DIGITAL ASSIGNMENT 2

NAME : Lavanya Kushmakar

REGISTRATION NUMBER : 22BCE0038

SEMESTER : FALL SEMESTER 2024-25

COURSE CODE : BCSE306L

COURSE TITLE : ARTIFICIAL INTELLIGENCE

FACULTY NAME : SARAVANAGURU RA.K

SLOT : E1+TE1

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 in python:

```
import copy
import random
from typing import List, Tuple, Optional

class EightPuzzle:
    def __init__(self, initial_state: List[List[int]]):
        self.current_state = initial_state
        self.goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

    def find_empty_tile(self) -> Tuple[int, int]:
        for i in range(3):
            for j in range(3):
                if self.current_state[i][j] == 0:
                    return i, j
        return -1, -1

    def get_possible_moves(self, i: int, j: int) -> List[Tuple[int, int]]:
        moves = []
        for di, dj in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
            new_i, new_j = i + di, j + dj
            if 0 <= new_i < 3 and 0 <= new_j < 3:
                moves.append((new_i, new_j))
        return moves

    def make_move(self, from_pos: Tuple[int, int], to_pos: Tuple[int, int]) -> None:
        i1, j1 = from_pos
        i2, j2 = to_pos
        self.current_state[i1][j1], self.current_state[i2][j2] = \
            self.current_state[i2][j2], self.current_state[i1][j1]

    def manhattan_distance(self) -> int:
        distance = 0
        for i in range(3):
            for j in range(3):
                value = self.current_state[i][j]
                if value != 0:
                    goal_i, goal_j = (value-1) // 3, (value-1) % 3
                    distance += abs(i - goal_i) + abs(j - goal_j)
        return distance
```

```
    def misplaced_tiles(self) -> int:
        count = 0
        for i in range(3):
            for j in range(3):
                if self.current_state[i][j] != 0 and \
                   self.current_state[i][j] != self.goal_state[i][j]:
                    count += 1
        return count

    def hill_climbing(self, max_iterations: int = 1000,
                      heuristic: str = 'manhattan') -> Tuple[bool, int]:
        steps = 0

        while steps < max_iterations:
            current_score = self.manhattan_distance() if heuristic == 'manhattan' \
                            else self.misplaced_tiles()

            if current_score == 0:
                return True, steps

            empty_i, empty_j = self.find_empty_tile()
            possible_moves = self.get_possible_moves(empty_i, empty_j)

            best_score = current_score
            best_move = None

            for move in possible_moves:
                self.make_move((empty_i, empty_j), move)
                new_score = self.manhattan_distance() if heuristic == 'manhattan' \
                            else self.misplaced_tiles()
                self.make_move(move, (empty_i, empty_j))

                if new_score < best_score:
                    best_score = new_score
                    best_move = move

            if best_move is None:
                return False, steps

            self.make_move((empty_i, empty_j), best_move)
            steps += 1

        return False, steps

def print_board(state: List[List[int]]) -> None:
    for row in state:
        print(row)
    print()
```

```
def get_user_input():
    print("Enter the initial state of the 8-puzzle (use 0 for empty tile)")
    print("Enter numbers row by row (space-separated):")
    initial_state = []
    for i in range(3):
        row = list(map(int, input(f"Enter row {i+1}: ").strip().split()))
        initial_state.append(row)
    return initial_state

def main():
    print("8-Puzzle Solver using Hill Climbing")
    initial_state = get_user_input()

    print("\nSelect heuristic:")
    print("1. Manhattan Distance")
    print("2. Misplaced Tiles")
    choice = input("Enter choice (1 or 2): ")

    heuristic = 'manhattan' if choice == '1' else 'misplaced'

    puzzle = EightPuzzle(initial_state)
    print("\nInitial State:")
    print_board(puzzle.current_state)

    success, steps = puzzle.hill_climbing(heuristic=heuristic)

    print(f"Solution found: {success}")
    print(f"Steps taken: {steps}")
    print("Final State:")
    print_board(puzzle.current_state)

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

Output:

Algorithm passes

```
Output                                                    Clear
8-Puzzle Solver using Hill Climbing
Enter the initial state of the 8-puzzle (use 0 for empty tile)
Enter numbers row by row (space-separated):
Enter row 1: 1 2 3
Enter row 2: 4 0 6
Enter row 3: 7 5 8

Select heuristic:
1. Manhattan Distance
2. Misplaced Tiles
Enter choice (1 or 2): 1

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

Solution found: True
Steps taken: 2
Final State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

Algorithm Fails

Reason:

The algorithm works by evaluating neighboring states and selecting moves that decrease the distance to the goal state. However, in this configuration, all possible moves would temporarily increase the Manhattan distance, and since Hill Climbing doesn't allow for "worse" intermediate states that might eventually lead to a better solution, it fails immediately without taking any steps. This highlights a fundamental limitation of the Hill Climbing approach - its inability to escape local maxima by accepting temporarily suboptimal moves that could ultimately lead to the global optimum.

```
8-Puzzle Solver using Hill Climbing
Enter the initial state of the 8-puzzle (use 0 for empty tile)
Enter numbers row by row (space-separated):
Enter row 1: 4 5 6
Enter row 2: 8 9 4
Enter row 3: 6 7 8

Select heuristic:
1. Manhattan Distance
2. Misplaced Tiles
Enter choice (1 or 2): 1

Initial State:
[4, 5, 6]
[8, 9, 4]
[6, 7, 8]

Solution found: False
Steps taken: 0
Final State:
[4, 5, 6]
[8, 9, 4]
[6, 7, 8]
```

Limitations:

a) Local Maxima:

- The algorithm can get stuck in local maxima where no immediate moves improve the situation
- This happens when all neighboring states have worse heuristic values
- The implementation detects this when no better moves are available

b) Plateaus:

- Areas in the state space where neighboring states have equal heuristic values
- The algorithm might wander without making progress
- Current implementation might stop at plateaus since it requires strict improvement

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.

Note:

CSV file-
data.csv

```
Student_ID,Name,Attendance_percentage,CGPA
38,Lavanya,80,9.2
25,Ram,60,8.5
31,Nikhil,90,9.5
42,Shreya,70,6.8
```

Setup the Prolog Environment and Load the CSV

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

Code.pl

```
load_student_data :-
    csv_read_file("data.csv", Rows, [functor(student), arity(4)]),
    maplist(assert, Rows).
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.
start_server(Port) :-
   http_server(http_dispatch, [port(Port)]).

:- http_handler('/eligibility', eligibility_handler, []).

eligibility_handler(Request) :-
   http_parameters(Request, [id(Student_ID, [atom])]),
   eligibility_status(Student_ID, Status),
   reply_json_dict(Status).

eligibility_status(Student_ID, Status) :-
   (   eligible_for_scholarship(Student_ID)
   ->  Scholarship = "Eligible"
   ;   Scholarship = "Not Eligible"
   ),
   (   permitted_for_exam(Student_ID)
   ->  Exam = "Permitted"
   ;   Exam = "Debarred"
   ),
   Status = _{student_id: Student_ID, scholarship: Scholarship, exam_permission: Exam}.
```

Web page:

HTML file

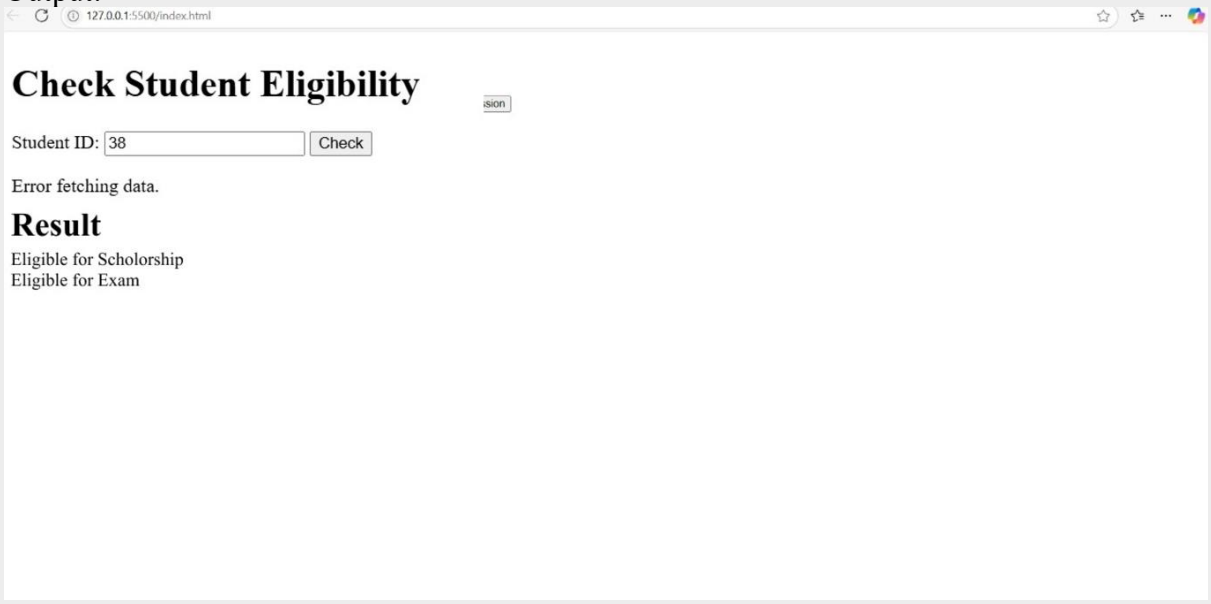
Index.html

```
<!DOCTYPE html>
<html>
<head>
   <title>Student Eligibility Checker</title>
</head>
<body>
   <h1>Check Student Eligibility</h1>
   <form id="eligibilityForm">
      <label for="studentId">Student ID:</label>
      <input type="text" id="studentId" name="studentId" required>
      <button type="submit">Check</button>
   </form>
   <div id="result"></div>

   <script>
      document.getElementById('eligibilityForm').addEventListener('submit', async (event)
=> {
         event.preventDefault();
         const studentId = document.getElementById('studentId').value;
         try {
            const response = await fetch(`/eligibility?id=${studentId}`);
            const data = await response.json();
            document.getElementById('result').innerHTML = `
```

```
                <p>Student ID: ${data.student_id}</p>
                <p>Scholarship: ${data.scholarship}</p>
                <p>Exam Permission: ${data.exam_permission}</p>
            `;
        } catch (error) {
            document.getElementById('result').innerHTML = `<p>Error fetching data.</p>`;
        }
    });
  </script>
</body>
</html>
```

Output:



Comparison: SQL vs Prolog

Aspect	SQL	Prolog
Query Type	Data retrieval and aggregation.	Logic-based conditions and reasoning.
Data Handling	Relational; tables with structured data.	Facts and rules for logical inference.
Suitability	Ideal for transactional and batch queries.	Best for reasoning over rules and logic.
Complex Queries	Uses joins and subqueries.	Uses recursive rules and backtracking.
Performance	Optimized for large datasets.	Efficient for logical constraints, not large datasets.

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.

Bayesian Belief Networks

Definition: Graphical models where nodes are random variables and edges show conditional dependencies.

Features:

- Each node has a Conditional Probability Distribution (CPD).
- Captures causal relationships between variables.
  Example: Weather Network
- **Variables:** Cloudy, Rain, Sprinkler, Wet Grass.
- **Edges:** Show causal links, e.g., Cloudy → Rain.

Monte Carlo Simulation

Definition: A method using random sampling to approximate probabilities or numerical results.

Steps:

1. Sample random values based on CPDs.
2. Count outcomes to estimate probabilities.
   Accuracy: Improves with more samples.

Applications

- Weather prediction.
- Decision-making under uncertainty.
- Diagnosing medical conditions.

Strengths and Limitations

Strengths:

- Handles uncertainty and incomplete data efficiently.
- Useful for complex problems where exact solutions are hard.
  Limitations:
- Needs many samples for accuracy.
- Relies on correct CPDs.

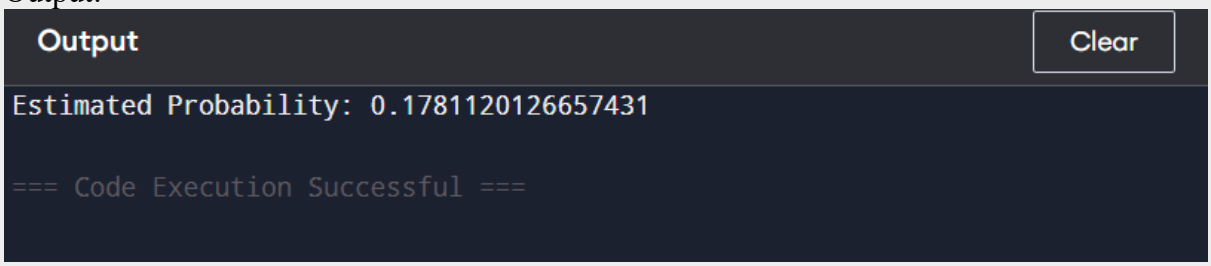
Code:

```
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_sprinkler_given_rain = 0
    count_rain = 0

    for _ in range(num_samples):
        # Simulate events
        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 sprinkler:
                count_sprinkler_given_rain += 1
    return count_sprinkler_given_rain / count_rain if count_rain > 0 else 0
print(f"Estimated Probability: {monte_carlo_simulation()}")
```

Output:



How sample size affects:
In Monte Carlo methods, the law of large numbers ensures that as the number of samples increases, the estimated probability converges toward the true probability. However, this convergence is stochastic, meaning that the accuracy improves in a probabilistic sense. The more samples you generate, the better the approximation of the target probability becomes. For example, with a small sample size, the estimate might be significantly biased due to random fluctuations. However, as the sample size grows, these fluctuations average out, and the result approaches the true value.