Motif Search and Entropy-Based Filtering in DNA Sequences

Manuel Ricardo Guerrero Cuéllar
University Francisco José de Caldas
Email: mrguerreroc@udistrital.edu.co

***Abstract*—This technical report outlines the design and implementation of a Java-based tool for detecting frequent motifs in DNA sequences. The tool generates artificial sequences, searches for motifs, filters sequences using Shannon entropy, and calculates the total execution time. Several technical decisions were made to improve flexibility, performance, and accuracy.**

I. Introduction

This report outlines the design and implementation of a Java-based tool for detecting frequent motifs in DNA sequences. The tool generates artificial DNA sequences, searches for motifs, filters sequences using Shannon entropy, and calculates the total execution time. The project involved several technical decisions aimed at improving flexibility, performance, and accuracy, including handling multiple motifs and optimizing execution with concurrency.

II. DNA Sequence Generation

The `DNAGenerator` class was used to generate artificial DNA sequences composed of nucleotides A, C, G, and T, representing the standard bases in genetic sequences. The length of each sequence and the probability distribution for nucleotide selection were configurable.

A. Decision

Probabilities for nucleotide selection were customizable via an array {A_prob, C_prob, G_prob, T_prob}. This allows the user to simulate different biological environments where some nucleotides may be more frequent than others.

B. Reason

In real-world data, nucleotides are not always evenly distributed. Allowing customizable probabilities ensures that this tool can simulate different conditions, making the results more realistic.

C. Performance Consideration

Random nucleotide generation was optimized using `ThreadLocalRandom` for thread safety and better performance in multi-threaded environments.

D. Reason

Random number generation must be thread-safe when used concurrently. `ThreadLocalRandom` is faster and more scalable than traditional `Random` in multi-threaded contexts.

III. Motif Detection

The `MotifFinder` class handles motif detection by scanning each sequence for motifs of a given size and counting their occurrences.

A. Decision

We handled multiple motifs that occur with the same frequency.

B. Reason

In cases where more than one motif has the same number of occurrences, the algorithm should not arbitrarily discard motifs. Biologically, multiple motifs could be equally significant, so the program returns all motifs with the maximum frequency.

C. Concurrency

We used `ExecutorService` and `ConcurrentHashMap` to parallelize the process of motif detection. Each sequence is processed in a separate thread to speed up the search.

D. Reason

DNA sequences can be long, and the dataset can be large (up to 2 million sequences). Parallelizing motif detection reduces time complexity and makes the tool suitable for larger datasets.

E. Shutdown Handling

A special mechanism was implemented to handle the shutdown of the thread pool (`ExecutorService`) only after all tasks are completed.

F. Reason

Reusing the thread pool allowed us to run multiple motif searches (for both original and filtered datasets) without encountering a `RejectedExecutionException`. Proper shutdown management ensures efficient resource usage and avoids resource leaks.

IV. Entropy-Based Filtering

Shannon entropy was used to filter out repetitive or redundant sequences, retaining only those with enough diversity (chaos). Shannon entropy was calculated for each sequence, and only sequences with entropy higher than a user-defined threshold were retained.

A. Decision

The entropy threshold was made configurable.

B. Reason

Different datasets may require different thresholds depending on the level of randomness or repetition expected in the sequences. A lower entropy threshold allows more repetitive sequences, while a higher threshold filters out sequences with little variation.

C. Concurrency

Entropy calculations and filtering were also parallelized using `ExecutorService`.

D. Reason

Filtering can become a bottleneck when working with large datasets. Parallelization ensured that the filtering process did not slow down the overall performance of the tool.

V. Multiple Motif Search Scenarios

The tool was implemented to search for motifs in both the original dataset and the filtered dataset (after applying the entropy filter). This allows for comparing the most frequent motifs before and after removing redundant sequences.

A. Decision

Two separate motif searches were conducted: one on the original sequences and one on the filtered sequences.

B. Reason

By comparing the motifs from both datasets, we can evaluate how filtering based on entropy impacts the frequent motifs. This helps in understanding how much redundant information is removed by the entropy filter.

C. Handling of Empty Filtered Dataset

If no sequences pass the entropy filter, the program gracefully handles this case and informs the user.

D. Reason

This ensures robustness and avoids runtime errors when dealing with very strict filtering criteria.

VI. Execution Time Calculation

We measured the total time taken for the execution of the entire process, from sequence generation to motif detection and filtering. The total execution time was calculated using `System.nanoTime()` for high precision.

A. Decision

We measured the total execution time for the entire program rather than segmenting individual steps.

B. Reason

The user is primarily interested in how long the entire process takes, not just individual steps. Measuring total execution time provides an overall sense of performance, especially when working with large datasets.

VII. Handling Different Motif Sizes

The tool was designed to handle different motif sizes, ranging from small motifs (e.g., 4 bases) to larger motifs (e.g., 10 bases).

A. Decision

The motif size (s) was configurable, allowing users to search for both short and long motifs.

B. Reason

Different biological analyses require searching for motifs of varying lengths. For example, short motifs may appear more frequently, while longer motifs may have more biological significance but be harder to detect.

VIII. Performance Considerations

A. Parallel Processing

Concurrency was extensively used in both motif detection and entropy filtering.

B. Reason

Working with large datasets (e.g., up to 2 million sequences) can be computationally expensive. Parallelizing tasks allowed us to handle the load more efficiently, reducing execution times significantly.

C. Dynamic Thread Pool Management

The `ExecutorService` was reused and shut down only when necessary, ensuring efficient resource usage and preventing thread pool exhaustion.