NAME – KARAN DUGAR

REG. NUMBER – 22BCI0189

ARTIFICIAL INTELLIGENCE – BCSE306L

FACULTY – PROF. SARAVANAGURU RA.K

DIGITAL ASSIGNMENT - 2

Objective 1: Solve the 8-Puzzle Problem Using the Hill Climbing Algorithm

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.

1. **8-Puzzle Problem:**

- Configuration: The board has 8 numbered tiles (1 to 8) and one blank space (0) on a 3x3 grid.
- Operations: Tiles can slide into the blank space.
- Goal: Rearrange tiles to match a target configuration using minimal moves.

2. **Hill Climbing Algorithm:**

- Process: Iteratively select the next state with the lowest heuristic cost.
- Heuristics:
    - Manhattan Distance: Ideal for sliding puzzles since it calculates the minimum steps required to place a tile in its goal position.
    - Misplaced Tiles: Simpler but less accurate, counts tiles not in their goal positions.
- Limitations:
    - Local Maxima: The algorithm gets stuck in a suboptimal state that seems better locally but isn't the global best.
    - Plateaus: Flat heuristic values across multiple states can halt progress.
    - Lack of Backtracking: Hill climbing cannot recover from poor decisions.

3. **Applications:**

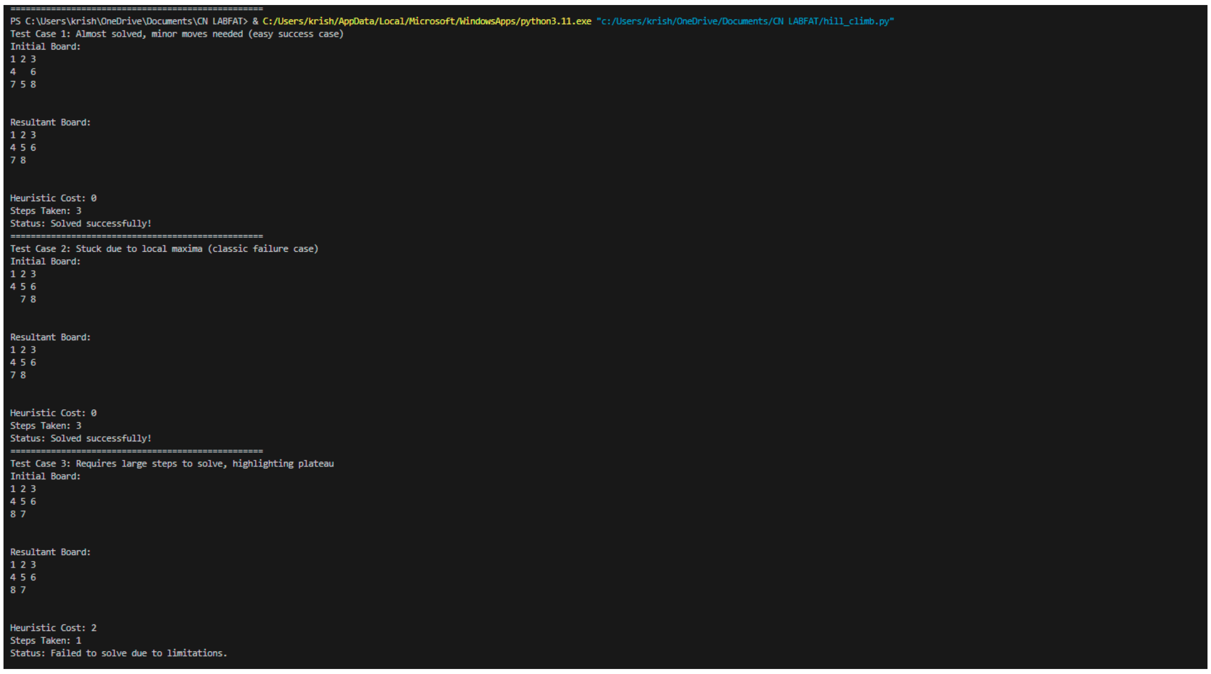
- Pathfinding problems.
- Game state optimization.

IMPLEMENTATION

```
import numpy as np
from copy import deepcopy

class Puzzle:
    def __init__(self, board):
        self.board = board
        self.goal = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

    def manhattan_distance(self, board=None):
        if board is None:
            board = self.board
        dist = 0
        for i in range(3):
            for j in range(3):
                value = board[i][j]
                if value != 0:
                    target_x, target_y = divmod(value - 1, 3)
                    dist += abs(target_x - i) + abs(target_y - j)
        return dist

    def get_neighbors(self, board=None):
        if board is None:
            board = self.board
        neighbors = []
        x, y = [(i, row.index(0)) for i, row in enumerate(board) if 0 in
row][0]
        moves = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]
        for dx, dy in moves:
            if 0 <= dx < 3 and 0 <= dy < 3:
                new_board = deepcopy(board)
                new_board[x][y], new_board[dx][dy] = new_board[dx][dy],
new_board[x][y]
                neighbors.append(new_board)
        return neighbors

    def hill_climbing(self):
        current = deepcopy(self.board)
        path = [current]
        while True:
            neighbors = self.get_neighbors(current)
            neighbors.sort(key=self.manhattan_distance)
            best_neighbor = neighbors[0]
            if self.manhattan_distance(best_neighbor) >=
self.manhattan_distance(current):
                break
            current = best_neighbor
            path.append(current)
```

```python
        return current, self.manhattan_distance(current), path

    def display_board(self, board):
        for row in board:
            print(" ".join(str(x) if x != 0 else " " for x in row))
        print("\n")
test_cases = [
    {
        "initial": [[1, 2, 3], [4, 0, 6], [7, 5, 8]],
        "description": "Almost solved, minor moves needed (easy success case)"
    },
    {
        "initial": [[1, 2, 3], [4, 5, 6], [0, 7, 8]],
        "description": "Stuck due to local maxima (classic failure case)"
    },
    {
        "initial": [[1, 2, 3], [4, 5, 6], [8, 7, 0]],
        "description": "Requires large steps to solve, highlighting plateau"
    }
]

for i, test in enumerate(test_cases, start=1):
    print(f"Test Case {i}: {test['description']}")
    puzzle = Puzzle(test["initial"])
    print("Initial Board:")
    puzzle.display_board(test["initial"])

    result, cost, path = puzzle.hill_climbing()

    print("Resultant Board:")
    puzzle.display_board(result)
    print(f"Heuristic Cost: {cost}")
    print(f"Steps Taken: {len(path)}")
    if cost == 0:
        print("Status: Solved successfully!")
    else:
        print("Status: Failed to solve due to limitations.\n")
    print("="*50)
```

RESULT



```
==================================================
PS C:\Users\krish\OneDrive\Documents\CN LABFAT> & C:/Users/krish/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/krish/OneDrive/Documents/CN LABFAT/hill_climb.py"
Test Case 1: Almost solved, minor moves needed (easy success case)
Initial Board:
1 2 3
4   6
7 5 8


Resultant Board:
1 2 3
4 5 6
7 8


Heuristic Cost: 0
Steps Taken: 3
Status: Solved successfully!
==================================================
Test Case 2: Stuck due to local maxima (classic failure case)
Initial Board:
1 2 3
4 5 6
  7 8


Resultant Board:
1 2 3
4 5 6
7 8


Heuristic Cost: 0
Steps Taken: 3
Status: Solved successfully!
==================================================
Test Case 3: Requires large steps to solve, highlighting plateau
Initial Board:
1 2 3
4 5 6
8 7


Resultant Board:
1 2 3
4 5 6
8 7


Heuristic Cost: 2
Steps Taken: 1
Status: Failed to solve due to limitations.
```

Objective 2: Prolog Rules for Eligibility & REST API

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.

Prolog Overview:

- A declarative logic programming language.
- Works with facts, rules, and queries.
- Ideal for scenarios involving symbolic reasoning and rule-based logic.

Eligibility Rules:

- Scholarships and exam permissions require meeting attendance and CGPA thresholds.
- These conditions can be logically represented as Prolog rules.

Comparison: SQL vs Prolog:

- **SQL**:
  - Focuses on structured data storage.
  - Uses SELECT queries for data retrieval.
  - Suitable for operations involving large datasets and relational tables.
- **Prolog**:
  - Focuses on logic and relationships.
  - Uses backward reasoning (starts with a goal and looks for facts to satisfy it).
  - Ideal for problems requiring reasoning and inference.

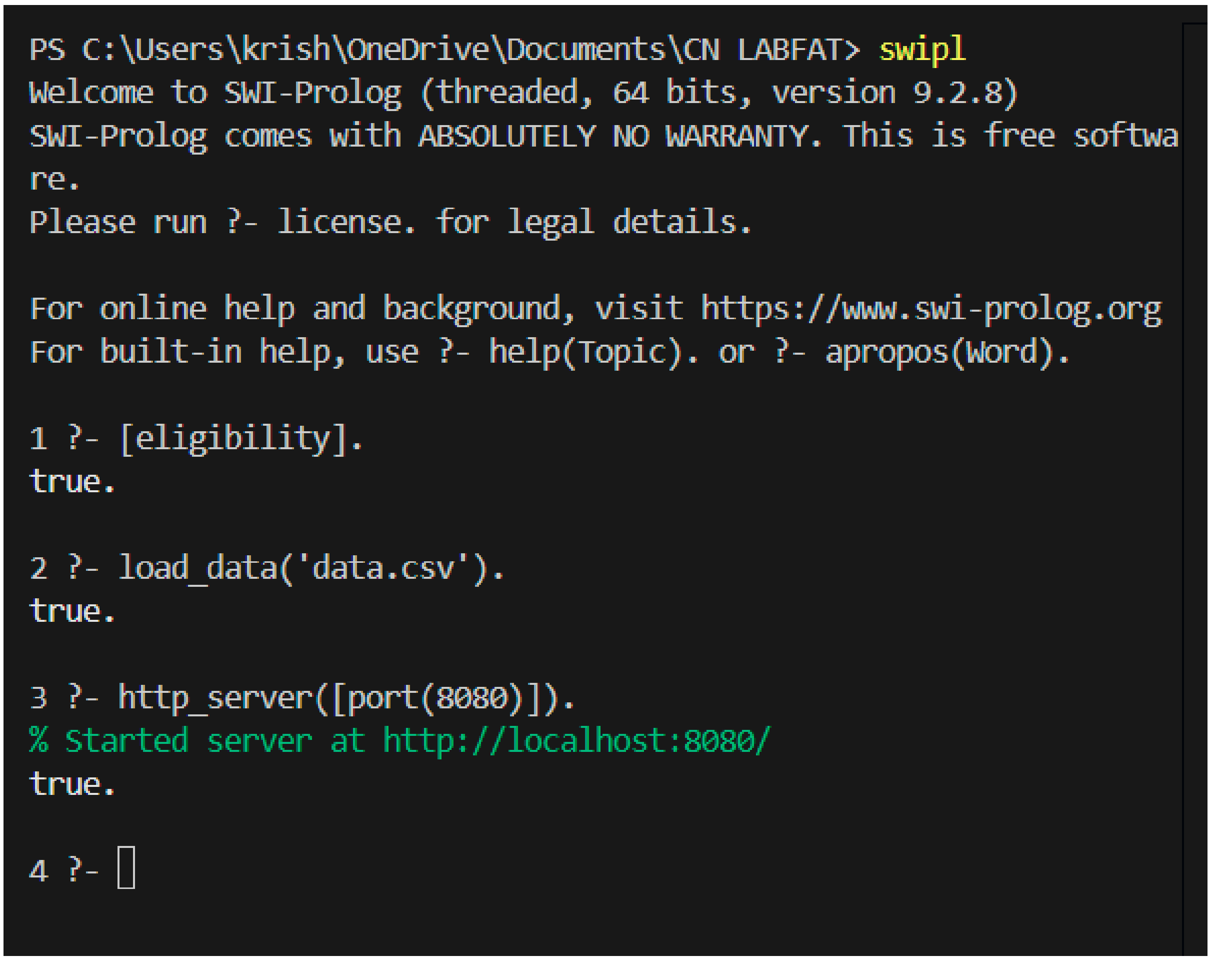
REST API Integration:

- Prolog's HTTP library can expose rules through endpoints.
- Enables web apps or systems to query eligibility dynamically.

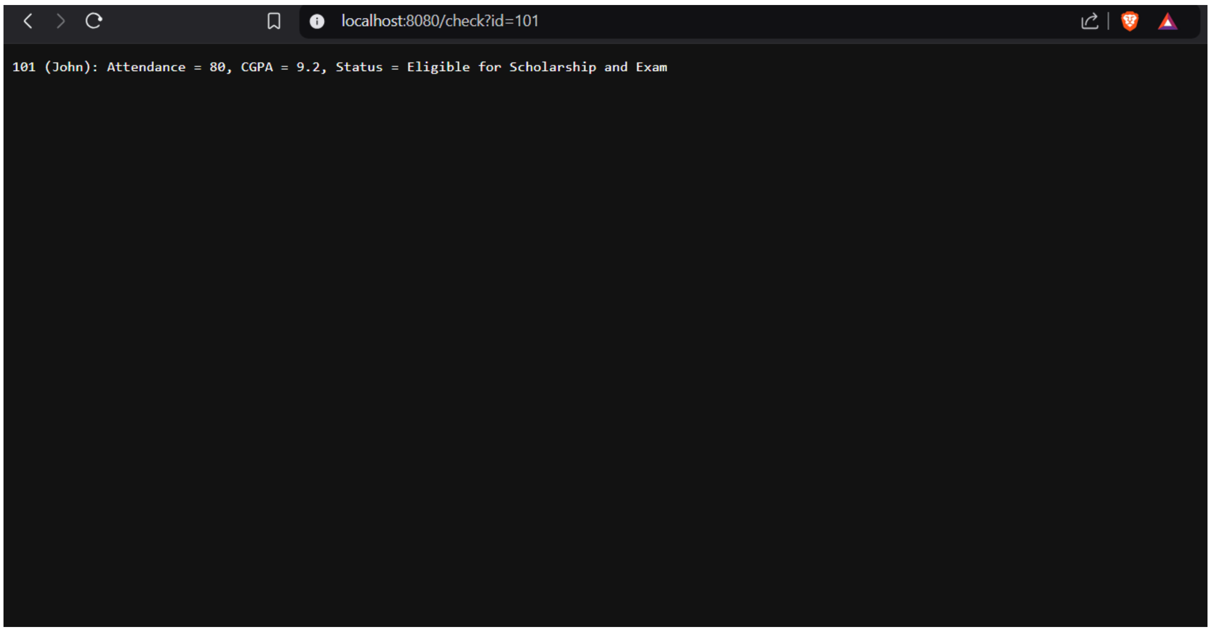
IMPLEMENTATION

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

% Dynamic predicate to store student data
:- dynamic student/4.

% Load data from CSV file
load_data(File) :-
    retractall(student(_, _, _, _)), % Clear existing data
    csv_read_file(File, Rows, [functor(student), arity(4)]),
    maplist(assert, Rows).

% Eligibility Rules
eligible_for_scholarship(Student_ID) :-
    student(Student_ID, _, Attendance, CGPA),
    Attendance >= 75, CGPA >= 9.0.

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

% REST API Handler
:- http_handler(root(check), handle_request, []).

% Handle Requests
handle_request(Request) :-
    (   http_parameters(Request, [id(StudentID, [integer])]) ->
        % Query for specific StudentID
        (   student(StudentID, Name, Attendance, CGPA) ->
            (   eligible_for_scholarship(StudentID) ->
                Status = "Eligible for Scholarship and Exam";
                permitted_for_exam(StudentID) ->
                Status = "Eligible for Exam Only";
                Status = "Not Eligible"
            ),
            format('Content-type: text/plain~n~n'),
            format('~w (~w): Attendance = ~w, CGPA = ~w, Status = ~w~n',
                   [StudentID, Name, Attendance, CGPA, Status])
        ;   % If StudentID not found
            format('Content-type: text/plain~n~n'),
            format('Student ID ~w not found~n', [StudentID])
        )
    ;   % Query for all students if no StudentID is provided
        findall(
            [StudentID, Name, Attendance, CGPA, Status],
            (
                student(StudentID, Name, Attendance, CGPA),
                (   eligible_for_scholarship(StudentID) ->
                    Status = "Eligible for Scholarship and Exam";
                    permitted_for_exam(StudentID) ->
                    Status = "Eligible for Exam Only";
                    Status = "Not Eligible"
                )
            ),
            Results
        ),
        format('Content-type: text/plain~n~n'),
        print_results(Results)
    ).

% Print results for all students
print_results([]).
print_results([[StudentID, Name, Attendance, CGPA, Status] | Rest]) :-
    format('~w (~w): Attendance = ~w, CGPA = ~w, Status = ~w~n',
           [StudentID, Name, Attendance, CGPA, Status]),
    print_results(Rest).
```

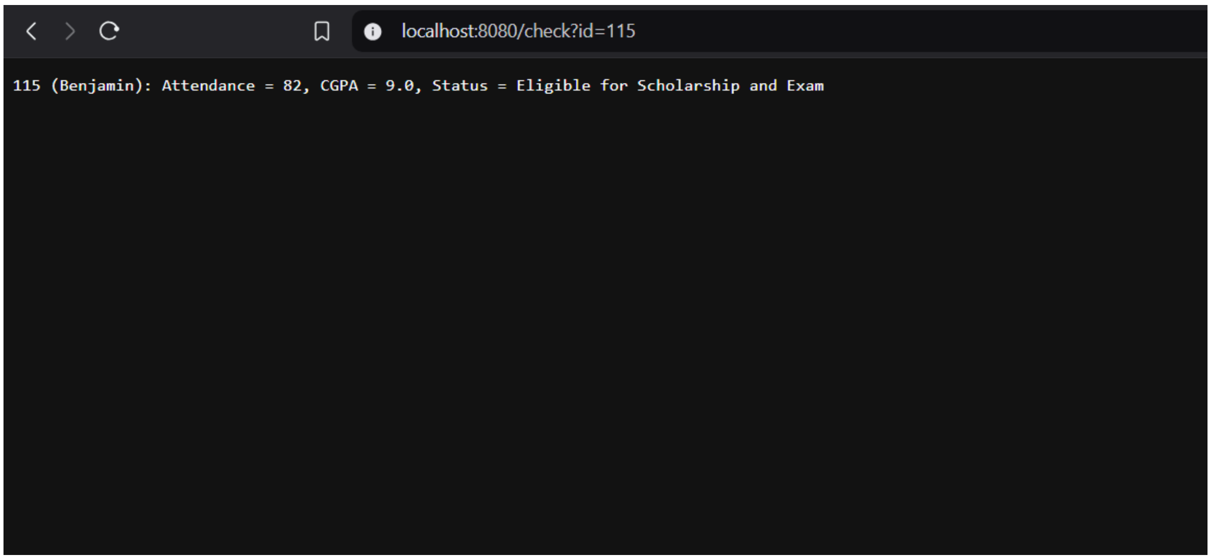
RESULT



```
PS C:\Users\krish\OneDrive\Documents\CN LABFAT> swipl
Welcome to SWI-Prolog (threaded, 64 bits, version 9.2.8)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free softwa
re.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

1 ?- [eligibility].
true.

2 ?- load_data('data.csv').
true.

3 ?- http_server([port(8080)]).
% Started server at http://localhost:8080/
true.

4 ?- 
```



```
101 (John): Attendance = 80, CGPA = 9.2, Status = Eligible for Scholarship and Exam
```



```
115 (Benjamin): Attendance = 82, CGPA = 9.0, Status = Eligible for Scholarship and Exam
```

Objective 3: Monte Carlo Simulation for Bayesian Belief Networks

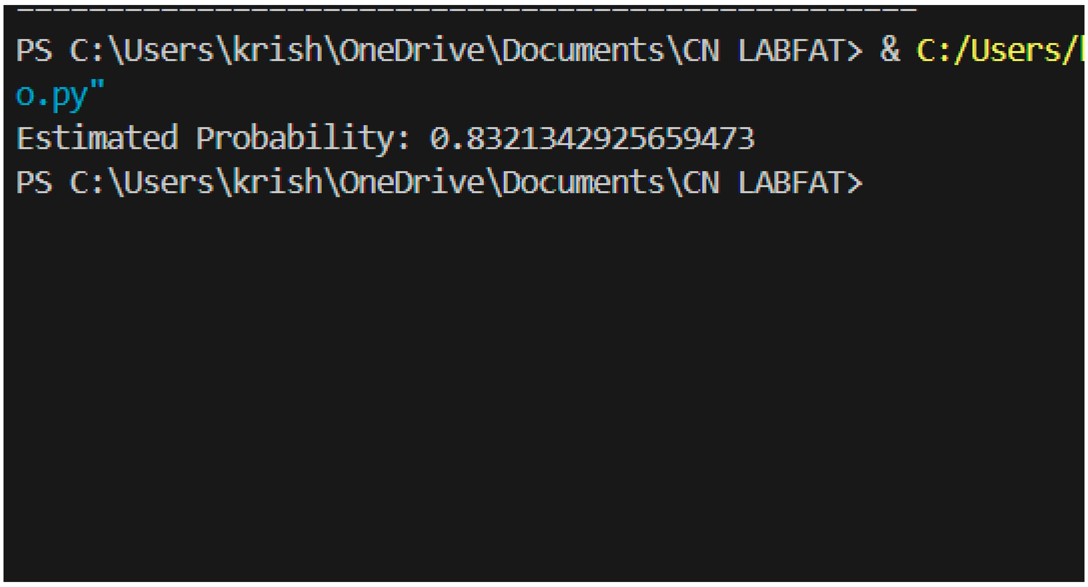
Expanded Theoretical Content

1. **Bayesian Belief Networks**:
   - Graphical models where nodes represent random variables, and edges represent conditional dependencies.
   - Each node has a conditional probability distribution (CPD).
   - Example: **Weather Network**:
     - Variables: Cloudy, Rain, Sprinkler, Wet Grass.
     - Edges represent causal relationships like Cloudy → Rain.
2. **Monte Carlo Simulation**:
   - A computational algorithm using random sampling to approximate results.
   - **Key Steps**:
     - Sample random values based on CPDs.
     - Count occurrences to estimate probabilities.
   - **Convergence**: Accuracy improves as the number of samples increases.
3. **Applications**:
   - Weather prediction.
   - Decision support systems.
   - Medical diagnosis (e.g., disease probabilities given symptoms).
4. **Strengths and Limitations**:
   - **Strengths**:
     - Handles uncertainty and incomplete data.
     - Provides approximate solutions where exact computation is infeasible.
   - **Limitations**:
     - Requires a large number of samples for accurate results.
     - Sensitive to the quality of CPDs

IMPLEMENTATION

```python
import numpy as np
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
}

def monte_carlo_simulation(num_samples=10000):
    count_wet_grass_given_rain = 0
    count_rain = 0
    for _ in range(num_samples):
        cloudy = np.random.rand() < P_Cloudy
        sprinkler = np.random.rand() < P_Sprinkler_given_Cloudy[cloudy]
        rain = np.random.rand() < P_Rain_given_Cloudy[cloudy]
        wet_grass = np.random.rand() <
P_WetGrass_given_Sprinkler_Rain[(sprinkler, rain)]
        if rain:
            count_rain += 1
            if wet_grass:
                count_wet_grass_given_rain += 1
    return count_wet_grass_given_rain / count_rain if count_rain > 0 else 0

print(f"Estimated Probability: {monte_carlo_simulation()}")
```

RESULT



```
PS C:\Users\krish\OneDrive\Documents\CN LABFAT> & C:/Users/
o.py"
Estimated Probability: 0.8321342925659473
PS C:\Users\krish\OneDrive\Documents\CN LABFAT>
```