# VISVESVARAYA TECHNOLOGICAL UNIVERSITY

**"JnanaSangama", Belgaum -590014, Karnataka.**

**LAB REPORT**
**on**

# Artificial Intelligence (23CS5PCAIN)

***Submitted by***

**Sohan A R (1BM22CS285)**

***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 **Sohan A R (1BM22CS285),** 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.

| | |
|---|---|
| Prof. Sunayana S<br>Assistant Professor<br>Department of CSE, BMSCE | Dr. Jyothi S Nayak<br>Professor & HOD<br>Department of CSE, BMSCE |

# Index

LAB-1

1) IMPLEMENT TIC TAC TOE GAME

ALGORITHM //functions : print board (board), check winner (board), tic tac toe():

1) INITIALISE THE GAME BOARD :
- Create a 3x3 board initialized with spaces

2) Define the Players i.e Create a list of players

3) Set Turn and Move Counters i.e Initialize a variable to keep track of the current turn (starting from 0), initialize a counter for the number of moves made (starting from 0).

4) Game Loop
- Call the print key board function to show the current state of the board

5) Get User Input prompt the current player to enter the rows and column number

6) Validate Input, check if the chosen cell is empty and if empty place the symbol in chosen cell increment, check for the winner by invoking the check winner function if the winner is found display and announce winner.

7) If winner not found display a message indicating cell already filled and prompt for another input

24/9/22

8) Finally if all 9 moves are made without a winner, display final board and announce draw

Output:

```
   |   |
---|---|---
   |   |
---|---|---
   |   |
```

Player X, enter your row (0-2): 2
Player X, enter your column (0-2): 2

```
   |   |
---|---|---
   |   |
---|---|---
   |   | X
```

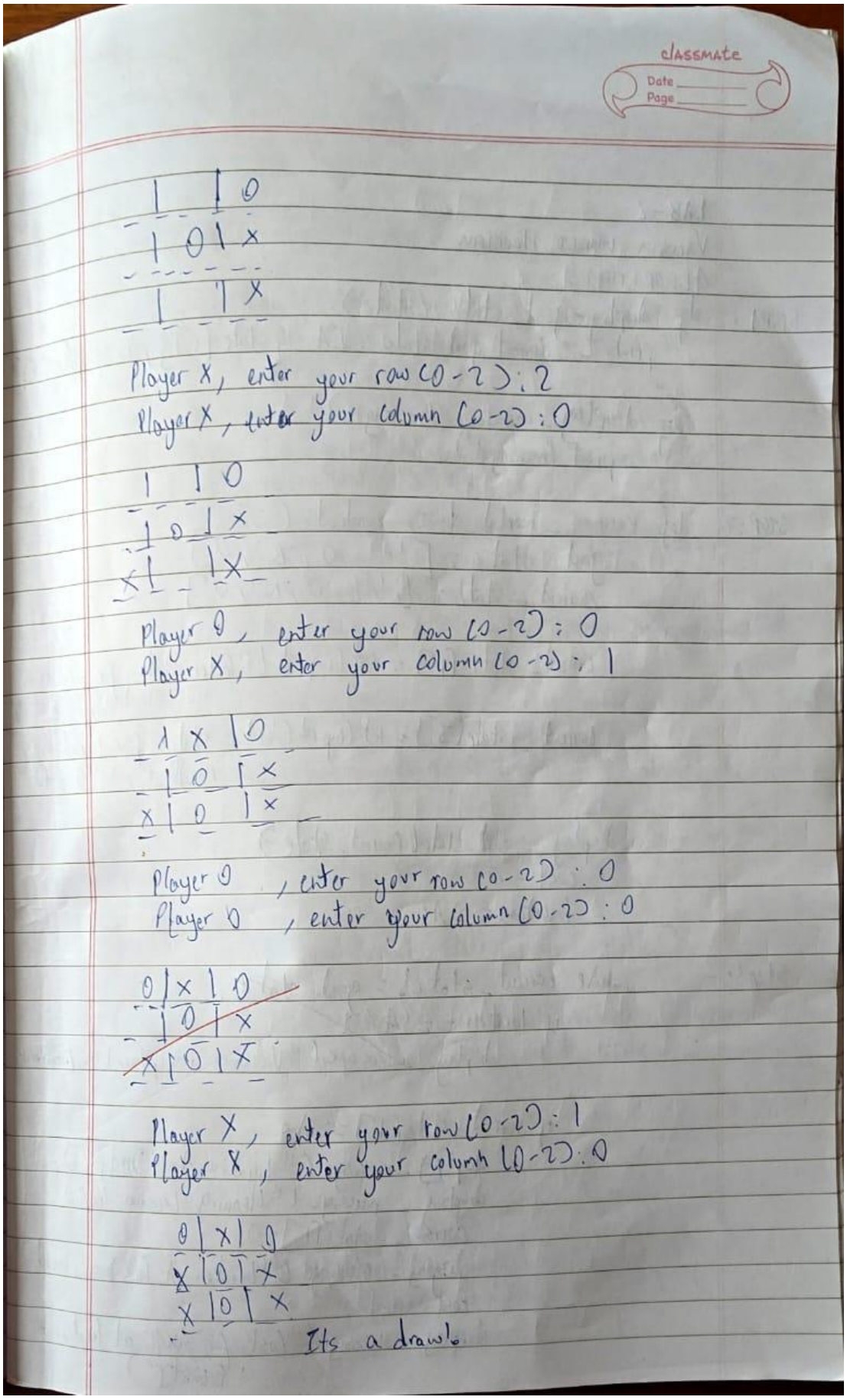
Player O, enter your row (0-2): 1
Player O, enter your column (0-2): 1

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

Player X, enter your row (0-2): 1
Player X, enter your column (0-2): 2

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

Player O, enter your row (0-2): 0
Player O, enter your column (0-2): 2



```
  |   | 0
---------
  | 0 | X
---------
  |   | X
```

Player X, enter your row (0-2): 2
Player X, enter your column (0-2): 0

```
  |   | 0
---------
  | 0 | X
---------
X |   | X
```

Player 0, enter your row (0-2): 0
Player X, enter your column (0-2): 1

```
  | X | 0
---------
  | 0 | X
---------
X | 0 | X
```

Player 0, enter your row (0-2): 0
Player 0, enter your column (0-2): 0

```
0 | X | 0
---------
  | 0 | X
---------
X | 0 | X
```

Player X, enter your row (0-2): 1
Player X, enter your column (0-2): 0

```
0 | X | 0
---------
X | 0 | X
---------
X | 0 | X
```

Its a draw!

**Program 1**

Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent
Tic-Tac-Toe

Code:

```
def check_win(board, r, c):
    if board[r - 1][c - 1] == 'X':
        ch = "O"
    else:
        ch = "X"
    if ch not in board[r - 1] and '-' not in board[r - 1]:
        return True
    elif ch not in (board[0][c - 1], board[1][c - 1], board[2][c - 1]) and '-' not in (board[0][c - 1],
board[1][c - 1], board[2][c - 1]):
        return True
    elif ch not in (board[0][0], board[1][1], board[2][2]) and '-' not in (board[0][0], board[1][1],
board[2][2]):
        return True
    elif ch not in (board[0][2], board[1][1], board[2][0]) and '-' not in (board[0][2], board[1][1],
board[2][0]):
        return True
    return False

def displayb(board):
  print(board[0])
  print(board[1])
  print(board[2])

board=[['-','-','-'],['-','-','-'],['-','-','-']]
displayb(board)
xo=1
flag=0
while '-' in board[0] or '-' in board[1] or '-' in board[2]:

  if xo==1:
    print("enter position to place X:")
    x=int(input())
    y=int(input())
    if(x>3 or y>3):
      print("invalid position")
      continue
    if(board[x-1][y-1]=='-'):
      board[x-1][y-1]='X'
      xo=0
      displayb(board)
    else:
```

```
      print("invalid position")
      continue
    if(check_win(board,x,y)):
       print("X wins")
       flag=1
       break
  else :
    print("enter position to place O:")
    x=int(input())
    y=int(input())
    if(x>3 or y>3):
      print("invalid position")
      continue
    if(board[x-1][y-1]=='-'):
      board[x-1][y-1]='O'
      xo=1
      displayb(board)
    else:
      print("invalid position")
      continue
    if(check_win(board,x,y)):
       print("0 wins")
       flag=1
       break
if flag==0:
  print("Draw")
print("Game Over")
```

```
['-', '-', '-']
['-', '-', '-']
['-', '-', '-']
enter position to place X:
1
1
['X', '-', '-']
['-', '-', '-']
['-', '-', '-']
enter position to place O:
1
2
['X', 'O', '-']
['-', '-', '-']
['-', '-', '-']
enter position to place X:
2
1
['X', 'O', '-']
['X', '-', '-']
['-', '-', '-']
enter position to place O:
2
2
['X', 'O', '-']
['X', 'O', '-']
['-', '-', '-']
enter position to place X:
3
1
['X', 'O', '-']
['X', 'O', '-']
['X', '-', '-']
X wins
Game Over
```

```
['-', '-', '-']
['-', '-', '-']
['-', '-', '-']
enter position to place X:
1
1
['X', '-', '-']
['-', '-', '-']
['-', '-', '-']
enter position to place O:
2
2
['X', '-', '-']
['-', 'O', '-']
['-', '-', '-']
enter position to place X:
3
3
['X', '-', '-']
['-', 'O', '-']
['-', '-', 'X']
enter position to place O:
1
2
['X', 'O', '-']
['-', 'O', '-']
['-', '-', 'X']
enter position to place X:
3
2
['X', 'O', '-']
['-', 'O', '-']
['-', 'X', 'X']
enter position to place O:
3
1
['X', 'O', '-']
['-', 'O', '-']
['O', 'X', 'X']
enter position to place X:
2
1
['X', 'O', '-']
['X', 'O', '-']
['O', 'X', 'X']
enter position to place O:
2
3
['X', 'O', '-']
['X', 'O', 'O']
['O', 'X', 'X']
enter position to place X:
1
3
['X', 'O', 'X']
['X', 'O', 'O']
['O', 'X', 'X']
Draw
Game Over
```

LAB-2

VACUUM WORLD PROBLEM

ALGORITHM:

STEP 1. def display_goal_state(state):
print("current goal state: [A: {state[1]}, B: {state[3]}]")

def display_message(message):
print(message)

STEP: 2 def vacuum_world_two_quadrants():
goal_state = ['A', 0, 'B', 0]
current_state = ['A', 0, 'B', 0]

current_state[1] = int(input("Enter status for location A
(0: No dust, 1: Dust:"))

current_state[3] = int(input("Enter status for location B
(0: No dust, 1: Dust:"))

display_goal_state(current_state)
cost = 0
location = 'A'

Step 3:- while current_state != goal_state:
if location == 'A':
display_message("Vacuum is placed in location A")

if current_state[1] == 1:
display_message("Location A is Dirty")
display_message("Cleaning Location A...")
current_state[1] = 0
display_message("Location A has been cleaned")
cost += 1
display_message("Cost for SUCK at location A
: {cost}")

step 4: else:
display_message ("Location A is already clean")

display_message ("Moving right to location B...")
location 'B'

step 5: elif location == 'B':
display_message ("Vacuum is placed in location B")
if current_state[3] == 1:
display_message ("Location B is Dirty")
display_message ("Cleaning location B...")
current_state[3] = 0
display_message ("Location B has been cleaned")
cost 1 = 1
display_message ("cost for SUCK at location B: " + str(cost1))
else:
display_message ("Location B is already clean")

display_message ("Moving left to location A...")
location = 'A'

display_goal_state (current_state)

display_message ("Goal state reached: Both rooms are clean")
display_goal_state (current_state)

vacuum_world_two_quadrants ().

Output:
Enter the status for Location A (0: No Dust, 1: Dust): 1
Enter the status for location B (0: No Dust, 1: Dust): 1
current Goal state: [A: 1, B: 1]

Cleaning location B...
Location B has been cleaned
Cost for SUCK at location B: 1
Moving left to Location A....
Current Goal state: [A:0, B:0]
Goal state reached. Both rooms are clean
Current Goal State: [A:0, B:0]

4 Quadrant Vacuum Cleaner Problem.

Algorithm:

STEP 1: Define A function to Display the current goal state

STEP 2: Define the function to display the messages based on the vacuum actions

STEP 3: Take the initial input for the room conditions and initialize the cost counter and print the initial state

STEP 4:

```
while current_state != goal state
    if location == 'A':
        display_message("Vacuum is placed in Location A")
        display_message("Location A is Dirty")
        display_message("Cleaning Location A...")
        current state[1] = 0
        display_message("Location A has been cleaned")
        cost += 1
        display_message("Cost for SUCK at location A:")
```

Vacuum is placed in location A
location A is dirty
Cleaning location A...
Location A is Cleaned.
COST for Suck at location A: 1
Moving right to location B...
Current Goal state: [A: 0, B: 1]
Vacuum is placed in location B.
Location B is Dirty
cleaning Location B...
location B has been Cleaned.
COST for SUCK at location B: 2
Current Goal state: [A: 0, B: 0]
Goal state reached: Both rooms are clean

2) Enter status for location A [0: No Dust, 1: Dust]: 0

Enter status for location B [0: No Dust, 1: Dust]: 0
Current Goal state: [A: 0: B 0]
Goal state reached: Both rooms are clean
Current Goal state: [A: 0, B: 0]

3 Enter status for location A [0: No dust, 1: Dust]: 0
Enter status for location B [0: No Dust, 1: Dust]: 1

Current Goal State: [A: 0, B: 1]
Vacuum is Placed in location A.
location A is already clean.
Moving right to location B....
Current Goal state: [A: 0, B: 1]
Vacuum is placed in location B.
Location B is Dirty

STEPS: Move to location and display the current goal state after each move

STEP 6: When the goal is reached
display - message ("Goal state reached, All rooms are
display - goal state ("clean")

STEP: 7 · Run the simulation for four Quadrats
vacuum - world - four - quadrants()

Output:
Enter the status for location A (0: No Dust, 1: Dust): 0
Enter the status for location B (0: No Dust, 1: Dust): 0
Enter the status for location C (0: No Dust, 1: Dust): 0
Enter the status for location D (0: No Dust, 1: Dust): 0
Current Goal State: [A: 0, B: 0, C: 0, D: 0]
Goal state reached: All rooms are clean
Current Goal state: [A: 0, B: 0, C: 0, D: 0]

11/10/2024

Vacuum Cleaner

Code:

```
count = 0
def rec(state, loc):
    global count
    if state['A'] == 0 and state['B'] == 0:
        print("Turning vacuum off")
        return

    if state[loc] == 1:
        state[loc] = 0
        count += 1
        print(f"Cleaned {loc}.")
        next_loc = 'B' if loc == 'A' else 'A'
        state[loc] = int(input(f"Is {loc} clean now? (0 if clean, 1 if dirty): "))
        if(state[next_loc]!=1):
          state[next_loc]=int(input(f"Is {next_loc} dirty? (0 if clean, 1 if dirty): "))
    if(state[loc]==1):
      rec(state,loc)
    else:
      next_loc = 'B' if loc == 'A' else 'A'
      dire="left" if loc=="B" else "right"
      print(loc,"is clean")
      print(f"Moving vacuum {dire}")
      if state[next_loc] == 1:
          rec(state, next_loc)

state = {}
state['A'] = int(input("Enter state of A (0 for clean, 1 for dirty): "))
state['B'] = int(input("Enter state of B (0 for clean, 1 for dirty): "))
loc = input("Enter location (A or B): ")
rec(state, loc)
print("Cost:",count)
print(state)
```

```
Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A
Turning vacuum off
Cost: 0
{'A': 0, 'B': 0}
```

```
Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): A
A is clean
Moving vacuum right
Cleaned B.
Is B clean now? (0 if clean, 1 if dirty): 0
Is A dirty? (0 if clean, 1 if dirty): 0
B is clean
Moving vacuum left
Cost: 1
{'A': 0, 'B': 0}
```

```
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A
Cleaned A.
Is A clean now? (0 if clean, 1 if dirty): 0
Is B dirty? (0 if clean, 1 if dirty): 0
A is clean
Moving vacuum right
Cost: 1
{'A': 0, 'B': 0}
```

```
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): A
Cleaned A.
Is A clean now? (0 if clean, 1 if dirty): 0
A is clean
Moving vacuum right
Cleaned B.
Is B clean now? (0 if clean, 1 if dirty): 0
Is A dirty? (0 if clean, 1 if dirty): 0
B is clean
Moving vacuum left
Cost: 2
{'A': 0, 'B': 0}
```

8-10-24

LAB-3

8 puzzle problem using BFS and DFS algorithm without Heuristic approach

ALGORITHM

STEP 1:

Initalize Puzzle: State state - Define the configuration of the puzzle
Goal state: Define the desired final configuration of the puzzle

STEP 2: Create a Node

Node Structure: Each node holds:

* state: Current puzzle configuration
* parent: The node that led to this configuration
* action: The move (up, down, left, right) that resulted in this state

STEP 3: Choose Search algorithm (BFS and DFS)

The user can choose:

BFS: Explore all states at the current level before going deeper
DFS: Explore as deep as possible before backtracking

STEP 4: Initalize the frontier

* Frontier: The list of nodes to be explored
  BFS: Use a queue (FIFO - first in first Out)
  DFS: Use a stack (LIFO - Last in first Out)
* Add the start node to the frontier

STEP-5:

Main Search Loop

Repeat the following steps until the goal state is found or no solution

1- Check if the frontier is empty.
   If yes, no solution exists, so terminate the search

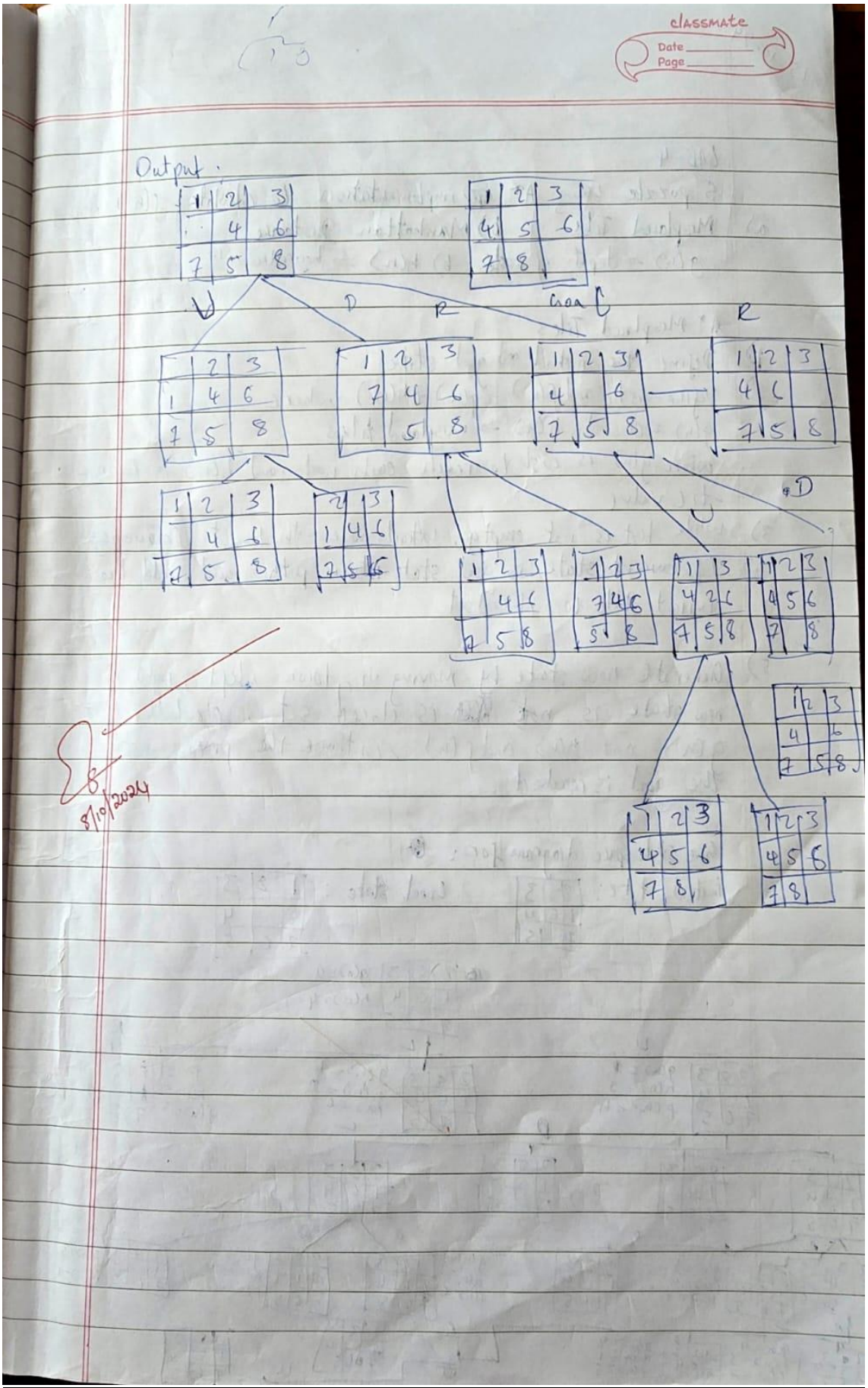
2 Remove a Node :
* For BFS : Remove the oldest node added (FIFO)
* for DFS : Remove the most recent node added (LIFO)

3 check if the current state is the Goal
* If yes, trace back from the current node to the start node using the parent links
* collect the sequence of actions (moves) that led to the goal
* Exit the loop

4. Add Neighbouring states to the frontier
* Generate all possible next states by moving the empty tile (0) up, down, left, or right.
* for each new state
  * Add it to the frontier if it hasn't already been explored or isn't in the frontier

6 Solution found
If a solution is found :
trace back from the goal state to the start state, output the sequence of moves and the corresponding puzzle states

7 Print the Solution
Display the :
* start state
* Goal state
* Number of States Explored
* Solution steps


classmate
Date
Page
Output.
goal
8/10/2024

**Program 2**

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

8 puzzle using DFS

Code:

```
def dfs(initial_board, zero_pos):
    stack = [(initial_board, zero_pos, [])]
    visited = set()

    while stack:
        current_board, zero_pos, moves = stack.pop()

        if is_goal(current_board):
            return moves, len(moves)  # Return moves and their count

        visited.add(tuple(current_board))

        for neighbor_board, neighbor_pos in get_neighbors(current_board, zero_pos):
            if tuple(neighbor_board) not in visited:
                stack.append((neighbor_board, neighbor_pos, moves + [neighbor_board]))

    return None, 0  # No solution found, return count as 0

# Initial state of the puzzle
initial_board = [1, 2, 3, 0, 4, 6, 7, 5, 8]
zero_position = (1, 0)  # Position of the empty tile (0)

# Solve the puzzle using DFS
solution, move_count = dfs(initial_board, zero_position)

if solution:
    print("Solution found with moves ({} moves):".format(move_count))
    for move in solution:
        print_board(move)
        print()  # Print an empty line between moves
else:
    print("No solution found.")
```

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

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

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

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

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

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

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

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

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

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

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

Implement Iterative deepening search algorithm

Code:

```
from collections import deque

class PuzzleState:
    def __init__(self, board, zero_pos, moves=0, previous=None):
        self.board = board
        self.zero_pos = zero_pos  # Position of the zero tile
        self.moves = moves          # Number of moves taken to reach this state
        self.previous = previous    # For tracking the path

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

    def get_possible_moves(self):
        moves = []
        x, y = self.zero_pos
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = [row[:] for row in self.board]
                # Swap the zero tile with the adjacent tile
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
                moves.append((new_board, (new_x, new_y)))
        return moves

def ids(initial_state, goal_state, max_depth):
    for depth in range(max_depth):
        visited = set()
        result = dls(initial_state, goal_state, depth, visited)
        if result:
            return result
    return None

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

    visited.add(tuple(map(tuple, state.board)))  # Mark this state as visited
    for new_board, new_zero_pos in state.get_possible_moves():
        new_state = PuzzleState(new_board, new_zero_pos, state.moves + 1, state)
```

```
        if tuple(map(tuple, new_board)) not in visited:
            result = dls(new_state, goal_state, depth - 1, visited)
            if result:
                return result
    visited.remove(tuple(map(tuple, state.board)))  # Unmark this state
    return None

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

# Define the initial state and goal state
initial_state = PuzzleState(
    board=[[1, 2, 3],
           [4, 0, 5],
           [7, 8, 6]],
    zero_pos=(1, 1)
)

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

# Perform Iterative Deepening Search
max_depth = 20  # You can adjust this value
solution = ids(initial_state, goal_state, max_depth)

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

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

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

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

15-10-24

LAB-4

8 puzzle using A* implementation to calculate f(n) using

a) Misplaced Tile b) Manhattan Distance

g(n) → depth of node b) h(n) → heuristic value

A* Misplaced Tiles

1) Define the initial and goal state
2) Define cost and f(n) = g(n) + h(n) where g(n) = 0 and h(n) → misplaced tiles where g(n) is cost to reach each node and h(n) is heuristic value
3) while list is not empty, extract node with lowest frequency
4) If current state is goal state return path and add the current state to closed set
5) Generate new state by moving up, down, left, right if new state is not is closed set, calculate g(n) and h(n) and f(n) / continue the process until the goal is reached.

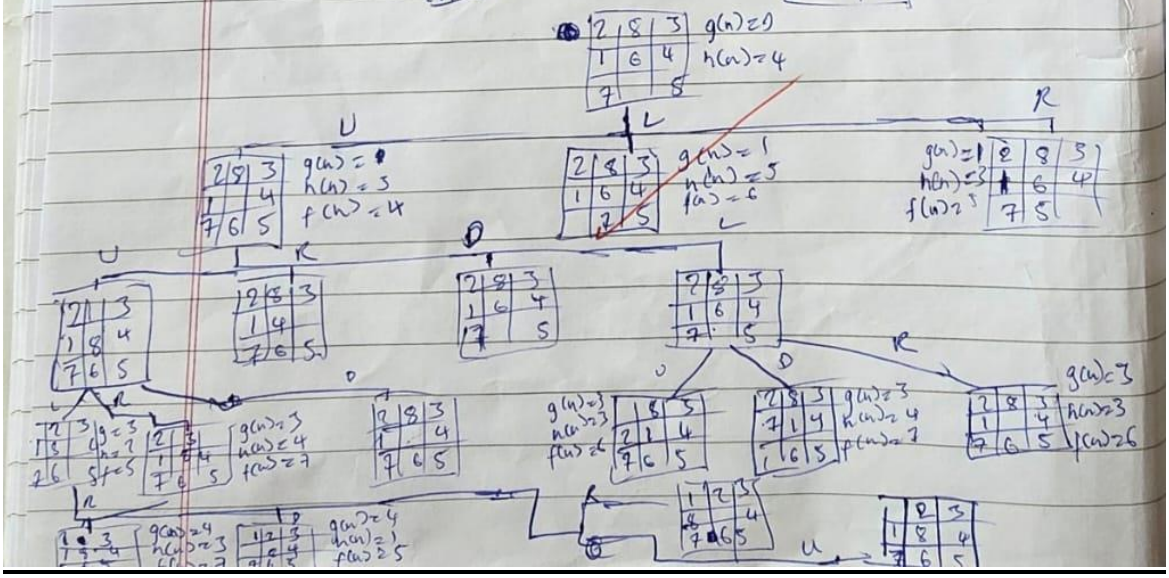
Give state space diagram for :

Initial state:

| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 6 | 4 |
| 7 | | 5 |

Goal state :

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | | 4 |
| 7 | 6 | 5 |



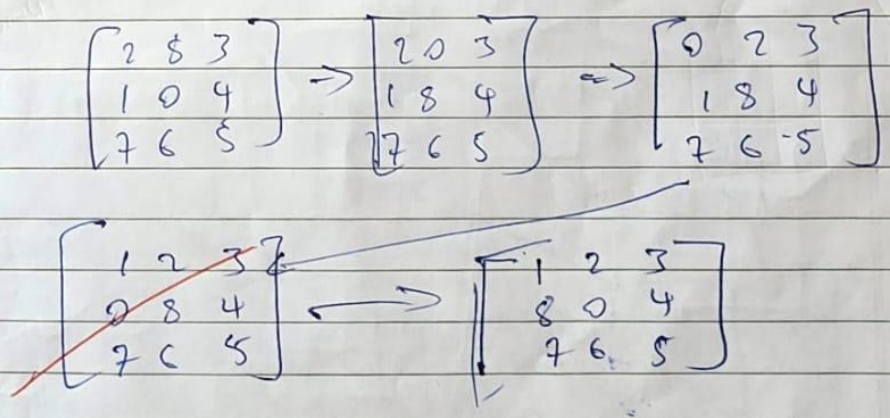
Manhattan Distance

* Define goal state and create a dictionary
1) Create a list containing $f(n)$, $g(n)$ and path with $g(n)$
Define a closed set to track visited states

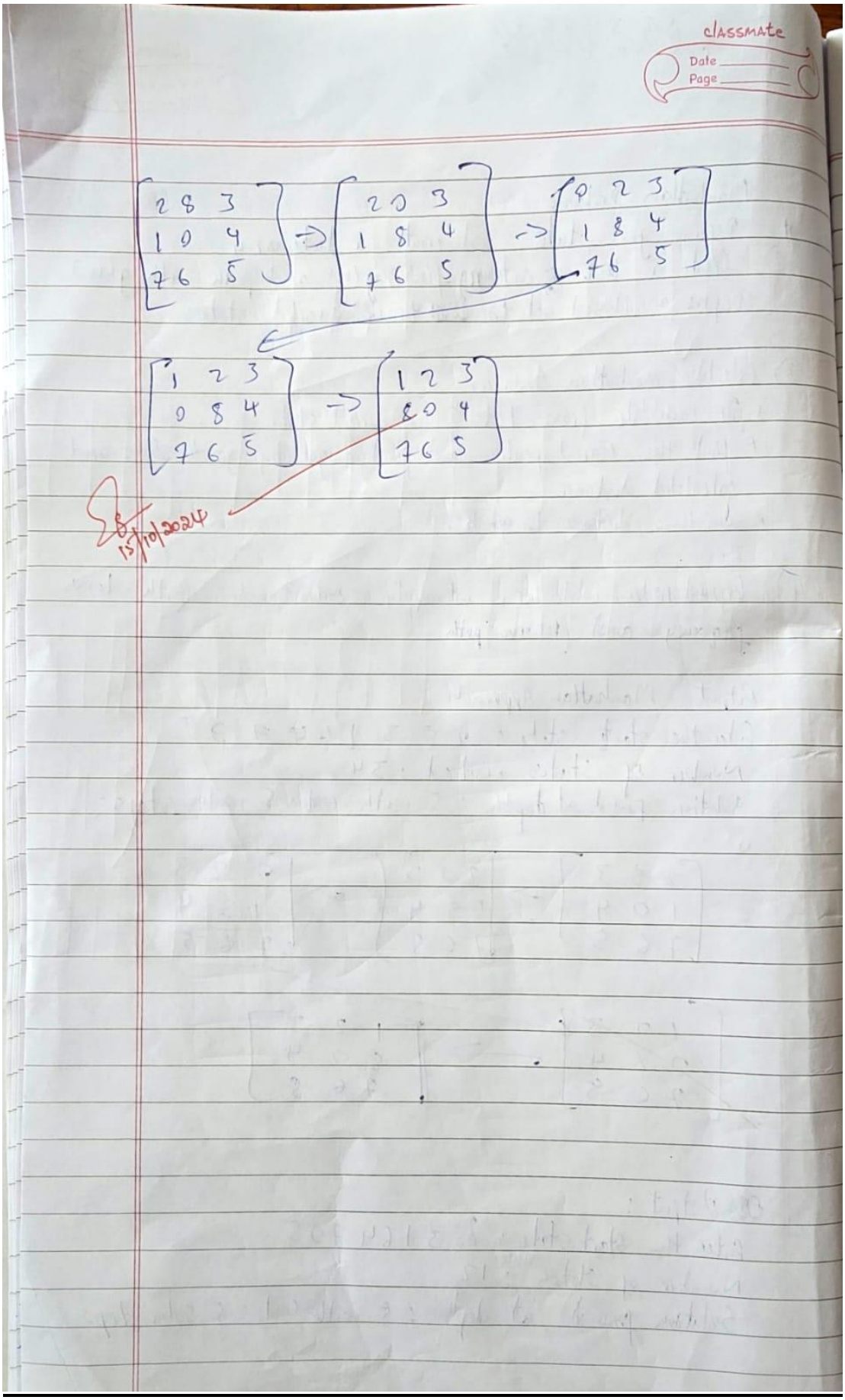
3) Calculate manhattan distances
* for each tile from 1 to 8 in current state
* find this current position and find target by goal state and calculated distance
* sum these distances to get $h(n)$

4) Expand nodes, while list is not empty, extract node with low frequency and return path

Output: Manhattan Approach
Enter the start state: 2 8 3 1 6 4 7 0 5
Number of states visited: 34
Solution found at depth: 5 with cost: 5 soln steps:



Output:
Enter the start state: 2 8 3 1 6 4 7 0 5
Number of states visited: 12
Solution found at depth: 5 with cost: 5 soln steps


classmate
Date
Page
15/10/2024

**Program 3**

Implement A* search algorithm

Code:

Misplaced Tiles

```
def mistil(state, goal):
    count = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != goal[i][j]:
                count += 1
    return count

def findmin(open_list, goal):
    minv = float('inf')
    best_state = None
    for state in open_list:
        h = mistil(state['state'], goal)
        f = state['g'] + h
        if f < minv:
            minv = f
            best_state = state
    open_list.remove(best_state)
    return best_state

def operation(state):
    next_states = []
    blank_pos = find_blank_position(state['state'])
    for move in ['up', 'down', 'left', 'right']:
        new_state = apply_move(state['state'], blank_pos, move)
        if new_state:
            next_states.append({
                'state': new_state,
                'parent': state,
                'move': move,
                'g': state['g'] + 1
            })
    return next_states

def find_blank_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return None

def apply_move(state, blank_pos, move):
```

```
    i, j = blank_pos
    new_state = [row[:] for row in state]
    if move == 'up' and i > 0:
        new_state[i][j], new_state[i - 1][j] = new_state[i - 1][j], new_state[i][j]
    elif move == 'down' and i < 2:
        new_state[i][j], new_state[i + 1][j] = new_state[i + 1][j], new_state[i][j]
    elif move == 'left' and j > 0:
        new_state[i][j], new_state[i][j - 1] = new_state[i][j - 1], new_state[i][j]
    elif move == 'right' and j < 2:
        new_state[i][j], new_state[i][j + 1] = new_state[i][j + 1], new_state[i][j]
    else:
        return None
    return new_state

def print_state(state):
    for row in state:
        print(' '.join(map(str, row)))

initial_state = [[2,8,3], [1,6,4], [7,0,5]]
goal_state = [[1,2,3], [8,0,4], [7,6,5]]
open_list = [{'state': initial_state, 'parent': None, 'move': None, 'g': 0}]
visited_states = []

while open_list:
    best_state = findmin(open_list, goal_state)
    print("Current state:")
    print_state(best_state['state'])
    h = mistil(best_state['state'], goal_state)
    f = best_state['g'] + h
    print(f"g(n): {best_state['g']}, h(n): {h}, f(n): {f}")
    if best_state['move'] is not None:
        print(f"Move: {best_state['move']}")
    print()
    if mistil(best_state['state'], goal_state) == 0:
        goal_state_reached = best_state
        break
    visited_states.append(best_state['state'])
    next_states = operation(best_state)
    for state in next_states:
        if state['state'] not in visited_states:
            open_list.append(state)

moves = []
while goal_state_reached['move'] is not None:
    moves.append(goal_state_reached['move'])
    goal_state_reached = goal_state_reached['parent']
moves.reverse()
```

```
print("\nMoves to reach the goal state:", moves)
print("\nGoal state reached:")
print_state(goal_state)
```

```
Current state:
2 8 3
1 6 4
7 0 5
g(n): 0, h(n): 5, f(n): 5

Current state:
2 8 3
1 0 4
7 6 5
g(n): 1, h(n): 3, f(n): 4
Move: up

Current state:
2 0 3
1 8 4
7 6 5
g(n): 2, h(n): 4, f(n): 6
Move: up

Current state:
2 8 3
0 1 4
7 6 5
g(n): 2, h(n): 4, f(n): 6
Move: left

Current state:
0 2 3
1 8 4
7 6 5
g(n): 3, h(n): 3, f(n): 6
Move: left

Current state:
1 2 3
0 8 4
7 6 5
g(n): 4, h(n): 2, f(n): 6
Move: down

Current state:
1 2 3
8 0 4
7 6 5
g(n): 5, h(n): 0, f(n): 5
Move: right


Moves to reach the goal state: ['up', 'up', 'left', 'down', 'right']

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

Manhattan Distance

```
def manhattan_distance(state, goal):
    distance = 0
    for i in range(3):
        for j in range(3):
            tile = state[i][j]
            if tile != 0:  # Ignore the blank space (0)
                # Find the position of the tile in the goal state
                for r in range(3):
                    for c in range(3):
                        if goal[r][c] == tile:
                            target_row, target_col = r, c
                            break
                # Add the Manhattan distance (absolute difference in rows and columns)
                distance += abs(target_row - i) + abs(target_col - j)
    return distance

def findmin(open_list, goal):
    minv = float('inf')
    best_state = None
    for state in open_list:
        h = manhattan_distance(state['state'], goal)  # Use Manhattan distance here
        f = state['g'] + h
        if f < minv:
            minv = f
            best_state = state
    open_list.remove(best_state)
    return best_state

def operation(state):
    next_states = []
    blank_pos = find_blank_position(state['state'])
    for move in ['up', 'down', 'left', 'right']:
        new_state = apply_move(state['state'], blank_pos, move)
        if new_state:
            next_states.append({
                'state': new_state,
                'parent': state,
                'move': move,
                'g': state['g'] + 1
            })
    return next_states

def find_blank_position(state):
    for i in range(3):
```

```
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return None

def apply_move(state, blank_pos, move):
    i, j = blank_pos
    new_state = [row[:] for row in state]
    if move == 'up' and i > 0:
        new_state[i][j], new_state[i - 1][j] = new_state[i - 1][j], new_state[i][j]
    elif move == 'down' and i < 2:
        new_state[i][j], new_state[i + 1][j] = new_state[i + 1][j], new_state[i][j]
    elif move == 'left' and j > 0:
        new_state[i][j], new_state[i][j - 1] = new_state[i][j - 1], new_state[i][j]
    elif move == 'right' and j < 2:
        new_state[i][j], new_state[i][j + 1] = new_state[i][j + 1], new_state[i][j]
    else:
        return None
    return new_state

def print_state(state):
    for row in state:
        print(' '.join(map(str, row)))

# Initial state and goal state
initial_state = [[2,8,3], [1,6,4], [7,0,5]]
goal_state = [[1,2,3], [8,0,4], [7,6,5]]

# Open list and visited states
open_list = [{'state': initial_state, 'parent': None, 'move': None, 'g': 0}]
visited_states = []

while open_list:
    best_state = findmin(open_list, goal_state)

    print("Current state:")
    print_state(best_state['state'])

    h = manhattan_distance(best_state['state'], goal_state)  # Using Manhattan distance here
    f = best_state['g'] + h
    print(f"g(n): {best_state['g']}, h(n): {h}, f(n): {f}")

    if best_state['move'] is not None:
        print(f"Move: {best_state['move']}")
    print()
    if h == 0:  # Goal is reached if h == 0
        goal_state_reached = best_state
```

```
        break

    visited_states.append(best_state['state'])
    next_states = operation(best_state)

    for state in next_states:
        if state['state'] not in visited_states:
            open_list.append(state)

# Reconstruct the path of moves
moves = []
while goal_state_reached['move'] is not None:
    moves.append(goal_state_reached['move'])
    goal_state_reached = goal_state_reached['parent']
moves.reverse()

print("\nMoves to reach the goal state:", moves)
print("\nGoal state reached:")
print_state(goal_state)
```

```
Current state:
2 8 3
1 6 4
7 0 5
g(n): 0, h(n): 5, f(n): 5

Current state:
2 8 3
1 0 4
7 6 5
g(n): 1, h(n): 4, f(n): 5
Move: up

Current state:
2 0 3
1 8 4
7 6 5
g(n): 2, h(n): 3, f(n): 5
Move: up

Current state:
0 2 3
1 8 4
7 6 5
g(n): 3, h(n): 2, f(n): 5
Move: left

Current state:
1 2 3
0 8 4
7 6 5
g(n): 4, h(n): 1, f(n): 5
Move: down
```

```
Current state:
1 2 3
8 0 4
7 6 5
g(n): 5, h(n): 0, f(n): 5
Move: right


Moves to reach the goal state: ['up', 'up', 'left', 'down', 'right']

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

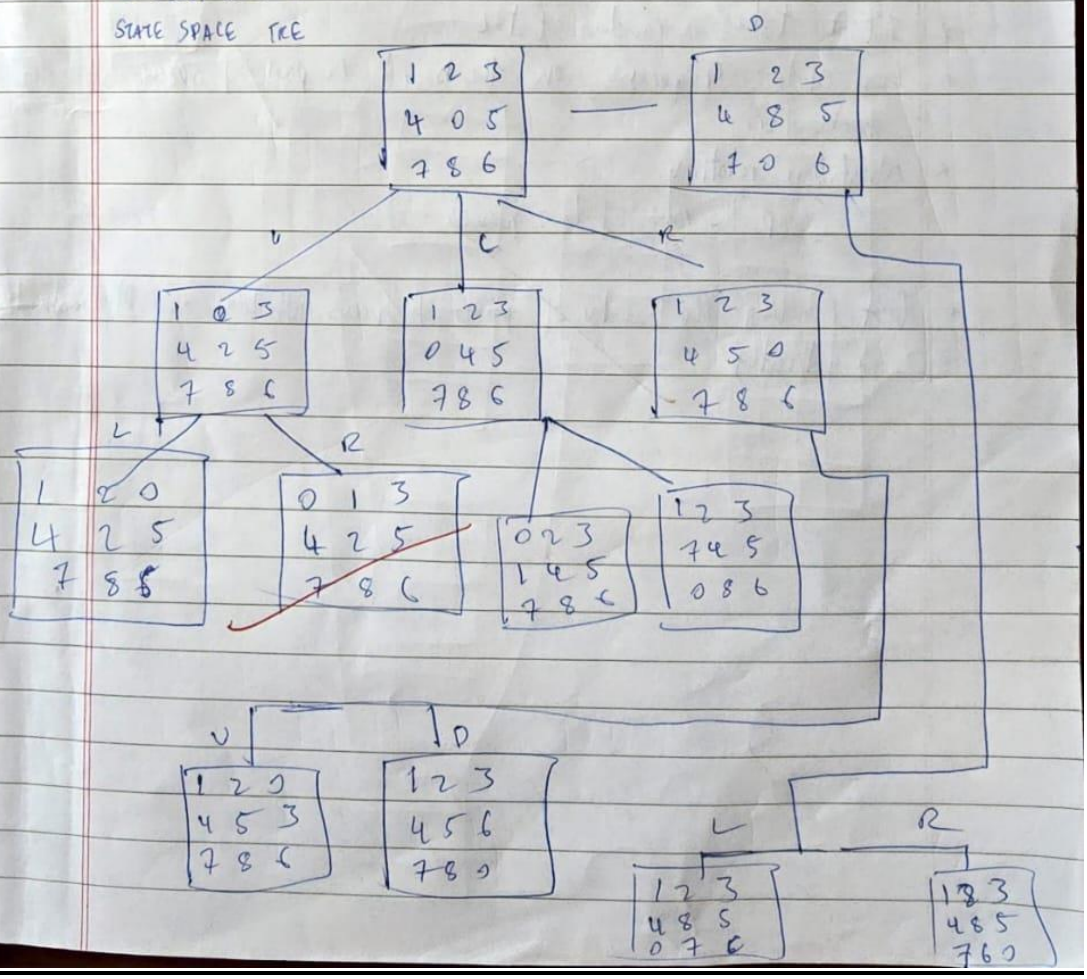
LAB-5

Implement iterative deepening search algorithm

Algorithm :-

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

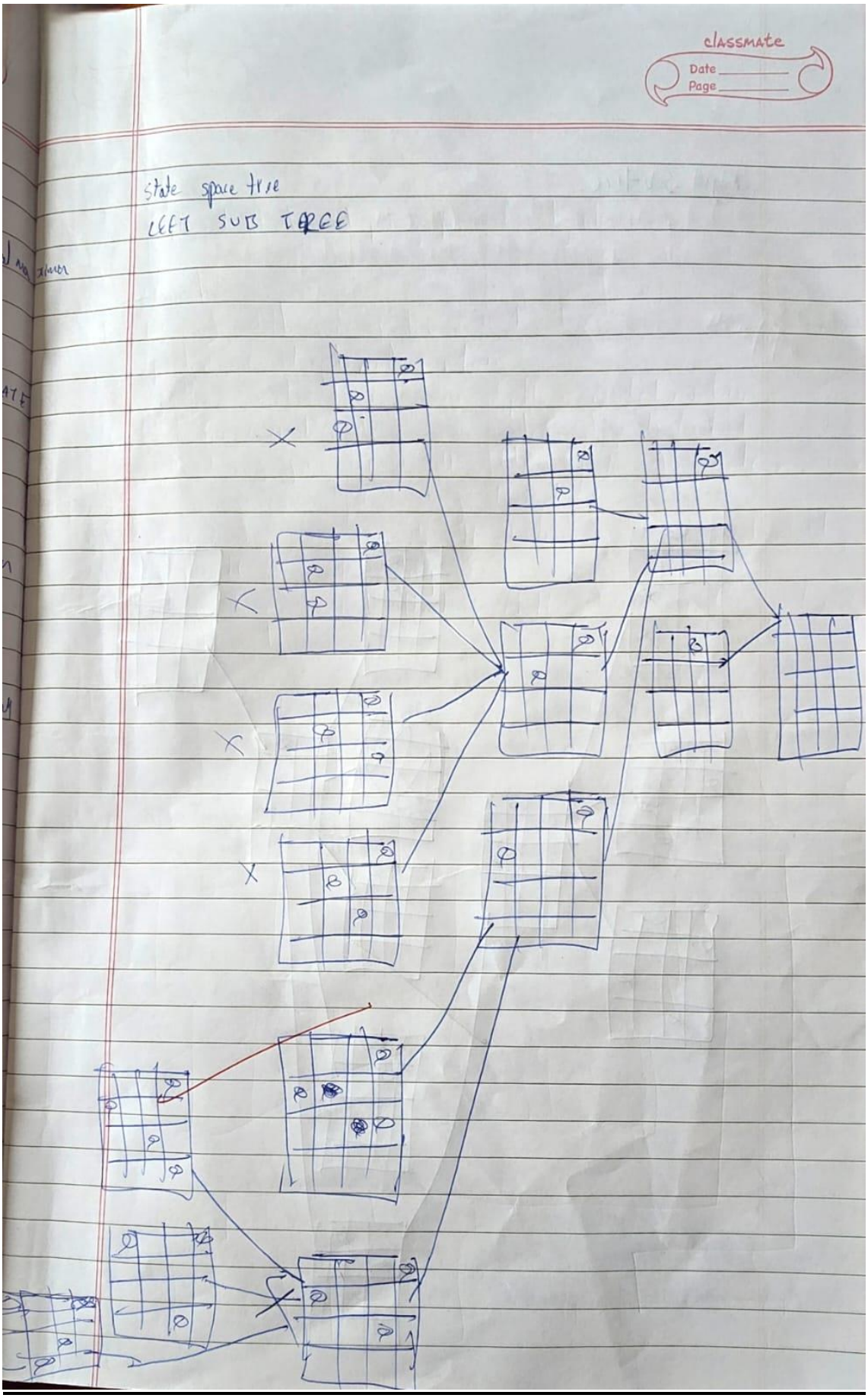
STATE SPACE TREE



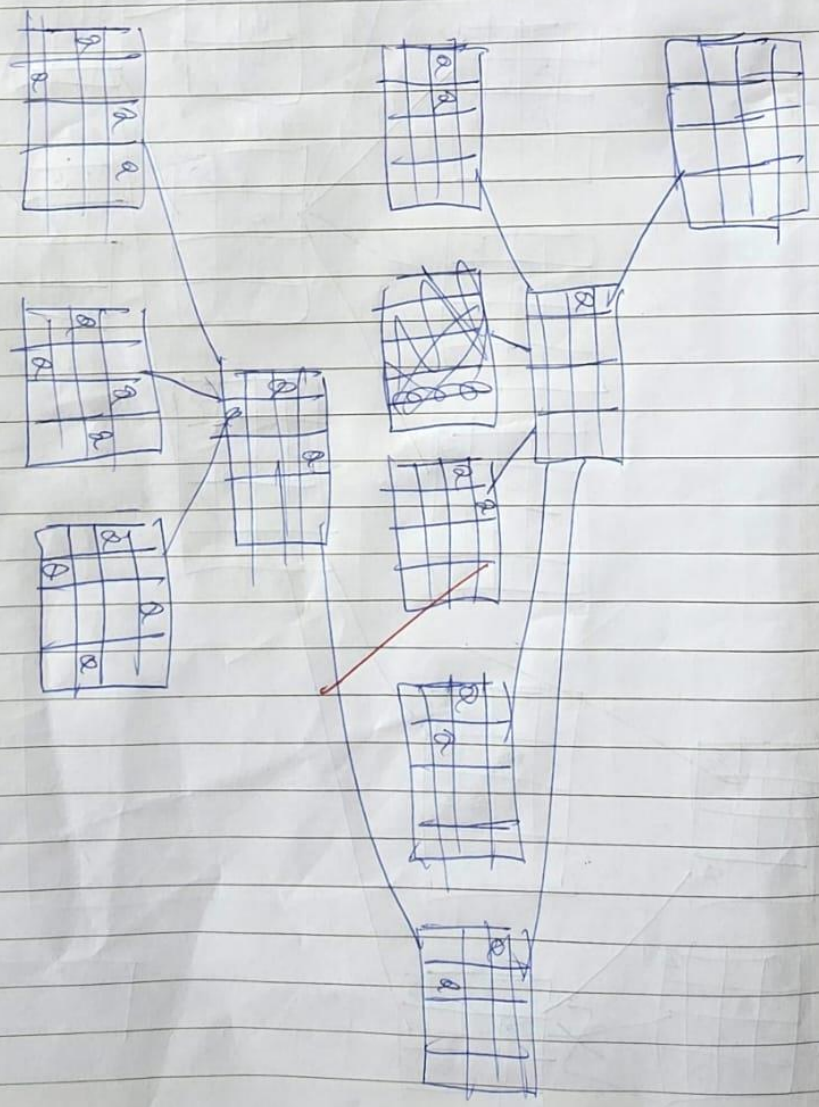
HILL CLIMBING SEARCH ALGORITHM

function HILL-CLIMBING (problem) return a state that is a lo
current ← MAKE-NODE (problem. INITIAL-STATE)
loop do
neighbor ← a highest-valued successor of current
if neighbor.Value ≤ current.Value then return current.St
current ← neighbour

- State 4 queens on the board. One queen per column
- Variables $x_0, x_1, x_2, x_3$ where $x_i$ is the row position of the queen in column i. Assume that there is one queen per colum
- 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 attack each other
- Neighbor relation:
  Swap the row of two queens
- Cost function: The number of pairs of queens attacking each other directly or indirectly


classmate
Date
Page
state space tree
LEFT SUB TREE

Right Sub-tree



**Program 4**

Implement Hill Climbing search algorithm to solve N-Queens problem

Code:

```
import random

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] or abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def hill_climbing(n):
    cost=0
    while True:
        # Initialize a random board
        current_board = list(range(n))
        random.shuffle(current_board)
        current_conflicts = calculate_conflicts(current_board)

        while True:
            # Generate neighbors by moving each queen to a different position
            found_better = False
            for i in range(n):
                for j in range(n):
                    if j != current_board[i]:  # Only consider different positions
                        neighbor_board = list(current_board)
                        neighbor_board[i] = j
                        neighbor_conflicts = calculate_conflicts(neighbor_board)
                        if neighbor_conflicts < current_conflicts:
                            print_board(current_board)
                            print(current_conflicts)
                            print_board(neighbor_board)
                            print(neighbor_conflicts)
                            current_board = neighbor_board
                            current_conflicts = neighbor_conflicts
                            cost+=1
                            found_better = True
                            break
                if found_better:
                    break

            # If no better neighbor found, stop searching
```

```
            if not found_better:
                break

        # If a solution is found (zero conflicts), return the board
        if current_conflicts == 0:
            return current_board, current_conflicts, cost

def print_board(board):
    n = len(board)
    for i in range(n):
        row = ['.'] * n
        row[board[i]] = 'Q'  # Place a queen
        print(' '.join(row))
    print()
print("===============")
# Example Usage
n = 4
solution, conflicts, cost = hill_climbing(n)
print("Final Board Configuration:")
print_board(solution)
print("Number of Cost:", cost)
```

```
==========
Q . . .
. . . Q
. . Q .
. Q . .

4
Q . . .
Q . . .
. . Q .
. Q . .

3
Q . . .
Q . . .
. . Q .
. Q . .

3
. . Q .
Q . . .
. . Q .
. Q . .

2
. . Q .
Q . . .
. . Q .
. Q . .

2
. . . Q
Q . . .
. . Q .
. Q . .

1
Final Board Configuration:
. Q . .
. . . Q
Q . . .
. . Q .
```

LAB-6

Program to Implement Simulated Annealing

Algorithm:

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

inputs problem, a problem

schedule, a mapping from time to "temperature"

current ← MAKE-NODE (problem.INITIALSTATE)

for t = 1 to ∞ do

T ← schedule(t)

if T = 0 then return current

next ← a randomly selected successor of current

ΔE ← next.VALUE − current.VALUE

if ΔE > 0 then current ← next

else current ← next only with probability $e^{\Delta E/T}$

e Termination:

Stop when the schedule completes or when a satisfactory solution is found

function simulated_annealing(N):

Initialize current solution randomly

Current fitness = calculate_fitness(current solution)

Best solution = current solution

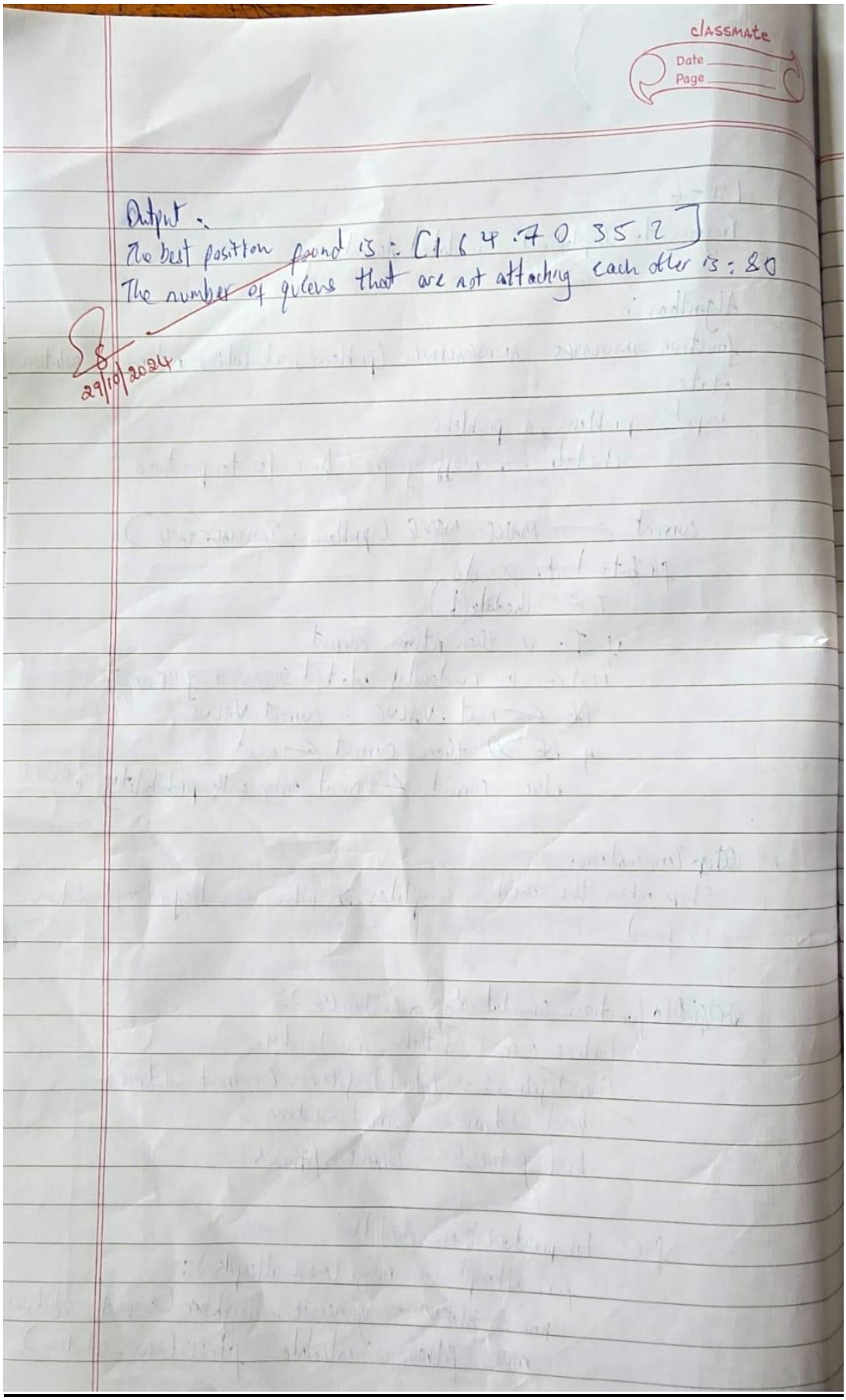
best fitness = current_fitness

for temperature in schedule:

for attempt in range(max attempts):

new_solution = generate_neighbor(current_solution)

new_fitness = calculate_fitness(new_solution)


classmate
Date
Page
Output :
The best position found is : [1 6 4 7 0 3 5 2]
The number of queens that are not attacking each other is : 8
29/10/2024

**Program 5**

Simulated Annealing to Solve 8-Queens problem

Code:

```
import numpy as np
from scipy.optimize import dual_annealing

def queens_max(position):
    # This function calculates the number of pairs of queens that are not attacking each other
    position = np.round(position).astype(int)  # Round and convert to integers for queen positions
    n = len(position)
    queen_not_attacking = 0

    for i in range(n - 1):
        no_attack_on_j = 0
        for j in range(i + 1, n):
            # Check if queens are on the same row or on the same diagonal
            if position[i] != position[j] and abs(position[i] - position[j]) != (j - i):
                no_attack_on_j += 1
        if no_attack_on_j == n - 1 - i:
            queen_not_attacking += 1
    if queen_not_attacking == n - 1:
        queen_not_attacking += 1
    return -queen_not_attacking  # Negative because we want to maximize this value

# Bounds for each queen's position (0 to 7 for an 8x8 chessboard)
bounds = [(0, 8) for _ in range(8)]

# Use dual_annealing for simulated annealing optimization
result = dual_annealing(queens_max, bounds)

# Display the results
best_position = np.round(result.x).astype(int)
best_objective = -result.fun  # Flip sign to get the number of non-attacking queens

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

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

LAB - 7

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

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

ALGORITHM:

function TT entails? (KB, $\alpha$) returns true or false
inputs: KB, the knowledge base, a sentence in propositional logic
$\alpha$, the query, a sentence in propositional logic
symbols ← a list of the proposition symbols in KB and $\alpha$
return TT-CHECK-ALL (KB, $\alpha$, symbols, {})

function TT-CHECK-ALL (KB, $\alpha$, symbols, model) returns true or false
if EMPTY? (symbols) then
if PL-TRUE? (KB, model) then return PL-TRUE? ($\alpha$, model)
else return true // when KB is false always return true
else do
P ← FIRST (symbols)
rest ← REST (symbols)
return (TT-check-ALL (KB, $\alpha$, rest, model ∪ {P = true}))
and
TT-CHECK-ALL (KB, $\alpha$, rest, model ∪ {P = false}))

Q- A ∨ B  KB = (A ∨ C) ∧ (B ∨ ¬C)

| A | B | C | A ∨ C | B ∨ ¬C | KB | Q |
|---|---|---|---|---|---|---|
| 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 |

12/11/2024

Code:

```
from itertools import product

def evaluate(expression, model):
    if isinstance(expression, str):
        return model[expression]
    elif isinstance(expression, tuple):
        op, *args = expression
        if op == 'NOT':
            return not evaluate(args[0], model)
        elif op == 'AND':
            return evaluate(args[0], model) or evaluate(args[1], model)
        elif op == 'OR':
            return evaluate(args[0], model) or evaluate(args[1], model)
    raise ValueError("Unknown expression format")
```

Output:

The KB entails the query.

A B C NOT A ANDS

**Program 6**

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

Code:

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

# Function to evaluate an expression
def evaluate_expression(a, b, c, expression):
    # Use eval() to evaluate the logical expression
    return eval(expression)

# Function to generate the truth table and evaluate a logical expression
def truth_table_and_evaluation(kb, query):
    # All possible combinations of truth values for a, b, and c
    truth_values = [True, False]
    combinations = list(itertools.product(truth_values, repeat=3))

    # Reverse the combinations to start from the bottom (False -> True)
    combinations.reverse()

    # Header for the full truth table
    print(f"{'a':<5}   {'b':<5} {'c':<5} {'KB':<20}{'Query':<20}")

    # Evaluate the expressions for each combination
    for combination in combinations:
        a, b, c = combination

        # Evaluate the knowledge base (KB) and query expressions
        kb_result = evaluate_expression(a, b, c, kb)
        query_result = evaluate_expression(a, b, c, query)

        # Replace True/False with string "True"/"False"
        kb_result_str = "True" if kb_result else "False"
        query_result_str = "True" if query_result else "False"

        # Convert boolean values of a, b, c to "True"/"False"
        a_str = "True" if a else "False"
        b_str = "True" if b else "False"
        c_str = "True" if c else "False"

        # Print the results for the knowledge base and the query
        print(f"{a_str:<5} {b_str:<5} {c_str:<5} {kb_result_str:<20} {query_result_str:<20}")
```

```
    # Additional output for combinations where both KB and query are true
    print("\nCombinations where both KB and Query are True:")
    print(f"{'a':<5}   {'b':<5} {'c':<5} {'KB':<20}{'Query':<20}")

    # Print only the rows where both KB and Query are True
    for combination in combinations:
        a, b, c = combination

        # Evaluate the knowledge base (KB) and query expressions
        kb_result = evaluate_expression(a, b, c, kb)
        query_result = evaluate_expression(a, b, c, query)

        # If both KB and query are True, print the combination
        if kb_result and query_result:
            a_str = "True" if a else "False"
            b_str = "True" if b else "False"
            c_str = "True" if c else "False"
            kb_result_str = "True" if kb_result else "False"
            query_result_str = "True" if query_result else "False"
            print(f"{a_str:<5} {b_str:<5} {c_str:<5} {kb_result_str:<20} {query_result_str:<20}")

# Define the logical expressions as strings
kb = "(a or c) and (b or not c)"  # Knowledge Base
query = "a or b"  # Query to evaluate

# Generate the truth table and evaluate the knowledge base and query
truth_table_and_evaluation(kb, query)
```

```
a       b     c     KB                   Query
False False False False                False
False False True  False                False
False True  False False                True
False True  True  True                 True
True  False False True                 True
True  False True  False                True
True  True  False True                 True
True  True  True  True                 True

Combinations where both KB and Query are True:
a       b     c     KB                   Query
False True  True  True                 True
True  False False True                 True
True  True  False True                 True
True  True  True  True                 True
```

26-11-24

LAB-8

FIRST ORDER COMP.- FORWARD CHAINING

ALGORITHM

function FOL-FC-ASK (KB, α) returns a substitution or fal
inputs:
KB, the knowledge base, a set of first order definite claus
α, the query, an atomic sentence

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

repeat until new is empty
new ← { }
for each rule in KB do
(p1 ∧ .... ∧ pn ⇒ q) ← STANDARDIZE-VARIABLE(
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
add q' to new
φ ← UNIFY(q', α)
if φ is not fail then return φ
add new to KB
return false

Output :

Enter facts
FACTS: American (Robert)
FACT: Enemy (A, America)
FACT: Owns (A, T1)
FACT: Missile (T1)

Enter Rules :
Rule: American(x) ^ Weapon(y) ^ Sells(x,y,
^ Hostile(z) => Criminal(x)
Missile(x) => Weapon(x)
Enemy(x, America) => Hostile(x)
done
The query ('Criminal Robert',) is provable

26-11-24

**Program 7**

Implement unification in first order logic

Code:

```
import re

def occurs_check(var, x):
    """Checks if var occurs in x (to prevent circular substitutions)."""
    if var == x:
        return True
    elif isinstance(x, list):  # If x is a compound expression (like a function or predicate)
        return any(occurs_check(var, xi) for xi in x)
    return False

def unify_var(var, x, subst):
    """Handles unification of a variable with another term."""
    if var in subst:  # If var is already substituted
        return unify(subst[var], x, subst)
    elif isinstance(x, (list, tuple)) and tuple(x) in subst:  # Handle compound expressions
        return unify(var, subst[tuple(x)], subst)
    elif occurs_check(var, x):  # Check for circular references
        return "FAILURE"
    else:
        # Add the substitution to the set (convert list to tuple for hashability)
        subst[var] = tuple(x) if isinstance(x, list) else x
        return subst

def unify(x, y, subst=None):
    """
    Unifies two expressions x and y and returns the substitution set if they can be unified.
    Returns 'FAILURE' if unification is not possible.
    """
    if subst is None:
        subst = {}  # Initialize an empty substitution set

    # Step 1: Handle cases where x or y is a variable or constant
    if x == y:  # If x and y are identical
        return subst
    elif isinstance(x, str) and x.islower():  # If x is a variable
        return unify_var(x, y, subst)
    elif isinstance(y, str) and y.islower():  # If y is a variable
        return unify_var(y, x, subst)
    elif isinstance(x, list) and isinstance(y, list):  # If x and y are compound expressions (lists)
        if len(x) != len(y):  # Step 3: Different number of arguments
            return "FAILURE"
```

```
        # Step 2: Check if the predicate symbols (the first element) match
        if x[0] != y[0]:  # If the predicates/functions are different
            return "FAILURE"

        # Step 5: Recursively unify each argument
        for xi, yi in zip(x[1:], y[1:]):  # Skip the predicate (first element)
            subst = unify(xi, yi, subst)
            if subst == "FAILURE":
                return "FAILURE"
        return subst
    else:  # If x and y are different constants or non-unifiable structures
        return "FAILURE"

def unify_and_check(expr1, expr2):
    """
    Attempts to unify two expressions and returns a tuple:
    (is_unified: bool, substitutions: dict or None)
    """
    result = unify(expr1, expr2)
    if result == "FAILURE":
        return False, None
    return True, result

def display_result(expr1, expr2, is_unified, subst):
    print("Expression 1:", expr1)
    print("Expression 2:", expr2)
    if not is_unified:
        print("Result: Unification Failed")
    else:
        print("Result: Unification Successful")
        print("Substitutions:", {k: list(v) if isinstance(v, tuple) else v for k, v in subst.items()})

def parse_input(input_str):
    """Parses a string input into a structure that can be processed by the unification algorithm."""
    # Remove spaces and handle parentheses
    input_str = input_str.replace(" ", "")

    # Handle compound terms (like p(x, f(y)) -> ['p', 'x', ['f', 'y']])
    def parse_term(term):
        # Handle the compound term
        if '(' in term:
            match = re.match(r'([a-zA-Z0-9_]+)(.*)', term)
            if match:
                predicate = match.group(1)
                arguments_str = match.group(2)
                arguments = [parse_term(arg.strip()) for arg in arguments_str.split(',')]
                return [predicate] + arguments
```

```
        return term

    return parse_term(input_str)

# Main function to interact with the user
def main():
    while True:
        # Get the first and second terms from the user
        expr1_input = input("Enter the first expression (e.g., p(x, f(y))): ")
        expr2_input = input("Enter the second expression (e.g., p(a, f(z))): ")

        # Parse the input strings into the appropriate structures
        expr1 = parse_input(expr1_input)
        expr2 = parse_input(expr2_input)

        # Perform unification
        is_unified, result = unify_and_check(expr1, expr2)

        # Display the results
        display_result(expr1, expr2, is_unified, result)

        # Ask the user if they want to run another test
        another_test = input("Do you want to test another pair of expressions? (yes/no): ").strip().lower()
        if another_test != 'yes':
            break

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

```
Enter the first expression (e.g., p(x, f(y))): p(b,x,f(g(z)))
Enter the second expression (e.g., p(a, f(z))): p(z,f(y),f(y))
Expression 1: ['p', '(b', 'x', ['f', '(g(z)))']]
Expression 2: ['p', '(z', ['f', '(y)'], ['f', '(y))']]
Result: Unification Successful
Substitutions: {'(b': '(z', 'x': ['f', '(y)'], '(g(z)))': '(y))'}
Do you want to test another pair of expressions? (yes/no): yes
Enter the first expression (e.g., p(x, f(y))): p(x,h(y))
Enter the second expression (e.g., p(a, f(z))): p(a,f(z))
Expression 1: ['p', '(x', ['h', '(y))']]
Expression 2: ['p', '(a', ['f', '(z))']]
Result: Unification Failed
Do you want to test another pair of expressions? (yes/no): yes
Enter the first expression (e.g., p(x, f(y))): p(f(a),g(y))
Enter the second expression (e.g., p(a, f(z))): p(x,x)
Expression 1: ['p', '(f(a)', ['g', '(y))']]
Expression 2: ['p', '(x', 'x)']
Result: Unification Successful
Substitutions: {'(f(a)': '(x', 'x)': ['g', '(y))']}
Do you want to test another pair of expressions? (yes/no): no
```

LAB-7

FIRST ORDER LOGIC {Implementing unification in first order logic}

ALGORITHM

STEP 1 : If $\Psi_1$ or $\Psi_2$ is a variable or constant, then:
a) If $\Psi_1$ or $\Psi_2$ are identical, then return NIL
b) Else if $\Psi_1$ is a variable,
a) then if $\Psi_1$ occurs in $\Psi_2$, then return FAILURE
b) Else return $\{\{\Psi_1 / \Psi_2\}\}$.

c) Else if $\Psi_2$ is a variable
a) If $\Psi_2$ occurs in $\Psi_1$ then return FAILURE
b) Else return $\{(\Psi_1 / \Psi_2)\}$

d) Else return FAILURE

STEP 2 : If the initial Predicate symbol in $\Psi_1$ and $\Psi_2$ are not same, then return FAILURE.

STEP 3 : If $\Psi_1$ and $\Psi_2$ have a different number of arguments, then return FAILURE

Step 4 : Set Substitution set (SUBST) to NIL

step 5 : for i = 1 to the number of elements in $\Psi_1$,
a) call Unify function with the $i^{th}$ element of $\Psi_1$ and $i^{th}$ element of $\Psi_2$ and put the result into S

b) If S = failure then returns failure
c) If S ≠ NIL then do,

a. Apply S to the remainder of both L1 and L2
b. SUBST = Append (S, SUBST)

Step 6: Return SUBST

$\Psi_1 = p(b, X, f(g(z)))$
$\Psi_2 = p(z, f(y), f(y))$

{'b': 'z', 'f(y)': 'x', 'f(g(z))': 'f(y)'}

$\Psi_1 = P(f(a), g(y))$
$\Psi_2 = P(x, x)$

Unification failed
substitute (f(a) / x)

P(f(a), g(y)) and P(f(a), f(a))
substitute (f(a) / g(y))

=

19/11/2024

**Program 8**

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

Code:

```
# Define the knowledge base (KB) as a set of facts
KB = set()

# Premises based on the provided FOL problem
KB.add('American(Robert)')
KB.add('Enemy(America, A)')
KB.add('Missile(T1)')
KB.add('Owns(A, T1)')

# Define inference rules
def modus_ponens(fact1, fact2, conclusion):
    """ Apply modus ponens inference rule: if fact1 and fact2 are true, then conclude conclusion """
    if fact1 in KB and fact2 in KB:
        KB.add(conclusion)
        print(f"Inferred: {conclusion}")

def forward_chaining():
    """ Perform forward chaining to infer new facts until no more inferences can be made """
    # 1. Apply: Missile(x) → Weapon(x)
    if 'Missile(T1)' in KB:
        KB.add('Weapon(T1)')
        print(f"Inferred: Weapon(T1)")

    # 2. Apply: Sells(Robert, T1, A) from Owns(A, T1) and Weapon(T1)
    if 'Owns(A, T1)' in KB and 'Weapon(T1)' in KB:
        KB.add('Sells(Robert, T1, A)')
        print(f"Inferred: Sells(Robert, T1, A)")

    # 3. Apply: Hostile(A) from Enemy(A, America)
    if 'Enemy(America, A)' in KB:
        KB.add('Hostile(A)')
        print(f"Inferred: Hostile(A)")

    # 4. Now, check if the goal is reached (i.e., if 'Criminal(Robert)' can be inferred)
    if 'American(Robert)' in KB and 'Weapon(T1)' in KB and 'Sells(Robert, T1, A)' in KB and
'Hostile(A)' in KB:
        KB.add('Criminal(Robert)')
        print("Inferred: Criminal(Robert)")

    # Check if we've reached our goal
    if 'Criminal(Robert)' in KB:
```

```
        print("Robert is a criminal!")
    else:
        print("No more inferences can be made.")

# Run forward chaining to attempt to derive the conclusion
forward_chaining()
```

```
Inferred: Weapon(T1)
Inferred: Sells(Robert, T1, A)
Inferred: Hostile(A)
Inferred: Criminal(Robert)
Robert is a criminal!
```

26-11-24

LAB-9

Convert FOL to resolution

Algorithm:

1) Convert all sentences to CNF

2) Negate conclusion S and convert result to CNF

3) add negated conclusions S to the premorses clauses

4) Repeat until contradiction or no progress is made

a) Select two classes (call them parent classes)

b) Resolve them together, performing all required unification

c) If resolvent is the empty class, a contradiction has been found (i.e, S follows from the premises

d) If not add resolvent to the premises

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

Output:

Does John like peanuts ? Yes

Code:

Define the knowledge base (KB)

```
KB = {
    "food(Apple)": True
    "food(vegetables)": True
    "eats(Anil, Peanuts)": True
    "alive(Anil)": True,
    "likes(John, X)": "food(x)": # John likes all food
    "food(X)": "eats(Y, X) and not killed(Y)"
    "eats(Harry, X)": "eats(Anil, X)",
    "alive(X)": "not killed(X)",
    "not killed(X)": "alive(X)",
}
```

# Function to evaluate

```
def resolve(predicate)
    if predicate in KB and is instance(KB[predicate], bool):
        return KB[predicate]
```

```
if "(" in predicate:
    func, args = predicate.split("(")
    args = args.strip(")").split(",")
```

```
if func == "likes" and args[0] == 'John' and
    args[1] == "Peanuts":
        return resolve("food(Peanuts)")
```

```
query = "likes(John, Peanuts)"
result = resolve(query)
```

# print(f"Does John like peanuts? {'Yes' if result else 'No'}")

**Program 9**

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

Code:

```
# Define the knowledge base (KB)
KB = {
    "food(Apple)": True,
    "food(vegetables)": True,
    "eats(Anil, Peanuts)": True,
    "alive(Anil)": True,
    "likes(John, X)": "food(X)",  # Rule: John likes all food
    "food(X)": "eats(Y, X) and not killed(Y)",  # Rule: Anything eaten and not killed is food
    "eats(Harry, X)": "eats(Anil, X)",  # Rule: Harry eats what Anil eats
    "alive(X)": "not killed(X)",  # Rule: Alive implies not killed
    "not killed(X)": "alive(X)",  # Rule: Not killed implies alive
}

# Function to evaluate if a predicate is true based on the KB
def resolve(predicate):
    # If it's a direct fact in KB
    if predicate in KB and isinstance(KB[predicate], bool):
        return KB[predicate]

    # If it's a derived rule
    if predicate in KB:
        rule = KB[predicate]
        if " and " in rule:  # Handle conjunction
            sub_preds = rule.split(" and ")
            return all(resolve(sub.strip()) for sub in sub_preds)
        elif " or " in rule:  # Handle disjunction
            sub_preds = rule.split(" or ")
            return any(resolve(sub.strip()) for sub in sub_preds)
        elif "not " in rule:  # Handle negation
            sub_pred = rule[4:]  # Remove "not "
            return not resolve(sub_pred.strip())
        else:  # Handle single predicate
            return resolve(rule.strip())

    # If the predicate is a specific query (e.g., likes(John, Peanuts))
    if "(" in predicate:
        func, args = predicate.split("(")
        args = args.strip(")").split(", ")
        if func == "food" and args[0] == "Peanuts":
            return resolve("eats(Anil, Peanuts)") and not resolve("killed(Anil)")
        if func == "likes" and args[0] == "John" and args[1] == "Peanuts":
```

```
            return resolve("food(Peanuts)")

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

# Query to prove: John likes Peanuts
query = "likes(John, Peanuts)"
result = resolve(query)

# Print the result
print(f"Does John like peanuts? {'Yes' if result else 'No'}")
```

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

LAB - 10

Alpha Beta Pruning Algorithm

Alpha ($\alpha$) - Beta ($\beta$) proposes to the final optimal path without looking at every node in the game tree

Max contains Alpha ($\alpha$) and Min contains Beta ($\beta$) bound during the calculations

- In both MIN and MAX Node, we return when $\alpha \leq \beta$ which compares with its parent node only
- Both minimax and alpha ($\alpha$) - Beta ($\beta$) cut-off same path
- Alpha ($\alpha$) - Beta ($\beta$) gives optimal solutions as it takes less time to get the value from the root node.

Output:

Enter numbers for the game tree (space - generated):

10 9 14 18 5 4 50 3

Final Result of Alpha beta Pruning : 50

3/12/2024

LAB-7

FIRST ORDER LOGIC {Implementing unification in first order logic}

ALGORITHM

STEP 1 : If $\Psi_1$ or $\Psi_2$ is a variable or constant, then:
a) If $\Psi_1$ or $\Psi_2$ are identical, then return NIL
b) Else if $\Psi_1$ is a variable,
a) then if $\Psi_1$ occurs in $\Psi_2$, then return FAILURE
b) Else return $\{\{\Psi_1 / \Psi_2\}\}$.

c) Else if $\Psi_2$ is a variable
a) If $\Psi_2$ occurs in $\Psi_1$ then return FAILURE
b) Else return $\{(\Psi_1 / \Psi_2)\}$

d) Else return FAILURE

STEP 2 : If the initial Predicate symbol in $\Psi_1$ and $\Psi_2$ are not same, then return FAILURE.

STEP 3 : If $\Psi_1$ and $\Psi_2$ have a different number of arguments, then return FAILURE

Step 4 : Set Substitution set (SUBST) to NIL

step 5 : for i = 1 to the number of elements in $\Psi_1$,
a) call Unify function with the $i^{th}$ element of $\Psi_1$ and $i^{th}$ element of $\Psi_2$ and put the result into S

b) If S = failure then returns failure
c) If S ≠ NIL then do,

**Program 10**

Implement Alpha-Beta Pruning.

Code:

```
# Alpha-Beta Pruning Implementation
def alpha_beta_pruning(node, alpha, beta, maximizing_player):
    # Base case: If it's a leaf node, return its value (simulating evaluation of the node)
    if type(node) is int:
        return node

    # If not a leaf node, explore the children
    if maximizing_player:
        max_eval = -float('inf')
        for child in node:  # Iterate over children of the maximizer node
            eval = alpha_beta_pruning(child, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)  # Maximize alpha
            if beta <= alpha:  # Prune the branch
                break
        return max_eval
    else:
        min_eval = float('inf')
        for child in node:  # Iterate over children of the minimizer node
            eval = alpha_beta_pruning(child, alpha, beta, True)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)  # Minimize beta
            if beta <= alpha:  # Prune the branch
                break
        return min_eval

# Function to build the tree from a list of numbers
def build_tree(numbers):
    # We need to build a tree with alternating levels of maximizers and minimizers
    # Start from the leaf nodes and work up
    current_level = [[n] for n in numbers]

    while len(current_level) > 1:
        next_level = []
        for i in range(0, len(current_level), 2):
            if i + 1 < len(current_level):
                next_level.append(current_level[i] + current_level[i + 1])  # Combine two nodes
            else:
                next_level.append(current_level[i])  # Odd number of elements, just carry forward
        current_level = next_level

    return current_level[0]  # Return the root node, which is a maximizer
```

```
# Main function to run alpha-beta pruning
def main():
    # Input: User provides a list of numbers
    numbers = list(map(int, input("Enter numbers for the game tree (space-separated): ").split()))

    # Build the tree with the given numbers
    tree = build_tree(numbers)

    # Parameters: Tree, initial alpha, beta, and the root node is a maximizing player
    alpha = -float('inf')
    beta = float('inf')
    maximizing_player = True  # The root node is a maximizing player

    # Perform alpha-beta pruning and get the final result
    result = alpha_beta_pruning(tree, alpha, beta, maximizing_player)

    print("Final Result of Alpha-Beta Pruning:", result)

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

```
Enter numbers for the game tree (space-separated): 10 9 14 18 5 4 50 3
Final Result of Alpha-Beta Pruning: 50
```