ARTIFICIAL INTELLIGENCE

DIGITAL ASSIGNMENT-2

22BCE3409

V HEMANJALI

Objective 1: To solve the 8 Puzzle problem using the Hill Climbing algorithm in Python, with heuristic guidelines and analyze its limitations, including local maxima and plateaus.

Question: Implement the Hill Climbing algorithm in Python to solve the 8 Puzzle problem, starting from a given initial configuration and aiming to reach a specified goal state. Use a heuristic function, either the Manhattan distance or the number of misplaced tiles, to guide the search process. Test your implementation with different initial configurations, document the steps taken to reach the solution, and analyze cases where the algorithm fails due to local maxima or plateaus. Discuss any observed limitations of the Hill Climbing approach in solving the 8 Puzzle problem

CODE:

```
import numpy as np

class PuzzleSolver:
    def __init__(self, start_state):
        self.start_state = start_state
        self.target_state = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 0]])
        self.current_state = start_state

    def calculate_heuristic(self, state):
        distance = 0
        for i in range(1, 9):
            target_pos = np.argwhere(self.target_state == i)[0]
            curr_pos = np.argwhere(state == i)[0]
            distance += abs(target_pos[0] - curr_pos[0]) + abs(target_pos[1] -
curr_pos[1])
        return distance

    def get_adjacent_states(self, state):
        adjacent_states = []
        empty_pos = np.argwhere(state == 0)[0]
        x, y = empty_pos

        moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for dx, dy in moves:
            nx, ny = x + dx, y + dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                new_state = state.copy()
                new_state[x, y], new_state[nx, ny] = new_state[nx, ny], new_state[x, y]
```

```
                adjacent_states.append(new_state)
        return adjacent_states

    def solve_puzzle(self):
        self.current_state = self.start_state
        current_heuristic = self.calculate_heuristic(self.current_state)

        while True:
            neighbors = self.get_adjacent_states(self.current_state)
            next_state = None
            next_heuristic = current_heuristic

            for neighbor in neighbors:
                h = self.calculate_heuristic(neighbor)
                if h < next_heuristic:
                    next_heuristic = h
                    next_state = neighbor

            if next_state is None or next_heuristic >= current_heuristic:
                break

            self.current_state = next_state
            current_heuristic = next_heuristic

        return self.current_state, current_heuristic

start_state = np.array([[1, 2, 3], [4, 0, 6], [7, 5, 8]])
solver = PuzzleSolver(start_state)
final_state, final_heuristic = solver.solve_puzzle()

print("Final State:\n", final_state)
print("Final Heuristic Value:", final_heuristic)
```

OUTPUT:

```
PS C:\Users\asus\Workspace> & C:/Users/asus/AppData/Local/Microsoft/WindowsApps/python3.13.exe c:/Users/asus/Workspace/q1.py
Final State:
 [[1 2 3]
 [4 5 6]
 [7 8 0]]
Final Heuristic Value: 0
```

Objective 2 : To create Prolog rules for determining student eligibility for scholarships and exam permissions based on attendance, integrating these rules with a REST API and web app for querying eligibility and debar status.

Question: Create a system in Prolog to determine student eligibility for scholarships and exam permissions based on attendance data stored in a CSV file. Write Prolog rules to evaluate whether a student qualifies for a scholarship or is permitted for exams, using specified attendance thresholds. Load the CSV data into Prolog, define

the rules, and expose these eligibility checks through a REST API that can be accessed by a web app. Develop a simple web interface that allows users to input a student ID and check their eligibility status. Additionally, write a half-page comparison on the differences between SQL and Prolog for querying, focusing on how each handles data retrieval, logic-based conditions, and use case suitability.

1. CSV data sample (Student_ID , Attendance_percentage , CGPA)

2. Load data in Proglog(:- use_module(library(csv)).

csv_read_file("data.csv", Rows, [functor(student), arity(4)]), mpalist(assert, Rows).

3. Define rules in Prolog

Eligible_for_Scholarship(Student_ID) :-

student(Student_ID,_, Attendance_percentage, CGPA),

Attendance_Percentage>=75, CGPA >=9.0.

Permitted_for_exam(Student_ID) :-

student(Student_ID, _, Attendance, _), Attendance >= 75.

4. REST_API with HTTP Library

5. Web App to call the API

data.csv

```
Student_ID,Name,Attendance_Percentage,CGPA
101,Arjun Sharma,85,9.2
102,Aditi Verma,74,8.6
103,Vikram Iyer,90,9.1
104,Priya Nair,65,7.5
105,Rahul Gupta,78,8.8
106,Neha Singh,82,9.0
107,Siddharth Rao,70,8.1
108,Ishita Menon,92,9.5
109,Abhishek Mehta,87,9.3
110,Ananya Kapoor,66,7.8
111,Rohan Joshi,88,9.7
112,Kavya Mishra,60,6.9
113,Aryan Patil,80,8.9
114,Simran Arora,79,8.4
115,Karan Malhotra,73,8.2
116,Nidhi Das,85,9.4
117,Varun Chatterjee,77,8.5
118,Tanvi Roy,81,8.8
119,Aditya Bhatt,89,9.6
120,Meera Pillai,75,8.7
121,Sahil Deshmukh,82,9.2
122,Anjali Kulkarni,58,6.5
123,Yash Pandey,92,9.8
124,Ritika Sen,71,8.3
125,Arya Chakraborty,84,8.6
126,Devika Ghosh,90,9.4
127,Nikhil Reddy,68,7.3
128,Pooja Joshi,74,8.1
129,Rajat Kumar,88,9.0
130,Ira Agrawal,91,9.5
```

Prolog File- **eligibility_checker.pl**

```
:- use_module(library(csv)).
:- use_module(library(http/http_dispatch)).
:- use_module(library(http/http_parameters)).
:- use_module(library(http/http_json)).
:- use_module(library(http/http_server)).

% Helper: Add CORS headers to response
add_cors_headers :-
    format('Access-Control-Allow-Origin: *~n'),
    format('Access-Control-Allow-Methods: GET, POST, OPTIONS~n'),
    format('Access-Control-Allow-Headers: Content-Type~n').


% Load data from CSV and assert facts
load_csv_data :-
    csv_read_file("data.csv", Rows, [functor(student), arity(4)]),
    maplist(assert, Rows).

:- initialization(load_csv_data).

% Rules for scholarship and exam permission
eligible_for_scholarship(Student_ID) :-
    student(Student_ID, _, Attendance_Percentage, CGPA),
    Attendance_Percentage >= 75,
    CGPA >= 9.0.

permitted_for_exam(Student_ID) :-
    student(Student_ID, _, Attendance_Percentage, _),
    Attendance_Percentage >= 75.

% REST API Endpoints
:- http_handler('/check_scholarship', handle_scholarship, []).
:- http_handler('/check_exam_permission', handle_exam_permission, []).

% Start server
:- initialization(http_server(http_dispatch, [port(8080)])).

% Handle scholarship check
handle_scholarship(Request) :-
    add_cors_headers,
    http_parameters(Request, [id(Student_ID, [integer])]),
    ( eligible_for_scholarship(Student_ID)
    -> Reply = json([status="Eligible"])
    ;  Reply = json([status="Not Eligible"])
    ),
    reply_json(Reply).

% Handle exam permission check
handle_exam_permission(Request) :-
    add_cors_headers,
```

```
    http_parameters(Request, [id(Student_ID, [integer])]),
    ( permitted_for_exam(Student_ID)
    -> Reply = json([status="Permitted"])
    ;  Reply = json([status="Not Permitted"])
    ),
    reply_json(Reply).
```

Index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>VIT Portal</title>
    <style>
        /* General body styling */
        body {
            font-family: 'Arial', sans-serif;
            margin: 0;
            padding: 0;
            background: #f4f4f4;
            color: #333;
        }

        /* Navigation bar styling */
        .navbar {
            background-color: #6a11cb;
            color: #ffffff;
            padding: 10px 20px;
            display: flex;
            justify-content: space-between;
            align-items: center;
            box-shadow: 0px 4px 6px rgba(0, 0, 0, 0.2);
        }

        .navbar h1 {
            margin: 0;
            font-size: 24px;
        }

        /* Main container */
        .container {
            background: #ffffff;
            border-radius: 10px;
            box-shadow: 0px 4px 10px rgba(0, 0, 0, 0.1);
            width: 400px;
```

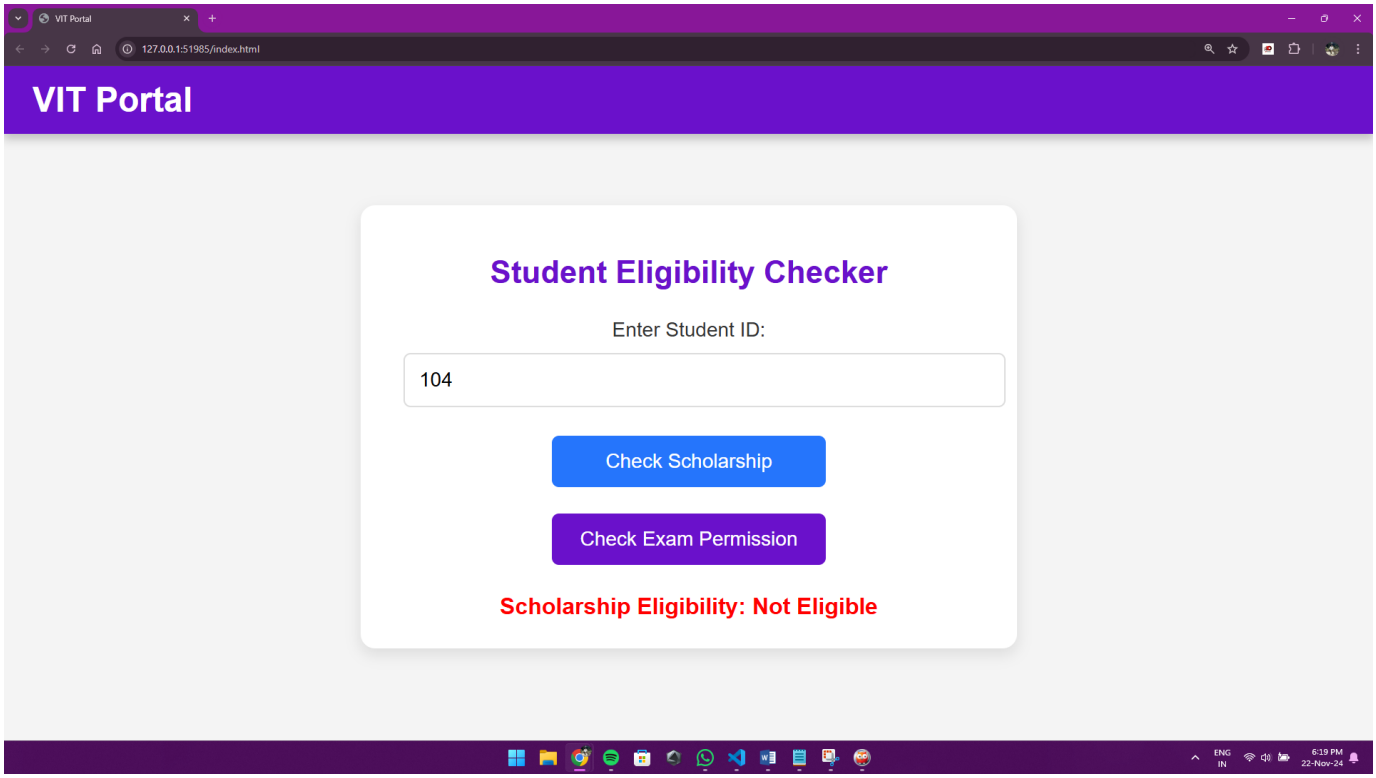
```
        margin: 50px auto;
        padding: 20px 30px;
        text-align: center;
    }

    /* Heading */
    .container h1 {
        font-size: 22px;
        margin-bottom: 20px;
        color: #6a11cb;
    }

    /* Label and input styles */
    .container label {
        display: block;
        font-size: 14px;
        margin-bottom: 8px;
    }

    .container input {
        width: 100%;
        padding: 10px;
        font-size: 14px;
        border: 1px solid #ddd;
        border-radius: 5px;
        margin-bottom: 20px;
        outline: none;
    }

    /* Button styles */
    .container button {
        width: 48%;
        padding: 10px;
        font-size: 14px;
        background: #6a11cb;
        color: #ffffff;
        border: none;
        border-radius: 5px;
        cursor: pointer;
        transition: all 0.3s ease;
        margin: 0 1%;
    }

    .container button:hover {
        background: #2575fc;
    }

    /* Result div */
    .result {
        margin-top: 20px;
        font-size: 16px;
        font-weight: bold;
    }
```

```
        /* Color-coded statuses */
        .success {
            color: green;
        }

        .error {
            color: red;
        }
    </style>
    <script>
        async function checkExamPerm() {
            const studentId = document.getElementById("student_id").value;
            const endpoint = '/check_exam_permission';
            const resultDiv = document.getElementById("result");
            try {
                const response = await
fetch(`http://localhost:8080${endpoint}?id=${studentId}`);
                const data = await response.json();
                resultDiv.textContent = `Exam Permission: ${data.status}`;
                resultDiv.className = data.status === "Permitted" ? "result success" :
"result error";
            } catch (error) {
                resultDiv.textContent = 'Error fetching exam permission data. Please try
again later.';
                resultDiv.className = "result error";
                console.error(error);
            }
        }

        async function checkScholarship() {
            const studentId = document.getElementById("student_id").value;
            const endpoint = '/check_scholarship';
            const resultDiv = document.getElementById("result");
            try {
                const response = await
fetch(`http://localhost:8080${endpoint}?id=${studentId}`);
                const data = await response.json();
                resultDiv.textContent = `Scholarship Eligibility: ${data.status}`;
                resultDiv.className = data.status === "Eligible" ? "result success" :
"result error";
            } catch (error) {
                resultDiv.textContent = 'Error fetching scholarship data. Please try
again later.';
                resultDiv.className = "result error";
                console.error(error);
            }
        }
    </script>
</head>
<body>
    <!-- Navigation bar -->
    <div class="navbar">
```

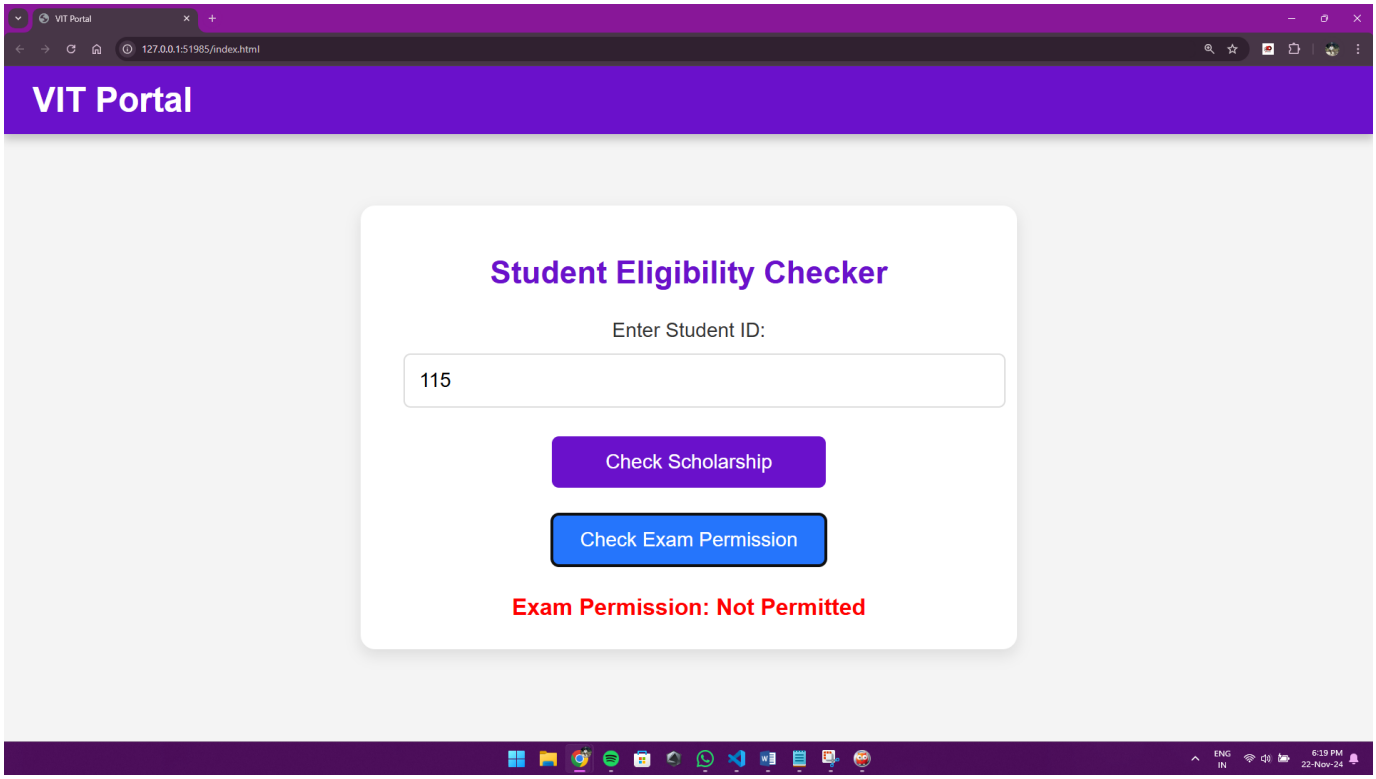
```
        <h1>VIT Portal</h1>
    </div>

    <!-- Main container -->
    <div class="container">
        <h1>Student Eligibility Checker</h1>
        <label for="student_id">Enter Student ID:</label>
        <input type="number" id="student_id" placeholder="Enter your Student ID"
min="101" max="130">
        <div>
            <button onclick="checkScholarship()">Check Scholarship</button>
            <br><br>
            <button onclick="checkExamPerm()">Check Exam Permission</button>
        </div>
        <div class="result" id="result"></div>
    </div>
</body>
</html>
```

OUTPUT:

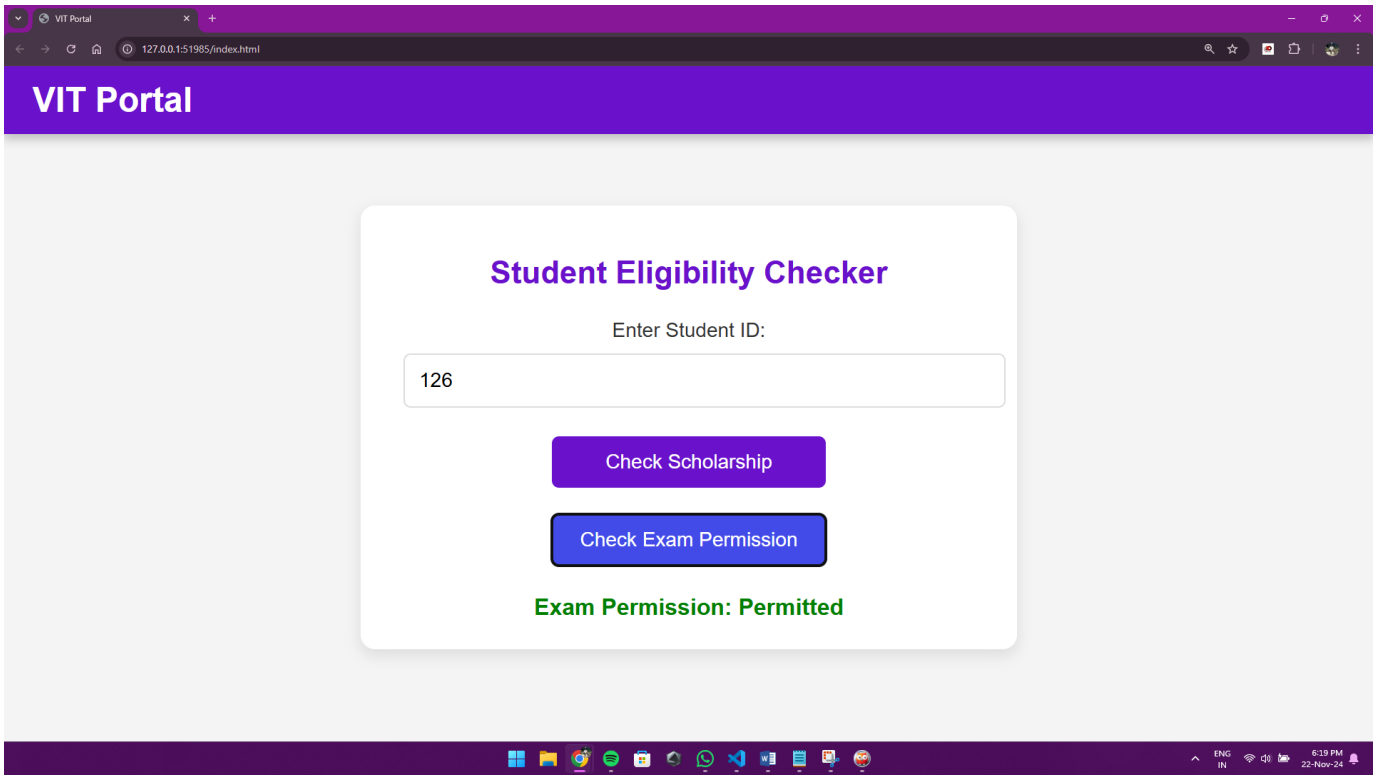
104,Priya Nair,65,7.5



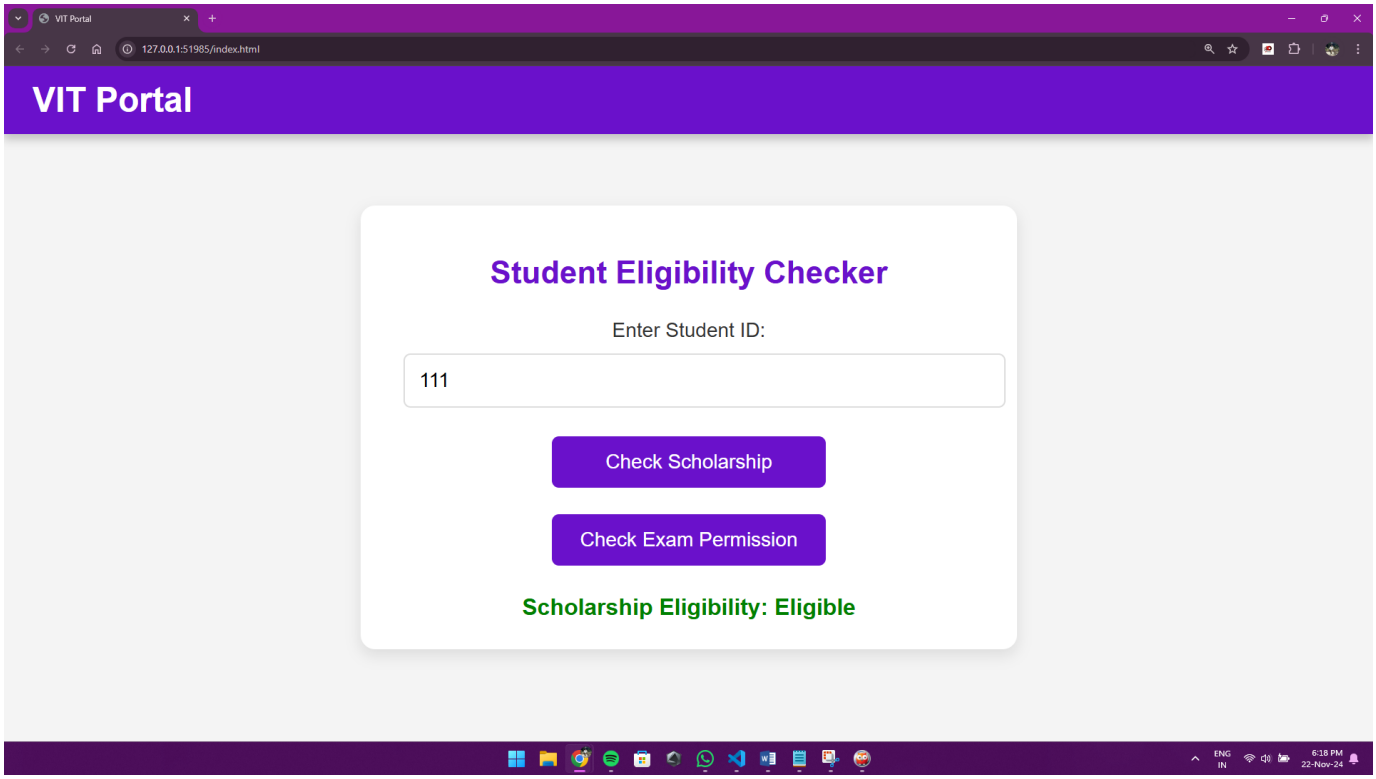
115,Karan Malhotra,73,8.2



126,Devika Ghosh,90,9.4



111,Rohan Joshi,88,9.7



SQL (Structured Query Language) and Prolog (Programming in Logic) are both tools designed to retrieve and manipulate data, but they operate under vastly different paradigms. SQL is primarily focused on interacting with structured, relational databases, whereas Prolog is rooted in logic programming and reasoning. This distinction influences how each handles data retrieval, logical conditions, and their applicability in different use cases.

---

1. Approach to Data Retrieval

SQL operates within a relational database framework, where data is organized into tables with predefined schemas. It adopts a declarative query style, meaning the user specifies *what* data they want, and the database management system determines *how* to retrieve it. For instance, a typical SQL query might look like this:
SELECT * FROM Students WHERE CGPA > 8.0;
This command fetches all students with a CGPA higher than 8. SQL emphasizes efficiency and scalability, with queries often involving multiple tables, filtering conditions, or aggregations.

Prolog, in contrast, focuses on data representation through facts and rules. It retrieves information using logical inference and pattern matching. A query in Prolog might take the form:
student(ID, Name, Attendance, CGPA), CGPA > 8.0.
Here, Prolog evaluates the logical relationships and conditions specified in the query rather than simply fetching data from a structured table. Instead of a tabular output, Prolog provides results as logical conclusions, which can be highly dynamic based on the rules defined in the system.

---

2. Handling Logic-Based Conditions

SQL implements logic through conditional clauses like WHERE, AND, OR, and NOT. While these operators are effective for standard data filtering, they are limited to comparisons based on relational algebra. Complex conditions often require techniques like nested subqueries, joins, or Common Table Expressions (CTEs). For example:

sql
Copy code
```
SELECT Name
FROM Students
WHERE CGPA > 9.0
AND Attendance > 75;
```

This query retrieves the names of students meeting both criteria but requires all conditions to be explicitly stated within the query.

Prolog, on the other hand, is built on the principles of first-order predicate logic. Logical conditions are expressed using rules, which can recursively evaluate relationships and infer conclusions. For example, a Prolog rule for scholarship eligibility might look like:

prolog
Copy code
```
eligible(ID) :- attendance(ID, A), A >= 75, cgpa(ID, C), C >= 9.0.
```

This rule defines eligibility in a declarative way, enabling Prolog to infer whether a student meets the criteria without the need for direct data manipulation. Prolog excels at evaluating complex, hierarchical, or recursive conditions that are challenging to express in SQL.

---

3. Use Case Suitability

SQL is the industry standard for applications requiring reliable and efficient data storage, retrieval, and analysis. It is ideal for systems that manage large volumes of structured data, such as customer relationship management (CRM) systems, financial reporting tools, and e-commerce platforms. Its focus on performance and scalability makes it indispensable for enterprise-level database operations. Moreover, SQL integrates seamlessly with most modern applications, ensuring compatibility and ease of use.

Prolog, on the other hand, is designed for logic-driven applications. Its ability to represent knowledge, evaluate logical relationships, and derive conclusions makes it highly suitable for artificial intelligence (AI), expert systems, and problem-solving tasks. For example, Prolog is often used in natural language processing, automated reasoning, and decision-support systems. However, Prolog's reliance on in-memory computation limits its performance when handling large datasets, making it less effective than SQL for data-intensive operations.

---

4. Key Differences in Syntax and Paradigm

- **Declarative vs. Logical Representation**:
  SQL queries are declarative and focus on specifying data relationships and conditions, whereas Prolog queries are logic-based, relying on a set of facts and rules.
- **Tabular vs. Logical Inference**:
  SQL results are typically presented in structured rows and columns, suitable for direct reporting or further processing. Prolog, however, provides results through logical inference, allowing for dynamic problem-solving.
- **Performance with Large Datasets**:
  SQL is optimized for operations on large, persistent datasets stored in relational databases. Prolog operates on in-memory data and performs best with smaller datasets or highly interconnected information.

---

5. Suitability Based on Application

- **SQL**:
  SQL is better suited for domains like business applications, financial systems, data warehousing, and other scenarios requiring robust data management. Its strengths lie in scalability, reliability, and support for a wide range of analytics tasks.
- **Prolog**:
  Prolog is ideal for use cases where reasoning and decision-making are critical. Examples include AI-based recommendation engines, rule-based eligibility systems, and complex problem-solving scenarios where logic and inference are needed.

Conclusion

While both SQL and Prolog are tools for querying data, their distinct approaches cater to different needs. SQL is unparalleled for managing and retrieving structured data from relational databases efficiently, whereas Prolog shines in scenarios that require logical reasoning and rule-based evaluations. Choosing between the two depends on the specific requirements of the task: SQL for structured data operations and scalability, Prolog for reasoning and complex logic evaluations. Together, they reflect the diversity of approaches available for data querying and processing in the modern technological landscape.

Objective 3: To use Monte Carlo simulation in Python to solve inference problems in a given Bayesian Belief Network (BBN) and analyze the results.

Question: Given a Bayesian Belief Network (BBN) with defined nodes and conditional probability distributions, implement a Monte Carlo simulation in Python to perform inference on this network. Select a target node for which you want to compute the probability given evidence on one or more other nodes. Run the Monte Carlo simulation by generating random samples and estimating the conditional probability of the target node based on these samples. Provide the computed probability and discuss the accuracy of the result by comparing it to known values (if available) or explaining how sample size affects convergence in your simulation

CODE:

```
import numpy as np
# Define conditional probabilities
P_Cloudy = 0.5
P_Sprinkler_given_Cloudy = {True: 0.1, False: 0.5}
P_Rain_given_Cloudy = {True: 0.8, False: 0.2}
P_WetGrass_given_Sprinkler_Rain = {
    (True, True): 0.99,
    (True, False): 0.90,
    (False, True): 0.80,
    (False, False): 0.00
}

# Monte Carlo simulation to estimate P(WetGrass=True | Rain=True)
def monte_carlo_simulation(sample_count=10000):
    wet_grass_and_rain_count = 0  # Count cases where WetGrass=True and Rain=True
    rain_count = 0               # Count cases where Rain=True

    for _ in range(sample_count):
        # Sample Cloudy
        is_cloudy = np.random.rand() < P_Cloudy

        # Sample Sprinkler given Cloudy
        is_sprinkler_on = np.random.rand() < P_Sprinkler_given_Cloudy[is_cloudy]

        # Sample Rain given Cloudy
        is_raining = np.random.rand() < P_Rain_given_Cloudy[is_cloudy]

        # Sample WetGrass given Sprinkler and Rain
        is_wet_grass = np.random.rand() <
P_WetGrass_given_Sprinkler_Rain[(is_sprinkler_on, is_raining)]

        # Increment counts for Rain and WetGrass given Rain
        if is_raining:
            rain_count += 1
            if is_wet_grass:
                wet_grass_and_rain_count += 1

    # Calculate conditional probability
    if rain_count == 0:  # Avoid division by zero
        return 0
    return wet_grass_and_rain_count / rain_count

# Run simulation
estimated_probability = monte_carlo_simulation()
print(f"Estimated P(WetGrass=True | Rain=True): {estimated_probability:.4f}")
```

OUTPUT:

```
PS C:\Users\asus\Workspace> & C:/Users/asus/AppData/Local/Microsoft/WindowsApps/python3.13.exe c:/Users/asus/Workspace/q3.py
Estimated P(WetGrass=True | Rain=True): 0.8256
PS C:\Users\asus\Workspace>
```