Hill Climbing and Simulated Annealing Algorithm Implementation

Overview

This submission implements two optimization algorithms over binary strings:

(1) Hill-Climbing (100 random restarts) to maximize f(v)=|13·one(v)-170| where v is a 40-bit vector

(2) (2) Simulated Annealing (200 random restarts) to maximize f(v)=|14·one(v)-190| where v is a 50-bit vector

How to Run

1) Unzip the file

2) Run .\dist\main.exe, and the output will be printed to Output.tx

Source Code (main.py)

```
import random
import math
from typing import List

# Count the number of 1's in a bit vector
def one_count(bits: List[int]) -> int:
    return sum(bits)

# Generate a random bit vector of length
def generate_initial_solution(length: int) -> List[int]:
    return [random.choice([0, 1]) for _ in range(length)]

# Return a NEW vector with bit at idx flipped
def flip_bit(bits: List[int], idx: int) -> List[int]:
    new_bits = bits[:]  # copy
    new_bits[idx] = 1 - new_bits[idx]
    return new_bits


# Part 1: Hill Climbing
# f(v) = |13*one(v) - 170|, v is 40 bits

def f_part1(bits: List[int]) -> int:
    return abs(13 * one_count(bits) - 170)

def hill_climb_once(max_Iterations: int) -> int:

    currnt_solution = generate_initial_solution(max_Iterations)
    currnt_value = f_part1(currnt_solution)

    improved = True
    while improved:
        improved = False

        # explore all neighbors
        best_neighbor = None
        best_val = currnt_value

        for _ in range(max_Iterations):
            neighbor = flip_bit(currnt_solution, _)
            idx_to_flip = f_part1(neighbor)
            if idx_to_flip > best_val:
                best_val = idx_to_flip
                best_neighbor = neighbor

        # move to new neighbor
        if best_neighbor is not None:
            currnt_solution = best_neighbor
            currnt_value = best_val
            improved = True

    return currnt_value

# Run hill climbing with multiple random restarts
def hill_climb_restart(max_restart: int = 100, max_Iterations: int = 40) -> List[int]:
    results = []
    for _ in range(max_restart):
        results.append(hill_climb_once(max_Iterations))
    return results

# Part 2: Simulated Annealing
# f(v) = |14*one(v) - 190|, v is 50 bits

def f_part2(bits: List[int]) -> int:
    return abs(14 * one_count(bits) - 190)

def simulated_annealing_once(
    max_Iterations: int,
```

```
    iterations: int = 800,
    T0: float = 20.0,
    cooling_rate: float = 0.995
) -> int:

    currnt_solution = generate_initial_solution(max_Iterations)
    currnt_value = f_part2(currnt_solution)
    best_val = currnt_value

    temp = T0
    for _ in range(iterations):
        # pick a random neighbor by flipping one random bit
        idx = random.randrange(max_Iterations)
        neighbor = flip_bit(currnt_solution, idx)
        neighbor_eval = f_part2(neighbor)

        diff = neighbor_eval - currnt_value
        if diff >= 0:
            currnt_solution = neighbor
            currnt_value = neighbor_eval
        else:
            if temp > 1e-12:
                prob = math.exp(diff / temp)
                if random.random() < prob:
                    currnt_solution, currnt_value = neighbor, neighbor_eval

        if currnt_value > best_val:
            best_val = currnt_value

        temp *= cooling_rate

    return best_val

def simulated_annealing_restarts(max_restart: int = 200, max_Iterations: int = 50) -> List[int]:
    results = []
    for _ in range(max_restart):
        results.append(simulated_annealing_once(max_Iterations))
    return results


def main() -> None:

    part1_results = hill_climb_restart(max_restart=100, max_Iterations=40)
    part2_results = simulated_annealing_restarts(max_restart=200, max_Iterations=50)

    with open("Output.txt", "w", encoding="utf-8") as f:
        f.write(",".join(str(x) for x in part1_results) + "\n")
        f.write(",".join(str(x) for x in part2_results) + "\n")

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

Output (Output.txt)

```
350,350,350,350,350,350,350,350,350,350,350,350,350,350,350,350,350,350,350,
170,350,350,350,350,350,350,350,350,350,350,350,350,350,350,350,350,350,350,
350,350,350,350,350,350,350,350,350,350,350,350,350,350,350,350,170,350,350,
350,350,350,350,350,350,350,350,350,350,350,350,350,350,350,350,350,350,350,
350,350,350,350,350,350,350,350,350,350,350,350,350,350,350,350,350,350,350,
350,350,350,350,350
510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,
510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,
510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,
510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,
510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,
510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,
510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,
510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,
510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,
510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,510,
510,510,510,510,510,510,510,510,510,510
```