Documentation on Minimum Genetic Mutation - Complete Documentation

Table of Contents

1. Problem Statement

A gene string is represented as an 8-character long string with characters from {'A', 'C', 'G', 'T'}. Given:

- A startGene (valid mutation starting point).
- An endGene (target mutation).
- A list bank containing valid intermediate gene mutations.

Objective:

Find the **minimum number of mutations** required to convert startGene into endGene, where:

- Each mutation changes exactly **one** character.
- The resulting gene **must be in the bank**.
- If no mutation path exists, return -1.

Example 1:

Input:

startGene = "AACCGGTT"
endGene = "AACCGGTA"
bank = ["AACCGGTA"]

Output:1

Example 2:

Input:

startGene = "AACCGGTT"
endGene = "AAACGGTA"
bank = ["AACCGGTA", "AACCGCTA", "AAACGGTA"]

Output: 2

2. Intuition

This problem can be visualized as a **graph traversal**:

- Each valid gene represents a **node**.
- A mutation that changes one character is an **edge**.
- We need the **shortest path** from startGene to endGene, which suggests using **Breadth-First Search (BFS)**.

BFS is ideal because:

- It explores mutations **level-by-level**, ensuring the shortest path is found first.
- It avoids unnecessary computations using a **visited set**.

3. Key Observations

- If endGene is **not in bank**, return -1 immediately.
- Each mutation can only change **one character at a time**.
- There are **only 8 positions** and **4 possible characters** per position → max 24 possible mutations per gene.
- The bank length is **at most 10**, making BFS feasible.

4. Approach

Step 1: Convert bank to a Set for Fast Lookup

- A **set** provides O(1) lookup time to check valid mutations.

Step 2: Initialize BFS

- Use a queue storing tuples (currentGene, mutationCount).
- Start with (startGene, 0).
- Track visited genes to avoid cycles.

Step 3: Process Queue (BFS Traversal)

1. **Extract** a gene from the queue.
2. **Try mutating each character** at every position (A, C, G, T).
3. **If the new mutation is in bank and not visited**:
    - If it matches endGene, return mutations count + 1.
    - Otherwise, add it to the queue for further exploration.
4. If the queue is exhausted without finding endGene, return -1.

5. Edge Cases

✓ endGene not in bank → Return -1.

✓ startGene == endGene → Return 0.

✓ No valid mutations possible → Return -1.

✓ Multiple mutation paths exist → BFS ensures the shortest path is chosen.

✓ Smallest case: bank is empty → Return -1.

6. Complexity Analysis

Time Complexity

- Each gene has **8 positions**.
- Each position has **3 possible mutations** (excluding itself).
- There are at most **10 genes in the bank**.
- Worst-case scenario: **O(8 * 3 * N) = O(N)**, which is very efficient.

Space Complexity

- BFS queue stores up to **N genes** → O(N).
- Visited set and bank set → O(N).
- **Overall Space Complexity: O(N)**.

7. Alternative Approaches

a. Depth-First Search (DFS)

- Could work but **may not find the shortest path first**.
- Requires backtracking, leading to **higher time complexity**.

2. Dijkstra's Algorithm

- Overkill for this problem since BFS finds the shortest path efficiently.

3. Bidirectional BFS

- Would optimize performance but is unnecessary given the small constraints.

8. Test Cases

```
solution = Solution()

# Test Case 1
print(solution.minMutation("AACCGGTT", "AACCGGTA", ["AACCGGTA"]))  # Expected Output: 1

# Test Case 2
print(solution.minMutation("AACCGGTT", "AAACGGTA",
["AACCGGTA","AACCGCTA","AAACGGTA"]))  # Expected Output: 2

# Test Case 3 (Impossible case)
print(solution.minMutation("AACCGGTT", "AAACGGTA", ["AACCGGTC"]))  # Expected Output: -1

# Test Case 4 (EndGene not in bank)
print(solution.minMutation("AACCGGTT", "AACCGGTA", []))  # Expected Output: -1
```

9. Final Thoughts

- **BFS** ensures the shortest mutation path is found efficiently.
- **Set lookups** make operations fast.
- The approach works well given the problem constraints (**N ≤ 10**).
- For **larger constraints**, bidirectional BFS could be explored.