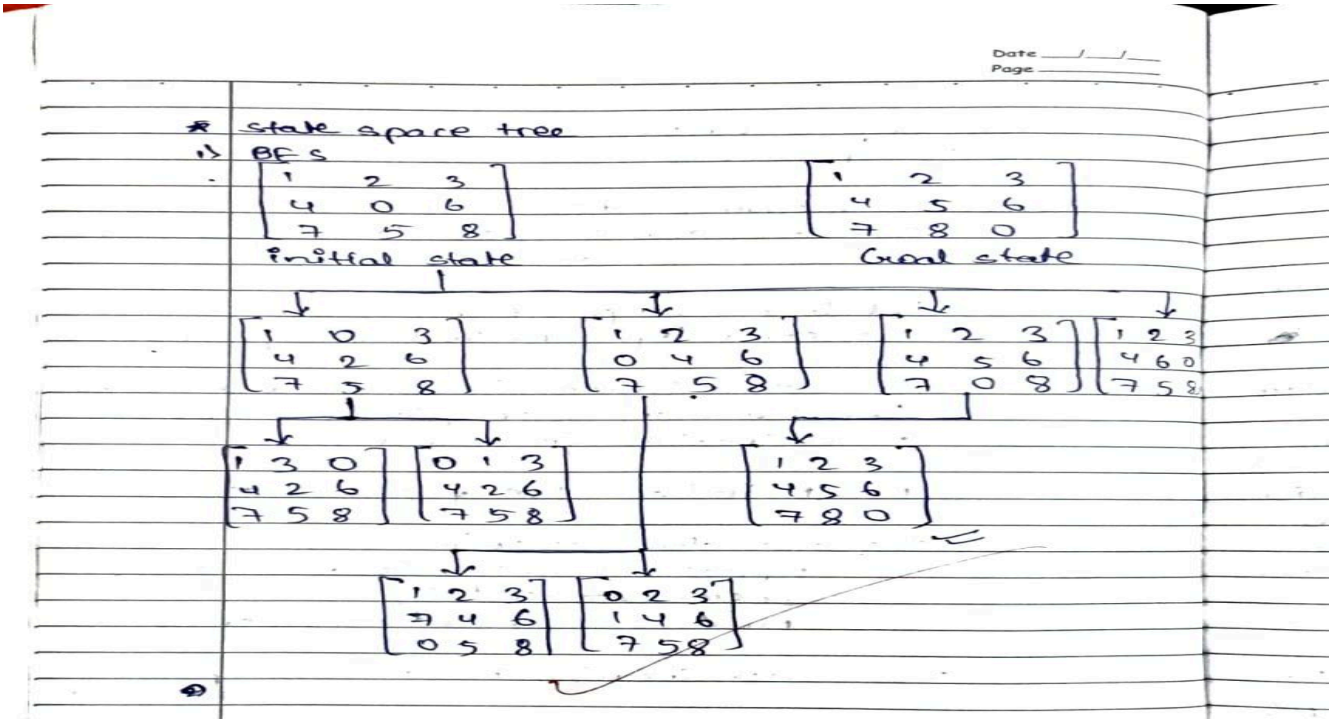
[7, 0, 8]

[1, 2, 3]

[4, 5, 6]

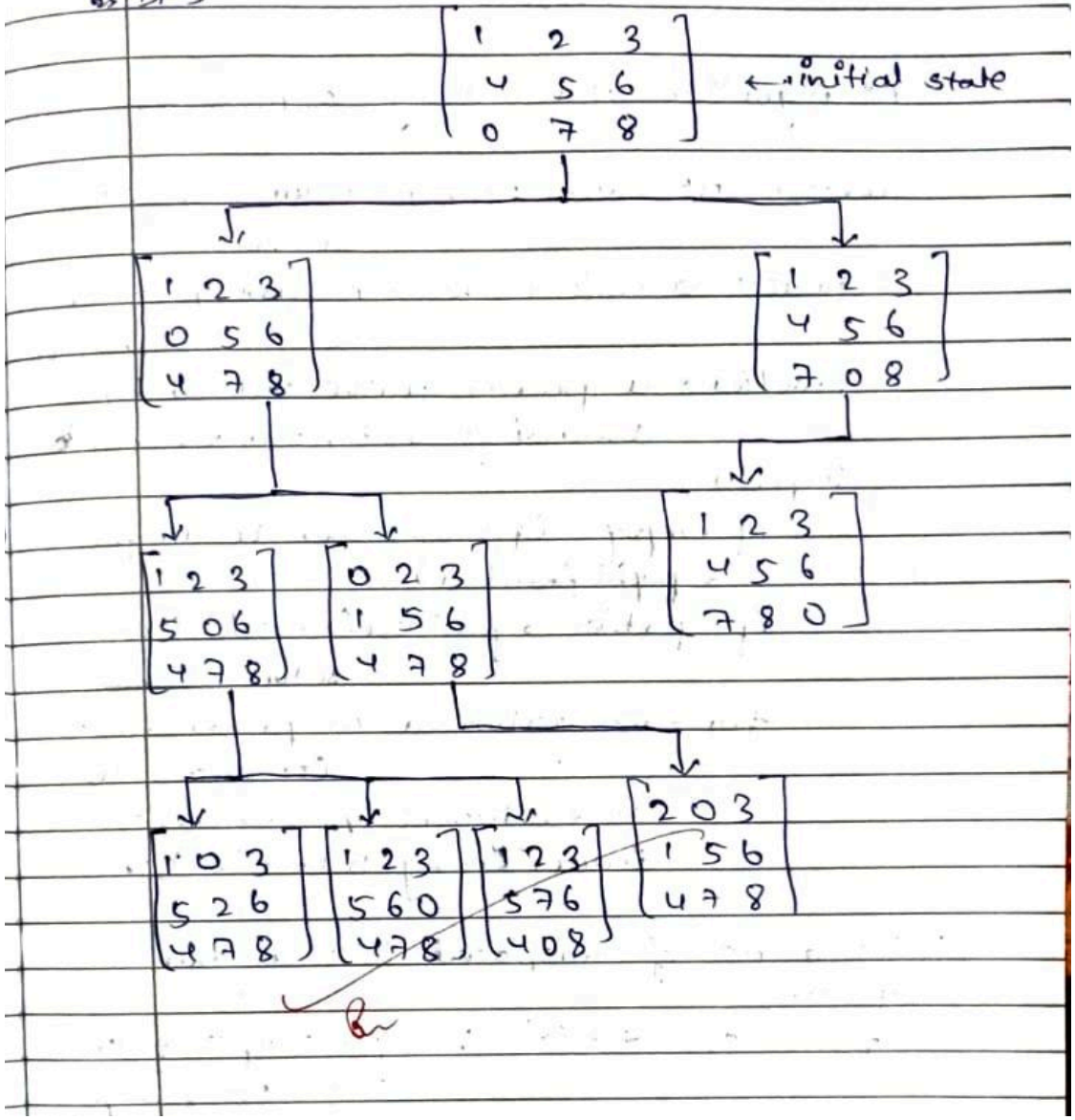
[7, 8, 0]

State Space Tree



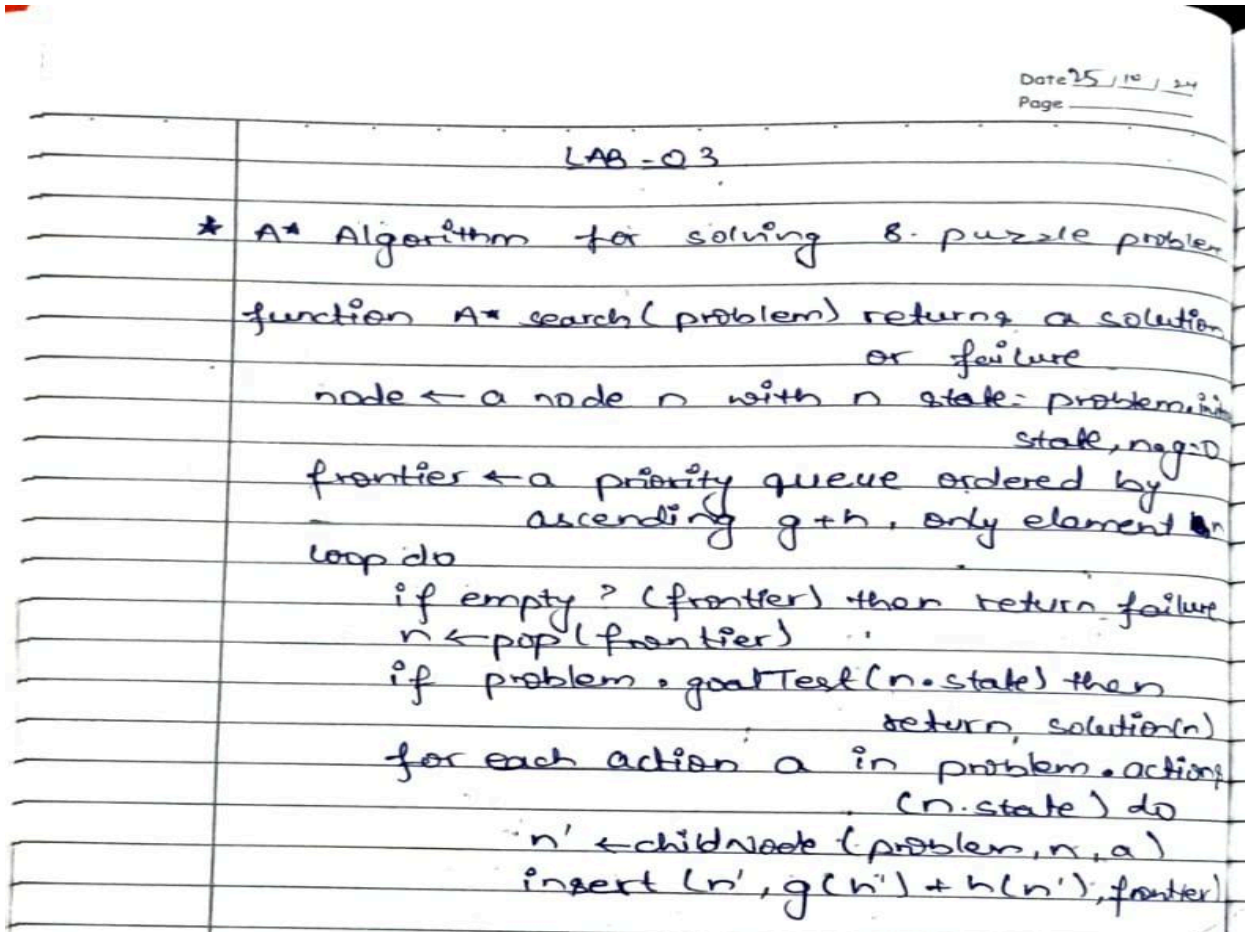
Date __/__/__
Page _____

DFS



Program 4 -A* Algorithm

Algorithm



LAB-03

* A* Algorithm for solving 8-puzzle problem

function A* search(problem) returns a solution or failure
    node ← a node n with n.state = problem.initial state, n.g=0
    frontier ← a priority queue ordered by ascending g+h, only element is n
    loop do
        if empty?(frontier) then return failure
        n ← pop(frontier)
        if problem.goalTest(n.state) then return solution(n)
        for each action a in problem.actions(n.state) do
            n' ← childNode(problem, n, a)
            insert(n', g(n')+h(n'), frontier)

Code:

Manhattan Distance

```
#Manhattan Distance
import heapq

class PuzzleState:
    def __init__(self, board, g=0):
        self.board = board
        self.g = g  # Cost from start to this state
        self.zero_pos = board.index(0)  # Position of the empty space
```

```
    def h(self):
        # Calculate the Manhattan distance
        distance = 0
        for i in range(9):
            if self.board[i] != 0:
                target_x, target_y = divmod(self.board[i] - 1, 3)
                current_x, current_y = divmod(i, 3)
                distance += abs(target_x - current_x) + abs(target_y - current_y)
        return distance

    def f(self):
        return self.g + self.h()

    def get_neighbors(self):
        neighbors = []
        x, y = divmod(self.zero_pos, 3)
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_zero_pos = new_x * 3 + new_y
                new_board = self.board[:]
                # Swap zero with the neighboring tile
                new_board[self.zero_pos], new_board[new_zero_pos] =
new_board[new_zero_pos], new_board[self.zero_pos]
                neighbors.append(PuzzleState(new_board, self.g + 1))
        return neighbors

def a_star(initial_state, goal_state):
    open_set = []
    heapq.heappush(open_set, (initial_state.f(), 0, initial_state))  # Push (f, unique_id, state)
    came_from = {}
    g_score = {tuple(initial_state.board): 0}

    while open_set:
```

```
        current_f, _, current = heapq.heappop(open_set)

        if current.board == goal_state:
            return reconstruct_path(came_from, current)

        for neighbor in current.get_neighbors():
            neighbor_tuple = tuple(neighbor.board)
            tentative_g_score = g_score[tuple(current.board)] + 1

            if neighbor_tuple not in g_score or tentative_g_score < g_score[neighbor_tuple]:
                came_from[neighbor_tuple] = current
                g_score[neighbor_tuple] = tentative_g_score
                # Push (f, unique_id, state)
                heapq.heappush(open_set, (neighbor.f(), neighbor.g, neighbor))  # Using g as a
tie-breaker

    return None  # If no solution is found

def reconstruct_path(came_from, current):
    path = []
    while current is not None:
        path.append(current.board)
        current = came_from.get(tuple(current.board), None)
    return path[::-1]

# Example usage
initial_state = PuzzleState([1, 2, 3, 4, 5, 6, 0, 7, 8])
goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]

solution = a_star(initial_state, goal_state)
print('Name:Swapnil Sahil','USN:1BM22CS300',sep="\n")
if solution:
    for step in solution:
        print(step)
else:
```

```
    print("No solution found")
```

Output:

Name:Swapnil Sahil

USN:1BM22CS300

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

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

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

CODE:

Number of Misplaced tiles

```
import heapq

def misplaced_tiles(state, goal):
    return sum(1 for i in range(len(state)) if state[i] != 0 and state[i] != goal[i])

def get_neighbors(state):
    neighbors = []
    zero_idx = state.index(0)  # Find the empty tile (represented as 0)
    row, col = divmod(zero_idx, 3)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right

    for dr, dc in directions:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_idx = new_row * 3 + new_col
            new_state = state[:]
            # Swap 0 with the neighbor
            new_state[zero_idx], new_state[new_idx] = new_state[new_idx],
new_state[zero_idx]
            neighbors.append(new_state)
    return neighbors

def a_star(initial_state, goal_state):
    # Priority queue for A* (min-heap)
    open_set = []
```

```
    heapq.heappush(open_set, (0, initial_state))  # (priority, state)

    # Dictionaries to store the cost and parent of each state
    g_cost = {tuple(initial_state): 0}  # Cost from start to current state
    parent = {tuple(initial_state): None}  # To reconstruct the path

    while open_set:
        # Get the state with the lowest f(n) = g(n) + h(n)
        _, current = heapq.heappop(open_set)

        # If we reach the goal, reconstruct the path
        if current == goal_state:
            return reconstruct_path(parent, current)

        for neighbor in get_neighbors(current):
            neighbor_tuple = tuple(neighbor)
            tentative_g_cost = g_cost[tuple(current)] + 1

            # If this path is better, update costs and add to open set
            if neighbor_tuple not in g_cost or tentative_g_cost < g_cost[neighbor_tuple]:
                g_cost[neighbor_tuple] = tentative_g_cost
                f_cost = tentative_g_cost + misplaced_tiles(neighbor, goal_state)
                heapq.heappush(open_set, (f_cost, neighbor))
                parent[neighbor_tuple] = current

    return None  # No solution found

# Helper function to reconstruct the path from start to goal
def reconstruct_path(parent, state):
    path = []
    while state is not None:
        path.append(state)
        state = parent[tuple(state)]
    return path[::-1]  # Reverse the path
# Main function
```

```
if __name__ == "__main__":
    # Define the initial state and goal state
    initial_state = [1, 2, 3, 0, 4, 6, 7, 5, 8]  # 0 represents the empty tile
    goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
    solution = a_star(initial_state, goal_state)

    if solution:
        print("Solution found:")
        for step in solution:
            print_board(step)
            print()
    else:
        print("No solution exists.")
# Helper function to print the 8-puzzle board
def print_board(state):
    for i in range(0, 9, 3):
        print(state[i:i + 3])
```

OUTPUT:

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

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

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

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

STATE SPACE:

i) Using no. of misplaced tiles as heuristic function.

$f(n) = g(n) + h(n)$; $g(n)$: depth of tree
$h(n)$: no. of misplaced tiles

→ State space tree

1	2	3
4	5	6
0	7	8

initial state

$f(n) = 0 + 2 = 2$

1	2	3
4	5	6
7	8	0

Goal state

1	2	3
0	5	6
4	7	8

$f(n) = 1 + 3 = 4$

1	2	3
4	5	6
7	0	8

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

1	2	3
4	0	6
7	5	8

$f(n) = 2 + 2 = 4$

1	2	3
4	5	6
7	8	0

$f(n) = 2 + 0 = 2$

Goal state.

ii) Using manhattan distance as heuristic function

$f(n) = g(n) + h(n)$; $g(n)$: depth of tree
$h(n)$: manhattan distance

1	2	3
4	5	6
0	7	8

initial state

1	2	3
4	5	6
7	8	0

Goal state

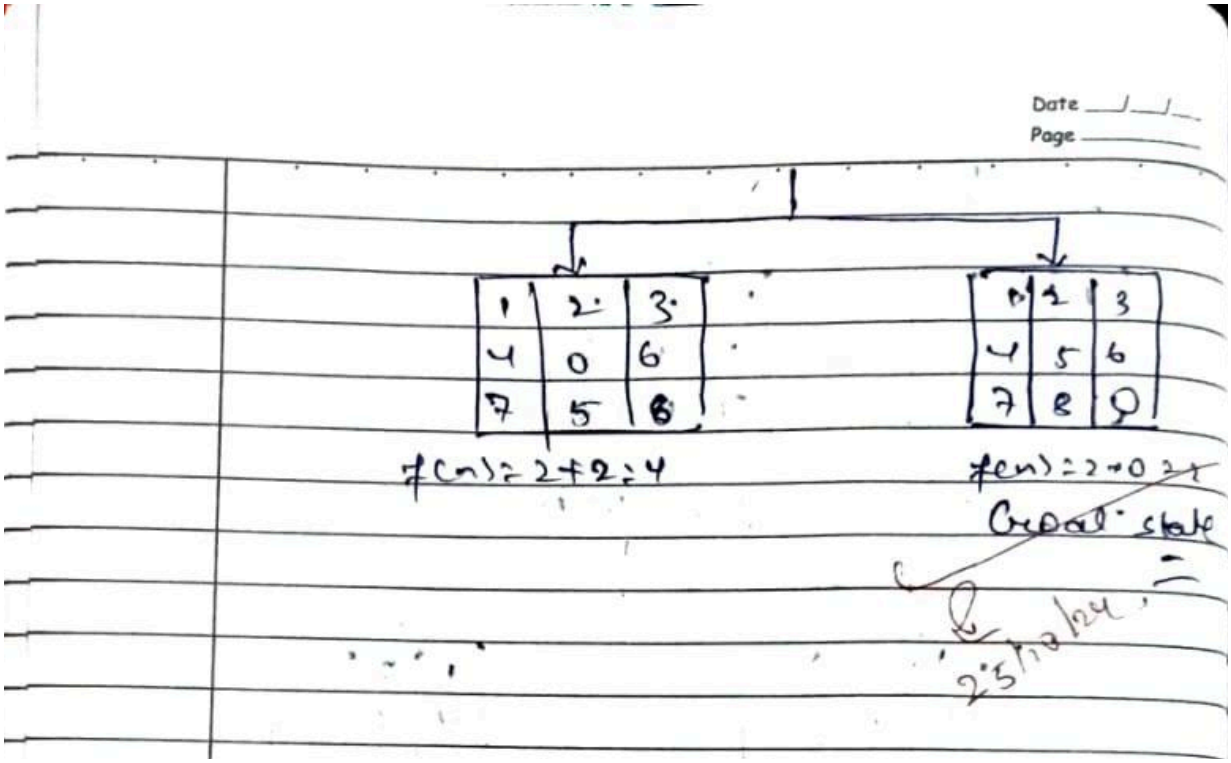
$f(n) = 0 + 3 = 3$

1	2	3
0	5	6
4	7	8

$f(n) = 1 + 4 = 5$

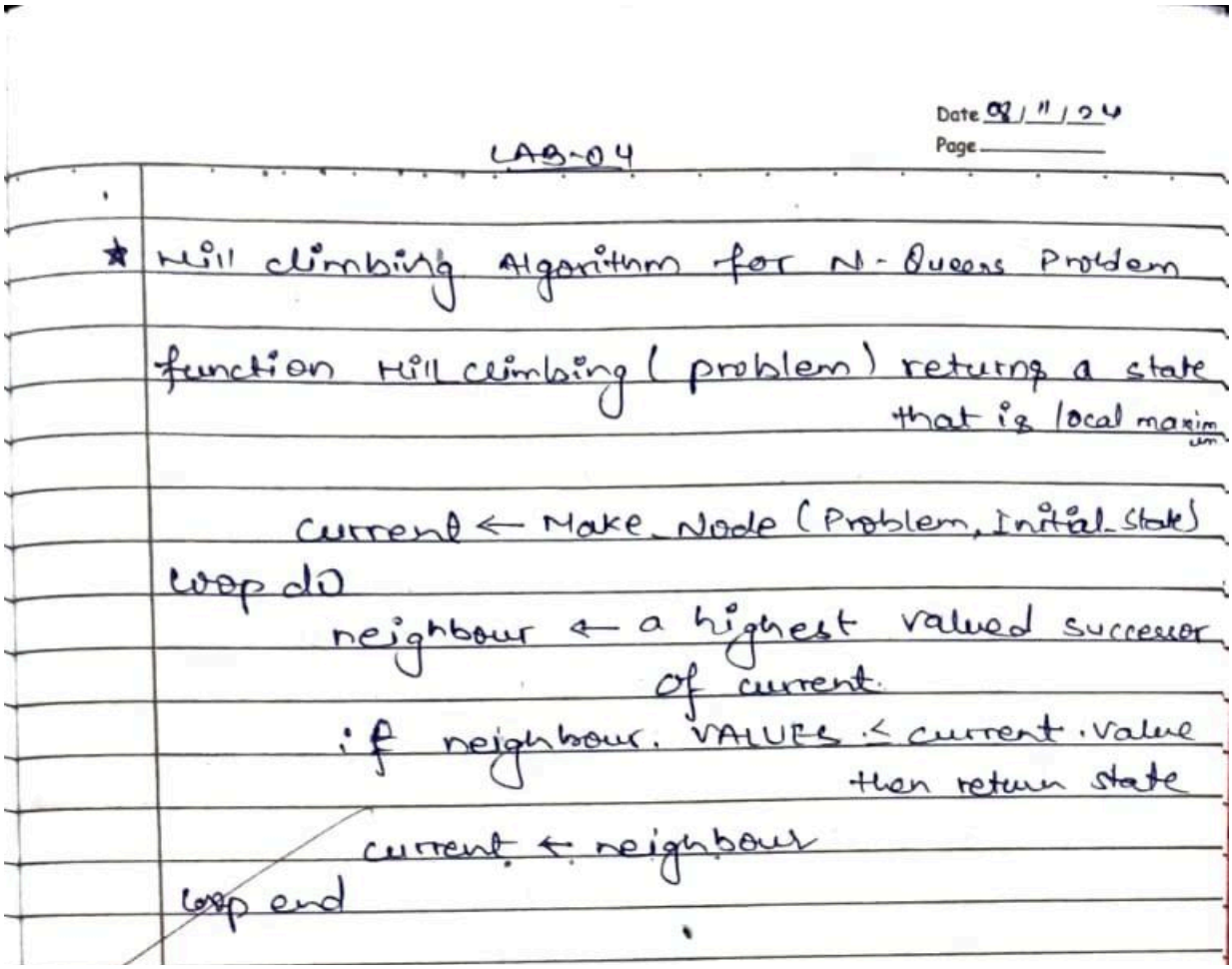
1	2	3
4	5	6
7	0	8

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


Date
Page
1 2 3
4 0 6
7 5 8
f(n) = 2+2 = 4
f(n) = 2+0
Goal state

PROGRAM 5:HILL CLIMBING(N-QUEENS)

Algorithm



CODE:

```
def count_conflicts(state):
    conflicts = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j]:
                conflicts += 1
            if abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
```

```
    return conflicts

def generate_neighbors(state):
    neighbors = []
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            neighbor = state[:]
            neighbor[i], neighbor[j] = neighbor[j], neighbor[i]  # Swap positions of queens i
and j
            neighbors.append(neighbor)
    return neighbors

def hill_climbing(n, initial_state):
    state = initial_state
    while True:
        current_conflicts = count_conflicts(state)
        if current_conflicts == 0:
            return state
        neighbors = generate_neighbors(state)
        best_neighbor = None
        best_conflicts = float('inf')
        for neighbor in neighbors:
            conflicts = count_conflicts(neighbor)
            if conflicts < best_conflicts:
                best_conflicts = conflicts
                best_neighbor = neighbor
        if best_conflicts < current_conflicts:
            state = best_neighbor
        else:
            return None
print('Swapnil Sahil(1BM22CS300)')

def get_user_input(n):
    while True:
```

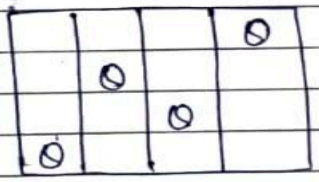
```
        try:
            user_input = input(f"Enter the row positions for the queens (space-separated
integers between 0 and {n-1}): ")
            initial_state = list(map(int, user_input.split()))
            if len(initial_state) != n or any(x < 0 or x >= n for x in initial_state):
                print(f"Invalid input. Please enter exactly {n} integers between 0 and {n-1}.")
                continue
            return initial_state
        except ValueError:
            print(f"Invalid input. Please enter a list of {n} integers.")

n = 4
initial_state = get_user_input(n)

solution = hill_climbing(n, initial_state)

if solution:
    print("Solution found!")
    for row in range(n):
        board = ['Q' if col == solution[row] else '.' for col in range(n)]
        print(' '.join(board))
else:
    print("No solution found (stuck in local minimum).")
```

OUTPUT:

```
Swapnil Sahil(1BM22CS300)
Enter the row positions for the queens (space-separated integers between 0 and 3): 3 1 2 0
Solution found!
. Q . .
. . . Q
Q . . .
. . Q .
```

STATE SPACE:

Q. 4- Queens Problem:



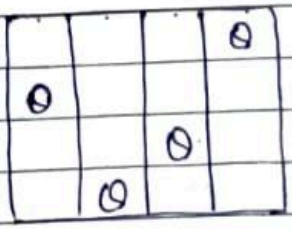
Initial state $x_0 = 3$, $x_1 = 1$, $x_2 = 2$, $x_3 = 0$
, cost = 2

- Neighbours:

$x_0 = 1$, $x_1 = 3$, $x_2 = 2$, $x_3 = 0$, cost = 1 (chosen)
$x_0 = 2$, $x_1 = 1$, $x_2 = 3$, $x_3 = 0$, cost = 1
$x_0 = 0$, $x_1 = 1$, $x_2 = 2$, $x_3 = 3$, cost = 6
$x_0 = 3$, $x_1 = 2$, $x_2 = 1$, $x_3 = 0$, cost = 6
$x_0 = 3$, $x_1 = 0$, $x_2 = 2$, $x_3 = 1$, cost = 1
$x_0 = 3$, $x_1 = 1$, $x_2 = 0$, $x_3 = 2$, cost = 1

Page

Next state chosen:

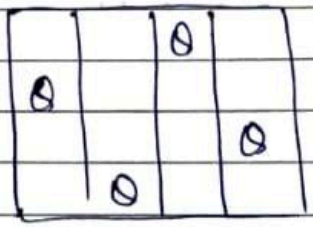


Neighbours:

$x_0 = 3$, $x_1 = 1$, $x_2 = 2$, $x_3 = 0$, cost = 2
$x_0 = 2$, $x_1 = 3$, $x_2 = 1$, $x_3 = 0$, cost = 2
$x_0 = 0$, $x_1 = 3$, $x_2 = 2$, $x_3 = 1$, cost = 4
$x_0 = 1$, $x_1 = 2$, $x_2 = 3$, $x_3 = 0$, cost = 4
$x_0 = 1$, $x_1 = 0$, $x_2 = 2$, $x_3 = 3$, cost = 2
$x_0 = 1$, $x_1 = 3$, $x_2 = 0$, $x_3 = 2$, cost = 0

Goal state

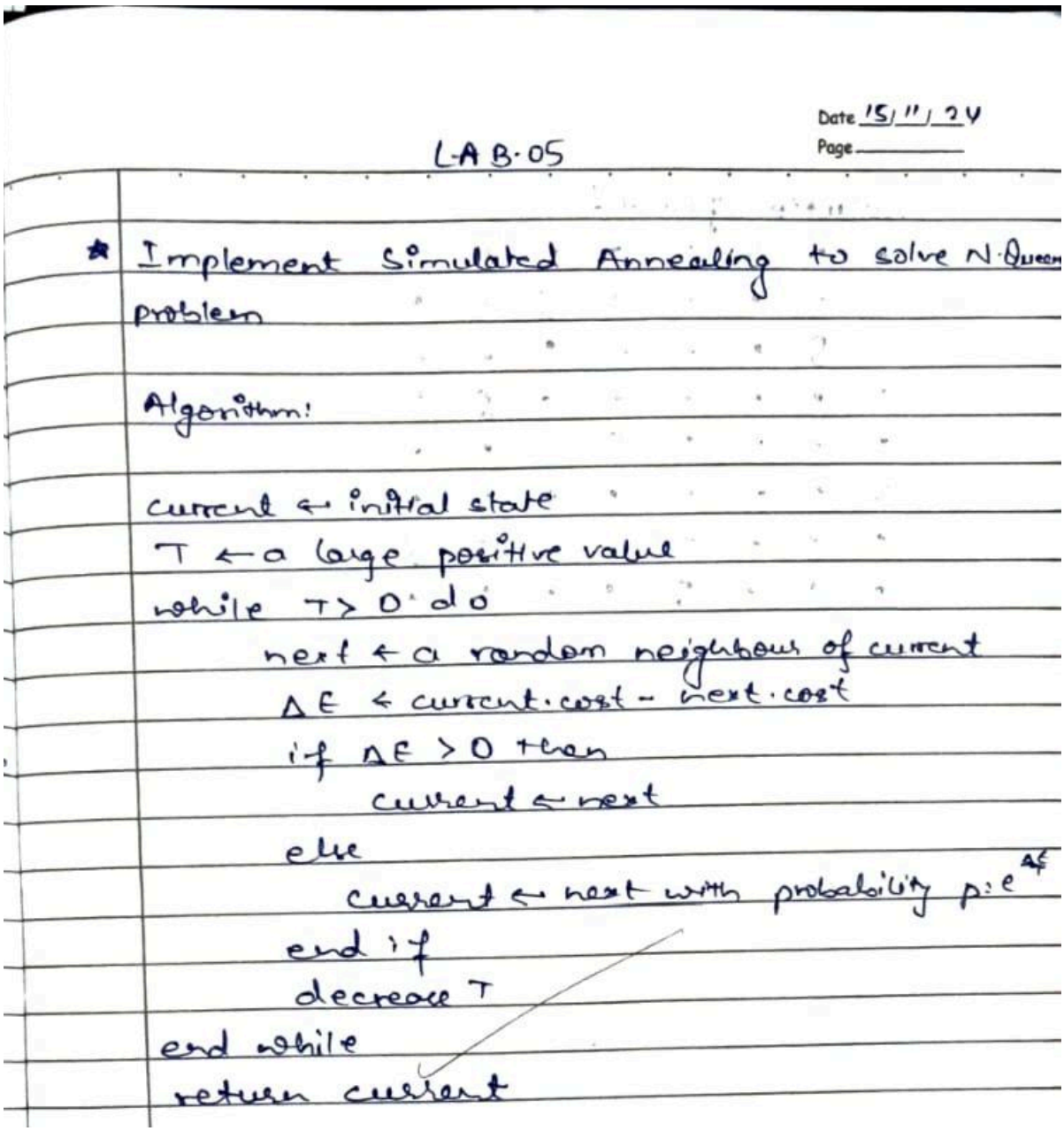
Goal state:



final state.

8/11/24

PROGRAM 6:SIMULATED ANNEALING ALGORITHM



LAB-05

Date 15/11/24
Page

★ Implement Simulated Annealing to solve N-Queen problem

Algorithm:

current ← initial state
T ← a large positive value
while T > 0 do
    next ← a random neighbour of current
    ΔE ← current.cost − next.cost
    if ΔE > 0 then
        current ← next
    else
        current ← next with probability p = $e^{\Delta E}$
    end if
    decrease T
end while
return current

CODE:

```
import random
import math

def calculate_conflicts(board):

    conflicts = 0
    n = len(board)
```

```
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j]:
                conflicts += 1
            elif abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def generate_neighbor(board):

    n = len(board)
    new_board = board[:]

    col = random.randint(0, n - 1)

    current_row = new_board[col]
    possible_rows = set(range(n)) - {current_row}

    valid_rows = set()
    for row in possible_rows:
        valid = True
        for c in range(n):
            if c != col and abs(row - new_board[c]) == abs(col - c):
                valid = False
                break
        if valid:
            valid_rows.add(row)

    if valid_rows:

        new_board[col] = random.choice(list(valid_rows))
    return new_board

def simulated_annealing(n, initial_state, max_iterations=10000, initial_temp=1000,
cooling_rate=0.99):
```

```
    """
    Simulated Annealing to solve the N-Queens problem.
    """
    current_state = initial_state
    current_conflicts = calculate_conflicts(current_state)
    temperature = initial_temp

    for iteration in range(max_iterations):
        if current_conflicts == 0:
            return current_state

        neighbor = generate_neighbor(current_state)
        neighbor_conflicts = calculate_conflicts(neighbor)

        delta = current_conflicts - neighbor_conflicts

        if delta > 0 or random.random() < math.exp(delta / temperature):
            current_state = neighbor
            current_conflicts = neighbor_conflicts

        temperature *= cooling_rate

    return None

def print_solution(board):
    """
    Prints the solution board in a human-readable format.
    """
    n = len(board)
    for row in range(n):
        board_row = ['Q' if col == board[row] else '.' for col in range(n)]
        print(' '.join(board_row))

print('Swapnil Sahil(1BM22CS300):')
```

```
n = int(input("Enter the number of queens: "))
initial_state_input = input(f"Enter the initial state (a list of {n} integers representing the row positions of queens in each column): ")

initial_state = list(map(int, initial_state_input.strip('[]').split(',')))


if len(initial_state) != n or any(queen < 0 or queen >= n for queen in initial_state):
    print("Invalid initial state! Please make sure it's a list of integers between 0 and n-1.")
else:
    solution = simulated_annealing(n, initial_state)

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

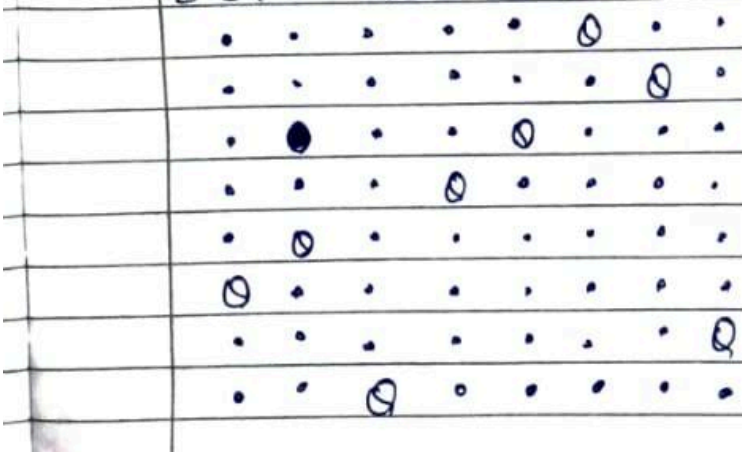
OUTPUT:

```
Swapnil Sahil(1BM22CS300):
Enter the number of queens: 8
Enter the initial state (a list of 8 integers representing the row positions of queens in each column): 0,1,2,3,4,5,6,7
Solution found:
. . . . . Q . .
. . . Q . . . .
. . . . . . Q .
Q . . . . . . .
. . . . . . . Q
. Q . . . . . .
. . . . Q . . .
. . Q . . . . .
```

// Output:

for 8 queens by taking random inputs

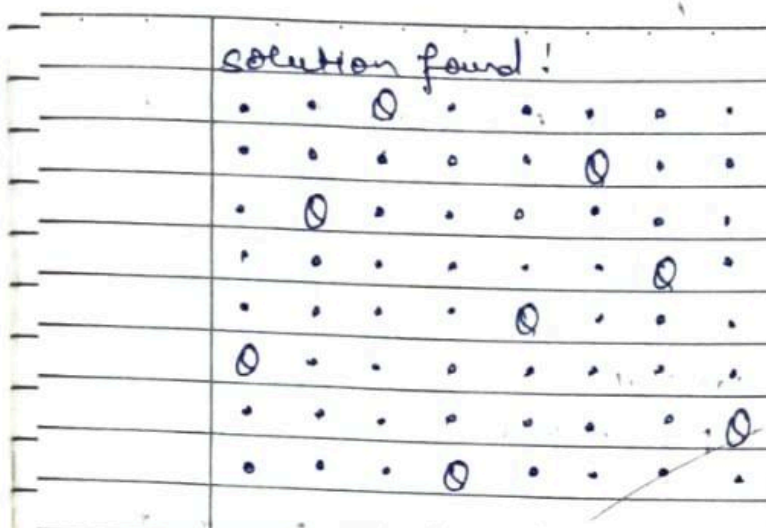
Initial state Board:



Date / /

Page

solution found!



PROGRAM 7:UNIFICATION IN FOL

ALGORITHM

Date 22/11/24
Page

LAB-06

★ Implement unification in first order logic

Algorithm: Unify(Ψ_1, Ψ_2)

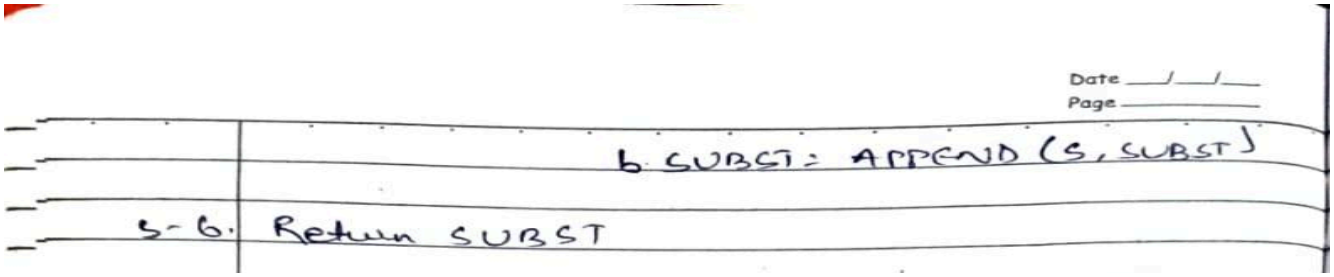
S-1 If Ψ_1 or Ψ_2 is a varible or constant, then:
a) If Ψ_1 or Ψ_2 are identical, then return NIL.
b) Else if Ψ_1 is a variable,
a. then if Ψ_1 occurs in Ψ_2, then return Failure
b. else return $\{(\Psi_2/\Psi_1)\}$.
c) Else if Ψ_2 is a variable,
a. If Ψ_2 occurs in Ψ_1, then return failure
b. Else return $\{(\Psi_1/\Psi_2)\}$.
d. Else return Failure

S-2 If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return Failure.

S-3 If Ψ_1 and Ψ_2 have a different number of arguments, then return Failure

S-4 Set substitution set (SUBST) to NIL.

S-5 For i=1 to the number of elements in Ψ_1
a) Call unify function with the ith element of Ψ_1 and ith element of Ψ_2 and put the result into S.
b) If S: failure then return Failure
c) If S ≠ NIL then do,
a. Apply S to the remainder of both L1 and L2



CODE:

```
#Implement unification in First Order Logic
def is_variable(x):
    """Checks if x is a variable (assuming variables are single lowercase letters)."""
    return isinstance(x, str) and x.islower() and len(x) == 1

def occurs_check(var, term):
    """Checks if a variable occurs in a term (used to avoid circular unification)."""
    if var == term:
        return True
    if isinstance(term, tuple):  # If term is a function (tuple), check its arguments.
        return any(occurs_check(var, t) for t in term)
    return False

def unify(x, y, substitution=None):
    """Unifies two terms x and y, applying substitutions."""
    if substitution is None:
        substitution = {}

    # Case 1: If both terms are the same, no unification needed
    if x == y:
        return substitution

    # Case 2: If x is a variable, try to unify
    elif is_variable(x):
        if x in substitution:
            return unify(substitution[x], y, substitution)
        elif occurs_check(x, y):
            raise ValueError(f"Unification fails due to occurs check for {x} in {y}")
```

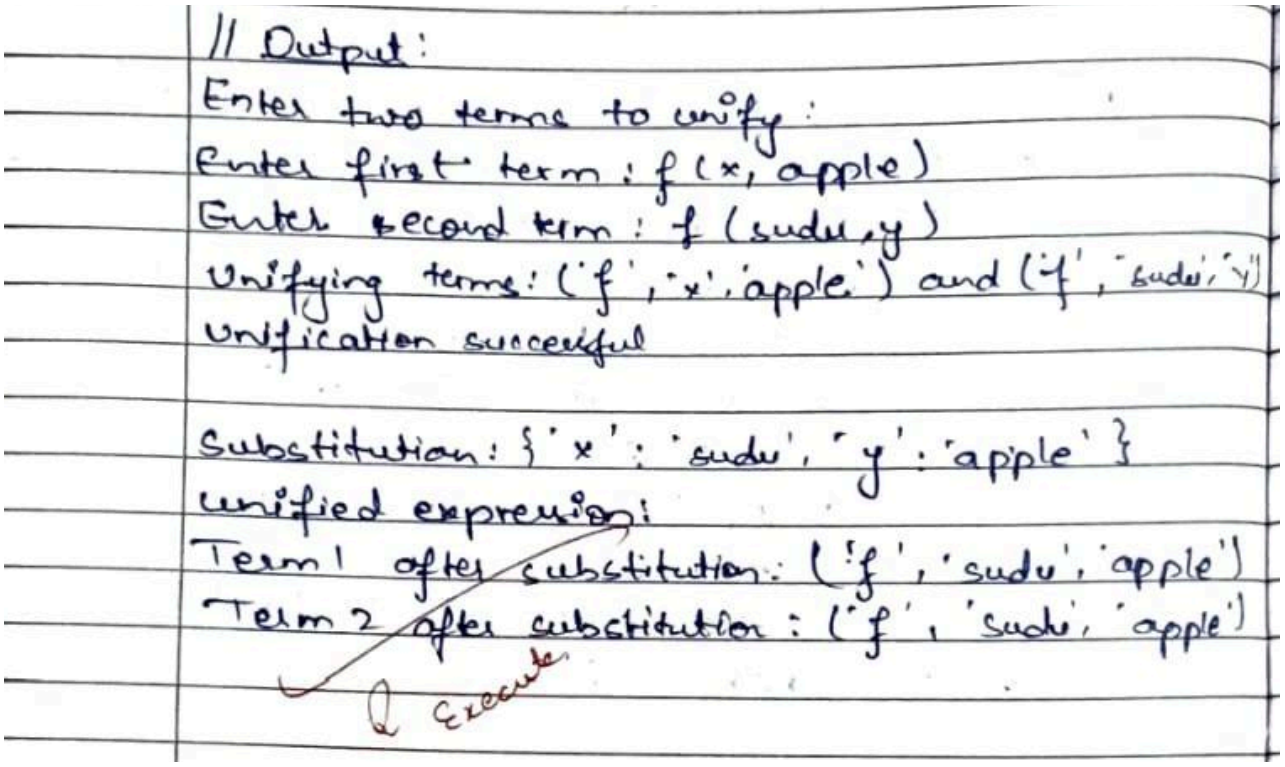
```
        else:
            substitution[x] = y
            return substitution

    # Case 3: If y is a variable, try to unify
    elif is_variable(y):
        return unify(y, x, substitution)

    # Case 4: If both terms are compound (functions), unify their components
    elif isinstance(x, tuple) and isinstance(y, tuple):
        if x[0] != y[0]:
            raise ValueError(f"Unification fails: {x[0]} != {y[0]}")
        # Recursively unify arguments
        for a, b in zip(x[1:], y[1:]):
            substitution = unify(a, b, substitution)
        return substitution

    # Case 5: Unification fails if x and y have no other cases
    else:
        raise ValueError(f"Unification fails: {x} cannot be unified with {y}")

def apply_substitution(term, substitution):
    """Applies the substitution to the term."""
    if isinstance(term, str):
        return substitution.get(term, term)
    elif isinstance(term, tuple):
        return (term[0], *[apply_substitution(t, substitution) for t in term[1:]])
    return term

def parse_term(term_str):
    """Parses a string representation of a term into a Python data structure."""
    term_str = term_str.strip()

    # Case 1: If it's a variable (single lowercase letter)
    if term_str.islower() and len(term_str) == 1:
```

```
        return term_str

    # Case 2: If it's a constant (any non-empty string, for example 'apple')
    if term_str.isalpha():
        return term_str

    # Case 3: If it's a function, e.g., 'f(x, y)'
    if term_str.startswith('f(') and term_str.endswith(')'):
        func_str = term_str[2:-1]  # Remove 'f(' and ')'
        parts = func_str.split(',')
        return ('f', *[parse_term(p.strip()) for p in parts])  # Function name, arguments

    # If none of these, raise an error
    raise ValueError(f"Invalid term format: {term_str}")
print('SWAPNIL SAHIL(1BM22CS300):')

def main():
    print("Enter two terms to unify (e.g., f(x, y), f(a, b)):")
    term1_str = input("Enter first term: ")
    term2_str = input("Enter second term: ")

    try:
        term1 = parse_term(term1_str)
        term2 = parse_term(term2_str)

        print(f"Unifying terms: {term1} and {term2}")
        substitution = unify(term1, term2)

        # Apply substitution to both terms to get the unified expression
        unified_term1 = apply_substitution(term1, substitution)
        unified_term2 = apply_substitution(term2, substitution)

        print("Unification successful!")
        print("Substitution:", substitution)
        print("Unified expression:")
```

```
        print(f"Term 1 after substitution: {unified_term1}")
        print(f"Term 2 after substitution: {unified_term2}")

    except ValueError as e:
        print("Unification failed:", e)

# Run the program
if __name__ == "__main__":
    main()
```

OUTPUT:

```
SWAPNIL SAHIL(1BM22CS300):
Enter two terms to unify (e.g., f(x, y), f(a, b)):
Enter first term: f(x,car)
Enter second term: f(bike,y)
Unifying terms: ('f', 'x', 'car') and ('f', 'bike', 'y')
Unification successful!
Substitution: {'x': 'bike', 'y': 'car'}
Unified expression:
Term 1 after substitution: ('f', 'bike', 'car')
Term 2 after substitution: ('f', 'bike', 'car')
```



PROGRAM 8:FORWARD CHAINING

ALGORITHM

Date 29/11/24
Page

1AI-07

* Create a knowledge base consisting of FOL statements and prove the given query using forward reasoning.

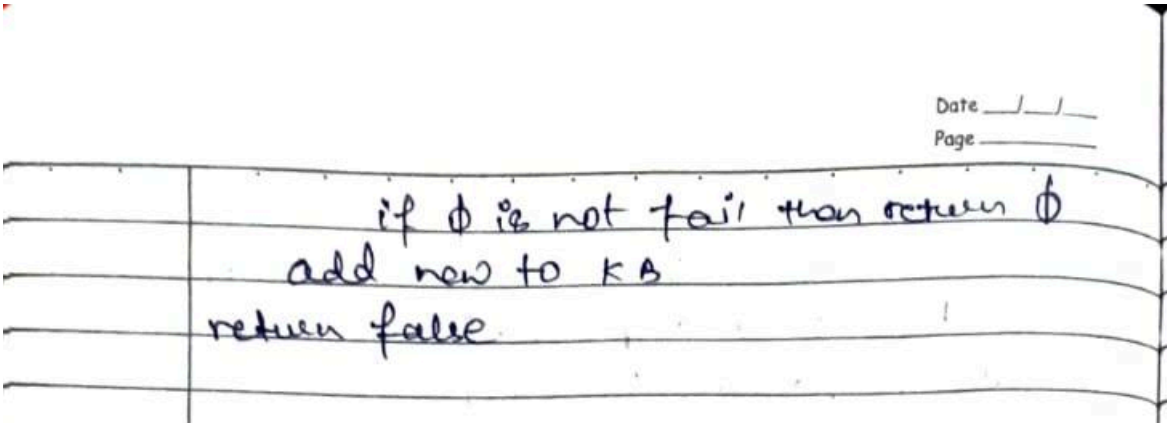
• Algorithm

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 ← { }
for each rule in KB do
(P1 ∧ ... ∧ Pn ⇒ q) ← standardize-variable(rule)
for each θ such that
subst(θ, P1 ∧ ... ∧ Pn) =
subst(θ, P1' ∧ ... ∧ Pn')
for some P1' ... Pn' in KB
q' ← subst(θ, q)
if q' does not unify with some sentence already in KB or new then
add q' to new
θ ← unify(q', α)



CODE:

```
knowledge_base = {
    "facts": {
        "American(Robert)": True,
        "Enemy(A, America)": True,
        "Owns(A, T1)": True,
        "Missile(T1)": True,
    },
    "rules": [
        {"if": ["Missile(x)"], "then": ["Weapon(x)"]},
        {"if": ["Enemy(x, America)"], "then": ["Hostile(x)"]},
        {"if": ["Missile(x)", "Owns(A, x)"], "then": ["Sells(Robert, x, A)"]},
        {
            "if": ["American(p)", "Weapon(q)", "Sells(p, q, r)", "Hostile(r)"],
            "then": ["Criminal(p)"],
        },
    ],
}

def forward_chaining(kb):
    facts = kb["facts"].copy()
    rules = kb["rules"]

    inferred = set()

    while True:
```

```
new_inferences = set()

for rule in rules:
    if_conditions = rule["if"]
    then_conditions = rule["then"]

    substitutions = {}
    all_conditions_met = True

    for condition in if_conditions:
        predicate, args = condition.split("(")
        args = args[:-1].split(",")

        matched = False
        for fact in facts:
            fact_predicate, fact_args = fact.split("(")
            fact_args = fact_args[:-1].split(",")

            if predicate == fact_predicate and len(args) == len(fact_args):
                temp_subs = {}
                for var, val in zip(args, fact_args):
                    if var.islower():
                        if var in temp_subs and temp_subs[var] != val:
                            break
                        temp_subs[var] = val
                    elif var != val:
                        break
                else:
                    matched = True
                    substitutions.update(temp_subs)
                    break

        if not matched:
            all_conditions_met = False
            break
```

```
            if all_conditions_met:
                for condition in then_conditions:
                    predicate, args = condition.split("(")
                    args = args[:-1].split(",")
                    new_fact = predicate + "(" + ",".join(substitutions.get(arg, arg) for arg in args)
+ ")"
                    new_inferences.add(new_fact)

        if new_inferences - inferred:
            inferred.update(new_inferences)
            facts.update({fact: True for fact in new_inferences})
        else:
            break

    return inferred

result = forward_chaining(knowledge_base)
print('SWAPNIL SAHIL(1BM22CS300):')

if "Criminal(Robert)" in result:
    print("Proved: Robert is a criminal.")
else:
    print("Could not prove that Robert is a criminal.")
```

OUTPUT:

SWAPNIL SAHIL(1BM22CS300):
Proved: Robert is a criminal.

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

Prove that "Robert is criminal".

Solution:

Step1: KB in FOL

- Fact 1 - It is a crime for an American to sell weapons to hostile nations.

→ FOL - $American(p) \wedge weapon(q) \wedge sells(p,q,r) \wedge Hostile(r) \Rightarrow Criminal(p)$

- Fact 2: Country A has some missiles

→ FOL: $\exists x\ Owns(A,x) \wedge Missile(x)$

- Fact 3: All the missiles were sold to country A by Robert

→ FOL: $\forall x\ Missile(x) \wedge Owns(A,x) \Rightarrow Sells(Robert, x, A)$

- Fact 4: Missiles are weapons

→ FOL: $Missile(x) \Rightarrow Weapon(x)$

Date __/__/__
Page _____

- Fact 5: Enemy of America is known as Hostile.
→ FOL: $\forall x$ Enemy(x, America) $\Rightarrow$ Hostile(x)

- Fact 6: Robert is an American
→ FOL: American(Robert)

- Fact 7: "The country A, an enemy of America
→ FOL: Enemy(A, America)

Step 2: Query
→ Robert is Criminal : criminal(Robert)

Step 3: Apply Inference
- By the rule $\forall x$ (Enemy(x, America) $\Rightarrow$ Hostile(x))
Enemy(A, America) $\Rightarrow$ Hostile(A)

$\therefore$ Hostile(A)

- By the rule Missile(x) $\Rightarrow$ Weapon(x)
Missile(T1) $\Rightarrow$ Weapon(T1)

$\therefore$ Weapon(T1)

- By the rule $\forall x$ (Missile(x) $\wedge$ Owns(A, x) $\Rightarrow$ Sells(Robert, x, A))

$\therefore$ Sells(Robert, T1, A)

We have: American(Robert)
Weapon(T1)
Hostile(A)
Sells(Robert, x, A)

$\therefore$ American(Robert) $\wedge$ Weapon(T1) $\wedge$ Sells(Robert, x, A) $\wedge$ Hostile(A) $\Rightarrow$ Criminal(Robert)

PROGRAM 9:ALPHA BETA PRUNING

ALGORITHM

Date __/__/__
Page _____

AB-08

* Implement Alpha-Beta Pruning

** Pseudocode:

```
Function ABPruning(depth, nodeIndex, isMaximizingPlayer,
                   values, alpha, beta, maxDepth):
    If depth == maxDepth:
        Return values[nodeIndex]

    If isMaximizingPlayer:
        best = -∞

        For i = 0 to 1
            val = ABPruning(depth+1, nodeIndex*2+i,
                            False, values, alpha, beta,
                            maxDepth)
            best = Max(best, val)
            alpha = max(alpha, best)

            If beta <= alpha:
                Break

        Return best
    Else:
        best = +∞
        For i = 0 to 1:
            val = ABPruning(depth+1, nodeIndex*2+i,
                            True, values, alpha, beta,
                            maxDepth)
            best = min(best, val)
            beta = min(beta, best)
            If beta <= alpha
                Break
    Return best
```

CODE:

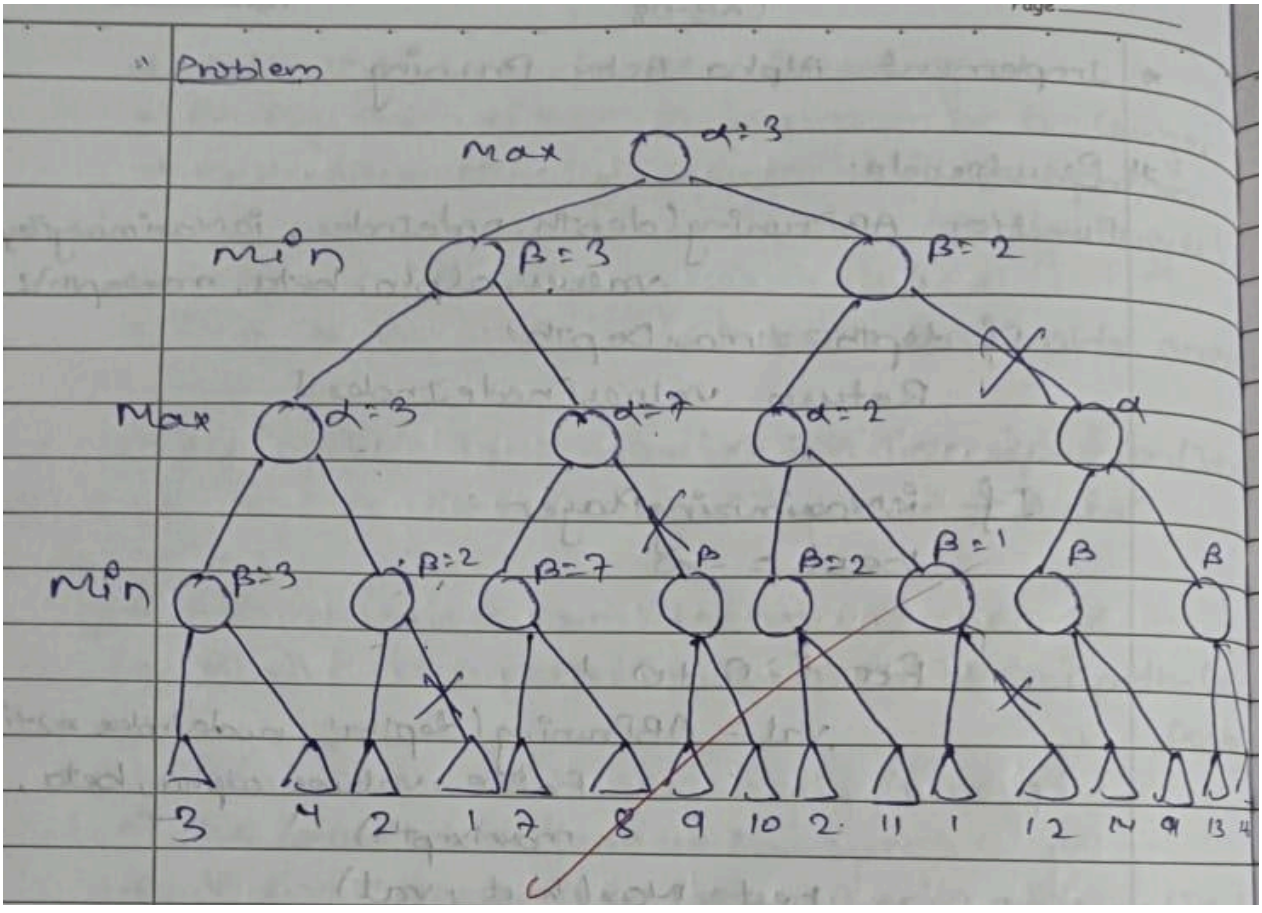
```
import math

def alpha_beta_pruning(depth, node_index, is_maximizing_player, values, alpha, beta, max_depth):
    # Base case: when the maximum depth is reached
    if depth == max_depth:
        return values[node_index]

    if is_maximizing_player:
        best = -math.inf

        # Recur for left and right children
        for i in range(2):
            val = alpha_beta_pruning(depth + 1, node_index * 2 + i, False, values, alpha, beta, max_depth)
            best = max(best, val)
            alpha = max(alpha, best)

            # Prune the remaining nodes
            if beta <= alpha:
                break
        return best

    else:
        best = math.inf

        # Recur for left and right children
        for i in range(2):
            val = alpha_beta_pruning(depth + 1, node_index * 2 + i, True, values, alpha, beta, max_depth)
            best = min(best, val)
            beta = min(beta, best)
```

```
            # Prune the remaining nodes
            if beta <= alpha:
                break
        return best
print("SWAPNIL SAHIL(1BM22CS300):")

# Example usage
if __name__ == "__main__":
    # Example tree represented as a list of leaf node values
    values = [3, 5, 6, 9, 1, 2, 0, -1]
    max_depth = 3  # Height of the tree

    result = alpha_beta_pruning(0, 0, True, values, -math.inf, math.inf, max_depth)
    print("The optimal value is:", result)
```

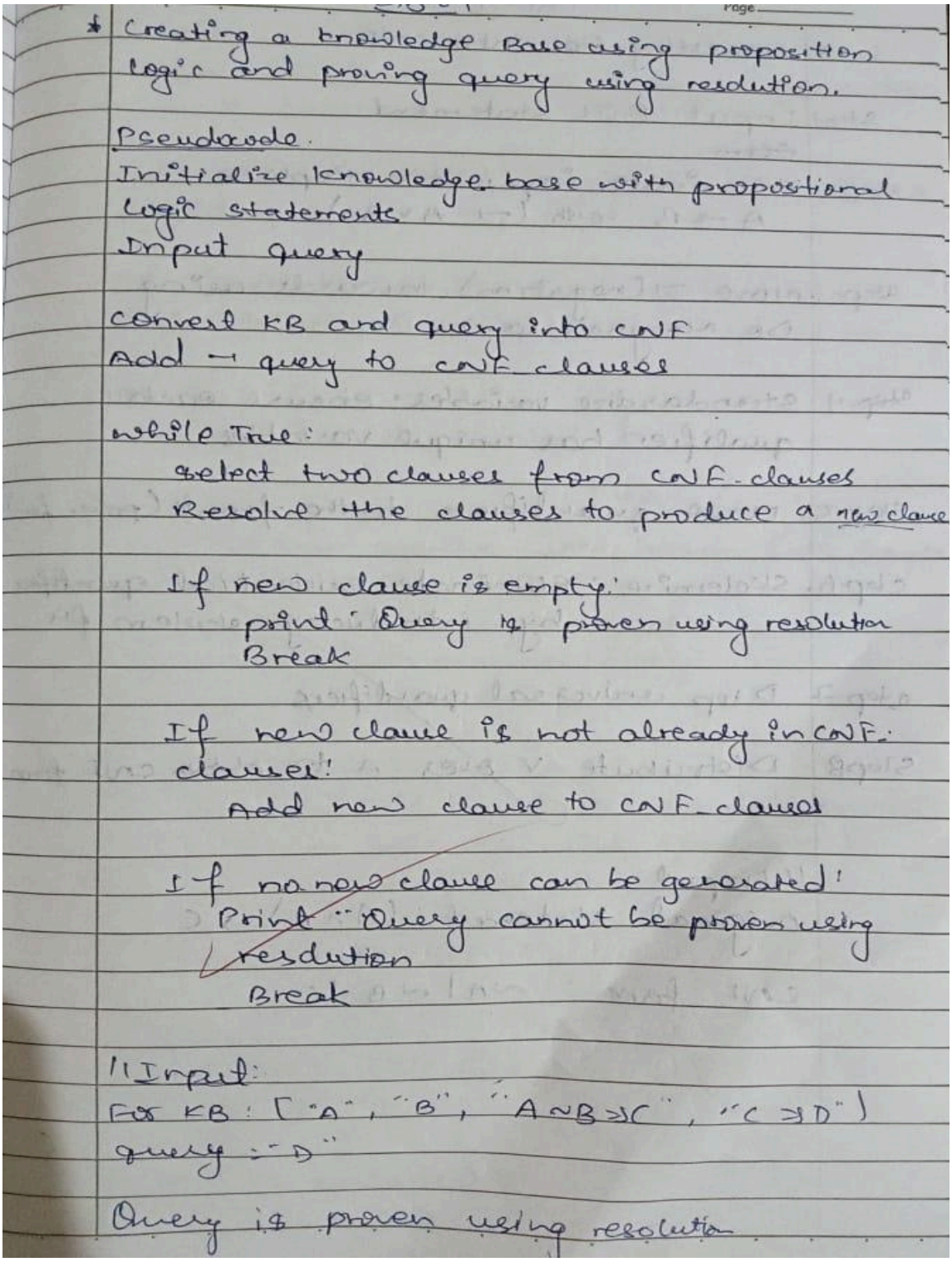
OUTPUT:

SWAPNIL SAHIL(1BM22CS300):

The optimal value is: 5



PROGRAM 10:PROVE A QUERY USING RESOLUTION ALGORITHM



* Creating a knowledge Base using proposition logic and proving query using resolution.

Pseudocode.

Initialize knowledge base with propositional logic statements

Input query

convert KB and query into CNF

Add ¬ query to CNF clauses

while True:

select two clauses from CNF clauses

Resolve the clauses to produce a new clause

If new clause is empty:

print "Query is proven using resolution"

Break

If new clause is not already in CNF clauses:

Add new clause to CNF clauses

If no new clause can be generated:

Print "Query cannot be proven using resolution"

Break

//Input:

For KB: ["A", "B", "A∧B⇒C", "C⇒D"]

query: "D"

Query is proven using resolution

CODE:

```
# Example propositional logic statements in CNF
kb = [
    {"¬B", "¬C", "A"},  # ¬B ∨ ¬C ∨ A
```

```
    {"B"},  # B
    {"¬D", "¬E", "C"},  # ¬D ∨ ¬E ∨ C
    {"E", "F"},  # E ∨ F
    {"D"}, #D
     {"¬F"}, #¬F
]
# Negate the query: If the query is "A", we negate it to "¬A"
def negate_query(query):
    if "¬" in query:
        return query.replace("¬", "")  # If it's negated, remove the negation
    else:
        return f"¬{query}"  # Otherwise, add negation in front

# Function to perform resolution on two clauses
def resolve(clause1, clause2):
    resolved_clauses = []

    # Try to find complementary literals
    for literal1 in clause1:
        for literal2 in clause2:
            # If literals are complementary (e.g., "A" and "¬A"), resolve them
            if literal1 == f"¬{literal2}" or f"¬{literal1}" == literal2:
                new_clause = (clause1 | clause2) - {literal1, literal2}
                resolved_clauses.append(new_clause)

    return resolved_clauses

# Perform resolution-based proof
def resolution(kb, query):
    # Step 1: Negate the query and add it to the knowledge base
    negated_query = negate_query(query)
    kb.append({negated_query})

    # Step 2: Initialize the set of clauses
    new_clauses = set(frozenset(clause) for clause in kb)
```

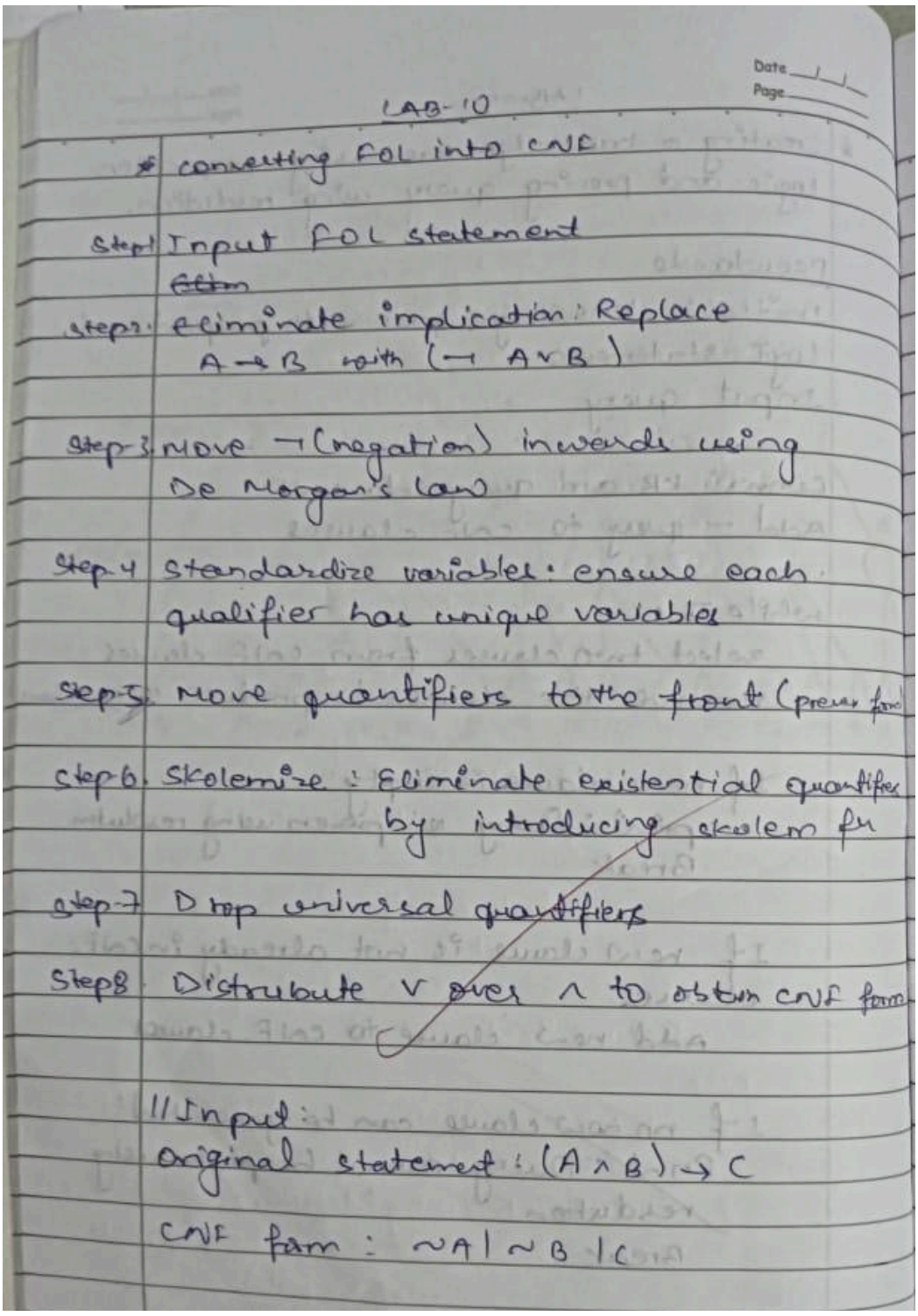
```
    while True:
        resolved_this_round = set()
        clauses_list = list(new_clauses)

        # Try to resolve every pair of clauses
        for i in range(len(clauses_list)):
            for j in range(i + 1, len(clauses_list)):
                clause1 = clauses_list[i]
                clause2 = clauses_list[j]

                # Apply resolution to the two clauses
                resolved = resolve(clause1, clause2)
                if frozenset() in resolved:
                    return True  # Found an empty clause (contradiction), query is provable
                resolved_this_round.update(resolved)

        # If no new clauses were added, stop
        if resolved_this_round.issubset(new_clauses):
            return False  # No new clauses, query is not provable
        # Add new resolved clauses to the set
        new_clauses.update(resolved_this_round)

# Query to prove: "A"
query = (input("Enter the query:"))
result = resolution(kb, query)
print("OUTPUT:(1BM22CS300)")
print("Using Resolution to prove a query")
print(f"Is the query '{query}' provable? {'Yes' if result else 'No'}")
```

OUTPUT:

```
Enter the query:A
OUTPUT:(1BM22CS300)
Using Resolution to prove a query
Is the query 'A' provable? Yes
```

PROGRAM 11:FOL TO CNF

ALGORITHM



LAB-10

* converting FOL into CNF

Step1 Input FOL statement

Step2 Eliminate implication: Replace $A \rightarrow B$ with $(\neg A \vee B)$

Step 3 Move $\neg$ (negation) inwards using De Morgan's laws

Step 4 Standardize variables: ensure each qualifier has unique variables

Step 5 Move quantifiers to the front (prenex form)

Step 6 Skolemize: Eliminate existential quantifiers by introducing skolem fn

Step 7 Drop universal quantifiers

Step 8 Distribute $\vee$ over $\wedge$ to obtain CNF form

// Input:

original statement: $(A \wedge B) \rightarrow C$

CNF form: $\sim A \mid \sim B \mid C$

CODE:

```
from sympy import symbols, Not, Or, And, Implies, Equivalent
from sympy.logic.boolalg import to_cnf

def fol_to_cnf(fol_expr):
```

```
    """
    Converts a First-Order Logic (FOL) statement to Conjunctive Normal Form (CNF).
    Arguments:
        fol_expr: A sympy logical expression representing the FOL statement.
    Returns:
        The CNF equivalent of the input expression.
    """
    # Step 1: Eliminate equivalences (A ↔ B) using (A → B) ∧ (B → A)
    fol_expr = fol_expr.replace(Equivalent, lambda a, b: And(Implies(a, b), Implies(b, a)))

    # Step 2: Eliminate implications (A → B) using (¬A ∨ B)
    fol_expr = fol_expr.replace(Implies, lambda a, b: Or(Not(a), b))

    # Step 3: Convert to CNF
    cnf_form = to_cnf(fol_expr, simplify=True)

    return cnf_form

def main():
    # Define propositional symbols instead of first-order predicates
    P = symbols("P")
    Q = symbols("Q")
    R = symbols("R")

    # Example 1: P → Q
    fol_expr1 = Implies(P, Q)
    print("Example 1: P → Q")
    print("Original FOL Expression:")
    print(fol_expr1)

    # Convert to CNF
    cnf1 = fol_to_cnf(fol_expr1)
    print("\nCNF Form:")
    print(cnf1)
```

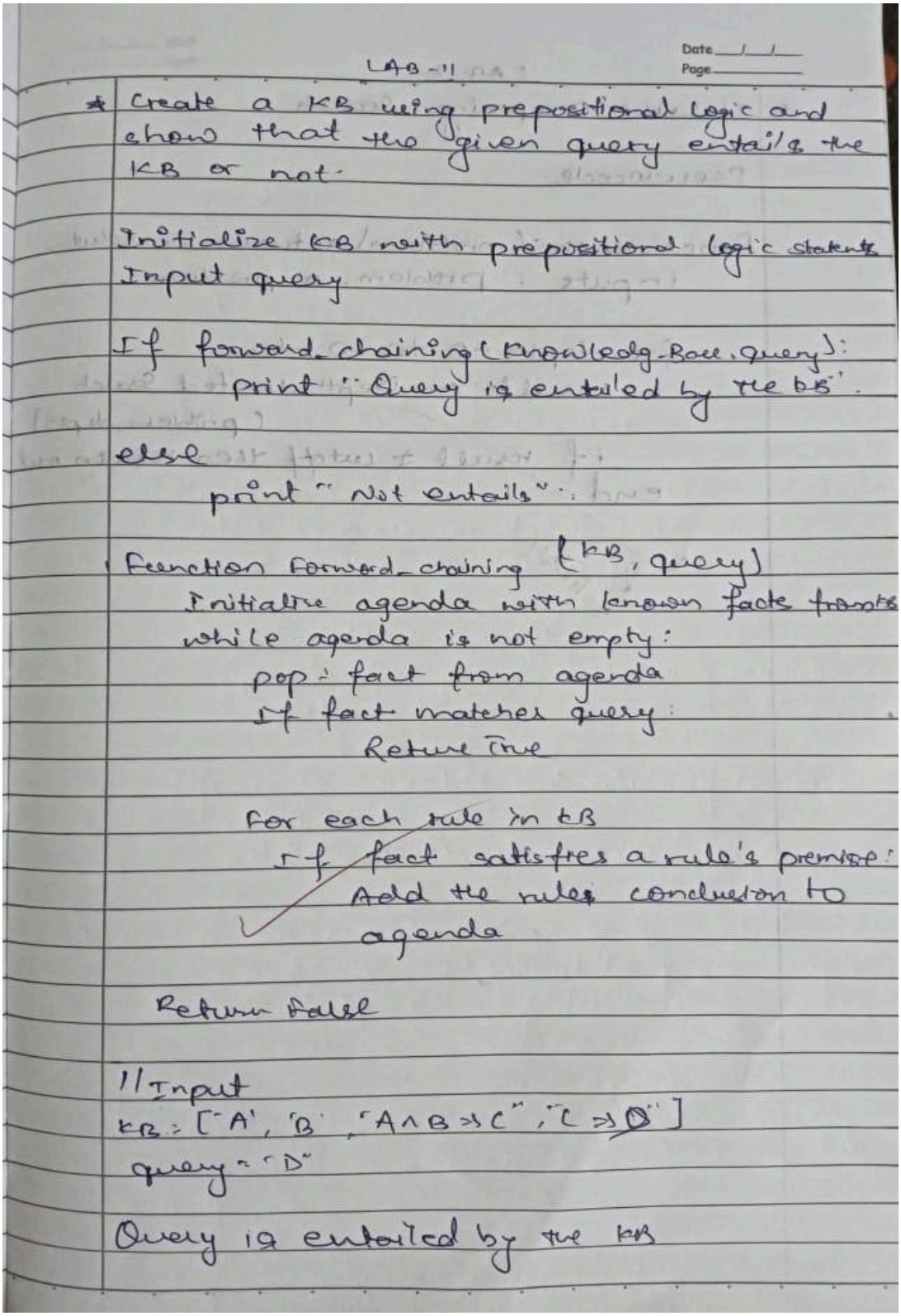
```
    # Example 2: (P ∨ ¬Q) → (Q ∨ R)
    fol_expr2 = Implies(Or(P, Not(Q)), Or(Q, R))
    print("\nExample 2: (P ∨ ¬Q) → (Q ∨ R)")
    print("Original FOL Expression:")
    print(fol_expr2)

    # Convert to CNF
    cnf2 = fol_to_cnf(fol_expr2)
    print("\nCNF Form:")
    print(cnf2)

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

OUTPUT:

```
Example 1: P → Q
Original FOL Expression:
Implies(P, Q)

CNF Form:
Q | ~P

Example 2: (P ∨ ¬Q) → (Q ∨ R)
Original FOL Expression:
Implies(P | ~Q, Q | R)

CNF Form:
Q | R
```

PROGRAM 12:ENTAILMENT

ALGORITHM



LAB-11

Date / /
Page

* Create a KB using propositional logic and show that the given query entails the KB or not.

Initialize KB with propositional logic statements
Input query

If forward_chaining(KnowledgeBase, query):
    print "Query is entailed by the KB".

else
    print "Not entails".

Function forward_chaining (KB, query)
    Initialize agenda with known facts from KB
    while agenda is not empty:
        pop fact from agenda
        If fact matches query:
            Return True

        For each rule in KB
            If fact satisfies a rule's premise:
                Add the rule's conclusion to agenda

    Return False

//Input
KB = ['A', 'B', 'A∧B⇒C', 'C⇒D']
query = 'D'

Query is entailed by the KB

CODE:

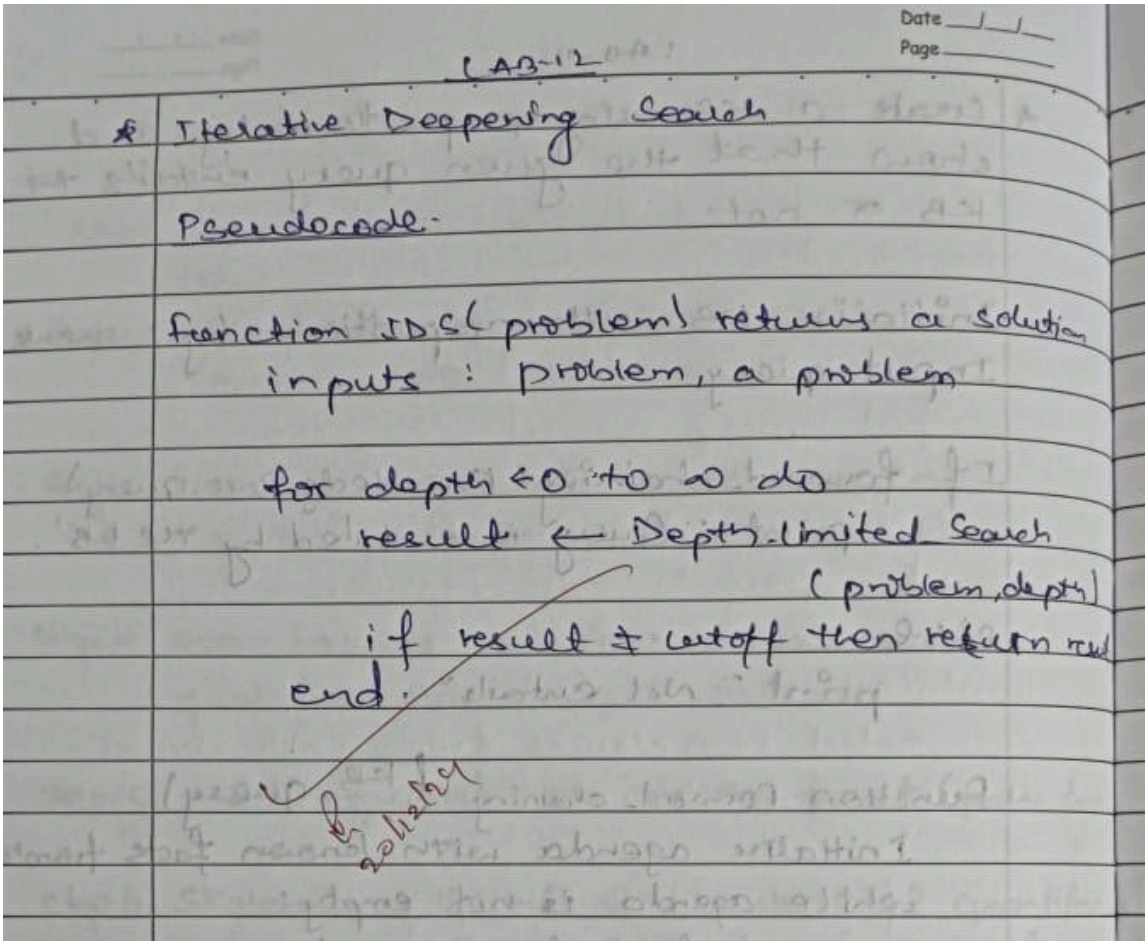
```
from sympy.logic.boolalg import Or, And, Not
from sympy.abc import A, B, C, D, E, F
```

```
from sympy import simplify_logic

def is_entailment(kb, query):

    # Negate the query
    negated_query = Not(query)

    # Add negated query to the knowledge base
    kb_with_negated_query = And(*kb, negated_query)

    # Simplify the combined KB to CNF
    simplified_kb = simplify_logic(kb_with_negated_query, form="cnf")

    # If the simplified KB evaluates to False, the query is entailed
    return simplified_kb == False



# Define a larger Knowledge Base
kb = [
    Or(A, B),         # A ∨ B
    Or(Not(A), C),    # ¬A ∨ C
    Or(Not(B), D),    # ¬B ∨ D
    Or(Not(D), E),    # ¬D ∨ E
    Or(Not(E), F),    # ¬E ∨ F
    F                 # F
]
# Query to check
query = Or(C, F)  # C ∨ F

# Check entailment
result = is_entailment(kb, query)
print(f"Is the query '{query}' entailed by the knowledge base? {'Yes' if result else 'No'}")
```

OUTPUT:

Is the query 'C | F' entailed by the knowledge base? Yes

PROGRAM 13:ITERATIVE DEEPENING SEARCH ALGORITHM



CODE:

```
from collections import defaultdict

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

    def add_edge(self, u, v):
        """Add an edge to the graph."""
        self.graph[u].append(v)

    def dls(self, node, target, depth):
        """
        Perform Depth-Limited Search (DLS) from the current node.
```

```
        :param node: Current node
        :param target: Target node
        :param depth: Maximum depth to explore
        :return: True if target is found, False otherwise
        """
        if depth == 0:
            return node == target
        if depth > 0:
            for neighbor in self.graph[node]:
                if self.dls(neighbor, target, depth - 1):
                    return True
        return False

    def iddfs(self, start, target, max_depth):
        """
        Perform Iterative Deepening Depth-First Search (IDDFS).

        :param start: Starting node
        :param target: Target node to search for
        :param max_depth: Maximum depth limit for IDDFS
        :return: True if target is found, False otherwise
        """
        for depth in range(max_depth + 1):
            print(f"Searching at depth: {depth}")
            if self.dls(start, target, depth):
                return True
        return False


# Example Usage
if __name__ == "__main__":
    g = Graph()
    # Construct the graph
    g.add_edge(0, 1)
```

```
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(1, 4)
g.add_edge(2, 5)
g.add_edge(2, 6)

start_node = 0
target_node = 5
max_depth = 3

# Perform IDDFS
if g.iddfs(start_node, target_node, max_depth):
    print(f"Target node {target_node} found within depth {max_depth}")
else:
    print(f"Target node {target_node} NOT found within depth {max_depth}")
```

OUTPUT:

```
Searching at depth: 0
Searching at depth: 1
Searching at depth: 2
Target node 5 found within depth 3
```