Logarithmic Dynamic Cuckoo Filter (LDCF)

Overview

This repository contains an implementation of a Logarithmic Dynamic Cuckoo Filter (LDCF) in Java. The LDCF is a probabilistic data structure that supports fast and space-efficient insertions, lookups, and deletions. It is particularly useful for applications in bioinformatics, such as sequence searching in large genomes.

Requirements and Usage

- Java Development Kit (JDK) 8 or higher

To compile and run the the program, these commands must be executed from the origin folder:

- javac bioinformatics/LDCF.java bioinformatics/mainclass.java
- java bioinformatics.mainclass

Features

- **Dynamic resizing**: Automatically adjusts the size of the filter to handle more items.
- **Efficient insertion**: Handles collisions through a limited number of relocations (kicks). This is performed through the next functions:

```
public boolean insert(String item) {
    for (int level = 0; level < buckets.length; level++) {
        if (insertAtLevel(item, level)) {
            return true;
        }
    }
    resize();

    int level = hash1(item, 0) % buckets.length;
    return insertAtLevel(item, level);
}
private boolean insertAtLevel(String item, int level) {
        int i1 = hash1(item, level);
        int i2 = hash2(item, level);

        for (int j = 0; j < bucketSizes[level]; j++) {
            if (buckets[level][i1][j] == null) {
                buckets[level][i1][j] = item;
                return true;
            }

            if (buckets[level][i2][j] == null) {
                buckets[level][i2][j] = item;
                return true;
            }
        }

        // Eviction process if both buckets are full
        int currentBucket = random.nextBoolean() ? i1 : i2;
        for (int n = 0; n < maxKicks; n++) {
```

```
            int j = random.nextInt(bucketSizes[level]);

            if (buckets[level][currentBucket][j] == null) {
                buckets[level][currentBucket][j] = item;
                return true;
            }

            String evictedItem = buckets[level][currentBucket][j];
            buckets[level][currentBucket][j] = item;
            item = evictedItem;
            currentBucket = (currentBucket == i1) ? i2 : i1;
        }

        return false;
    }
```

- **Fast lookup**: Quickly checks for the presence of an item. Mainly performed by:

```
public boolean lookup(String item) {
    for (int level = 0; level < buckets.length; level++) {
        if (lookupAtLevel(item, level)) {
            return true;
        }
    }
    return false;
}
private boolean lookupAtLevel(String item, int level) {
        int i1 = hash1(item, level);
        int i2 = hash2(item, level);

       for (int j = 0; j < bucketSizes[level]; j++) {
                        if (buckets[level][i1][j] != null &&
            buckets[level][i1][j].contains(item)) {
                            return true;
                        }

                        if (buckets[level][i2][j] != null &&
            buckets[level][i2][j].contains(item)) {
                            return true;
                        }
                    }


        return false;
    }
```

- **Deletion support**: Allows items to be removed from the filter. It works calling the next functions:

```
public boolean delete(String item) {
        for (int level = 0; level < buckets.length; level++) {
            if (deleteAtLevel(item, level)) {
                return true;
            }
        }

        return false;
    }
private boolean deleteAtLevel(String item, int level) {
```

```
        int i1 = hash1(item, level);
        int i2 = hash2(item, level);

        for (int j = 0; j < bucketSizes[level]; j++) {
            if (item.equals(buckets[level][i1][j])) {
                buckets[level][i1][j] = null;
                return true;
            }

            if (item.equals(buckets[level][i2][j])) {
                buckets[level][i2][j] = null;
                return true;
            }
        }

        return false;
    }
```

Code explanation

LCDF Class

The LDCF class implements the core functionality of the Logarithmic Dynamic Cuckoo Filter. We can find the following **attributes**:

- **buckets**: A 3D array representing multiple levels of buckets.
- **bucketSizes**: An array storing the size of buckets for each level.
- **maxKicks**: The maximum number of relocations (kicks) allowed before resizing.
- **random**: A random number generator for stochastic decisions.

And the following **methods**:

- **hash1** and **hash2**: Calculate bucket indices for an item.
- **resize**: Expands the filter when necessary.
- **insertAtLevel**and **insert**: Inserts an item into the filter at a specified level.
- **lookupAtLevel** and **lookup**: Check for the presence of an item.
- **deleteAtLevel** and **delete**: Remove an item from the filter.

Main class

The Main Class demonstrates a practical application of the LDCF, being these the main steps:

- Reads a genome file and a sequence to search.
- Generates kmers from the genome.
- Inserts kmers into the filter.
- Searches for the specified sequence within the filter.
- Measures and reports the execution time

In this case, we evaluate our program with three cases: a small, a medium and a large example, named like this due to the dataset and sequence length being used.

```
public class mainclass {

    public static String[] generateKMers(String genome, int k) {
        if (k <= 0) {
            throw new IllegalArgumentException("k must be a positive
integer");
        }
        if (genome == null || genome.length() < k) {
            throw new IllegalArgumentException("genome must be a
string of length at least k");
        }

        String[] kmers = new String[genome.length() - k + 1];

        for (int i = 0; i < genome.length() - k + 1; i++) {
            kmers[i] = genome.substring(i, i + k);
        }

        return kmers;
    }

    public static void main(String[] args) {
        // SMALL EXAMPLE
        // String genome =
"GTACGGTACTACTTACGCAGTGTACGATTACGCAGCTATACGCCGACT";
        // String sequence = "GTACGATTAC";
        // int[] kValues = {5, 10, 20, 30, 40, 50};

        // MEDIUM EXAMPLE
        String filePath =
"bioinformatics/datasets/synthetic_genome.fna";
        String sequence =
"TCGCGGGGTGTTAAACTGGGTTTAAATCTTGAAGTGTGATTCTAAGACTAACTAATCCCACTGGGCACA
AGGACC";

        // LARGE EXAMPLE (E.Coli example)
        // String filePath =
"bioinformatics/datasets/GCF_000008865.2_ASM886v2_genomic.fna";
        // String sequence =
"TCATTTGATCAGCAGTGATGGCGTAATTGTCATTAAGGCACAGGAATACCGCAGTCAGGAACTGAACCG
CGAAGCGGCGCTGGCCCGGCTGGTGGCAGTGATTAAAGATTTAACAACAGAACAAAAAGCCCGACGACCC
ACGCGGCCCACCCGTGCATCGAAAGAGCGCAGGCTGGCATCGAAAGCACAAAAATCAAGCGTGAAGGCGA
TGCGCAGCGGTCGGGAATAAAAAGAAGGAATGG";

        String genome = null;
        try {
            genome = new
String(Files.readAllBytes(Paths.get(filePath)));
        } catch (IOException e) {
            e.printStackTrace();
        }
        genome = genome.replaceAll(">.*\n", "").replaceAll("\n", "");
// Rremove FASTA header if present

        int size_sequence = sequence.length();
        int[] kValues = {size_sequence};

        System.out.println("Looking for this sequence: " + sequence);

        LDCF ldcf = new LDCF(1024, 4, 500);
```

```
        long startTime = System.currentTimeMillis();
        boolean exists = false;
        for (int k : kValues) {
            try {
                String[] kmers = generateKMers(genome, k);

                for (String kmer : kmers) {
                    ldcf.insert(kmer);
                    exists = ldcf.lookup(sequence);

                    if (exists) {
                        System.out.println("Sequence exists: " +
exists);
                        break;
                    }
                }

                if (exists) {
                    break;
                }

            } catch (IllegalArgumentException e) {
                System.out.println(e.getMessage());
            }
        }

        long endTime = System.currentTimeMillis();
        float duration = endTime - startTime;
        duration = duration / 1000;

        System.out.println("The program lasted " + duration + " s.");
    }
}
```

File Generator

A python script that creates a synthetical genome for evaluation purposes. Its lenght is randomly chosen between 10^3 and 10^7 characters.

Datasets folder

The folder where the datasets of genome sequences are placed. We already provide two examples: the E.Coli example and a syntethical generated dataset using the previous file generator.

Evaluation – Results

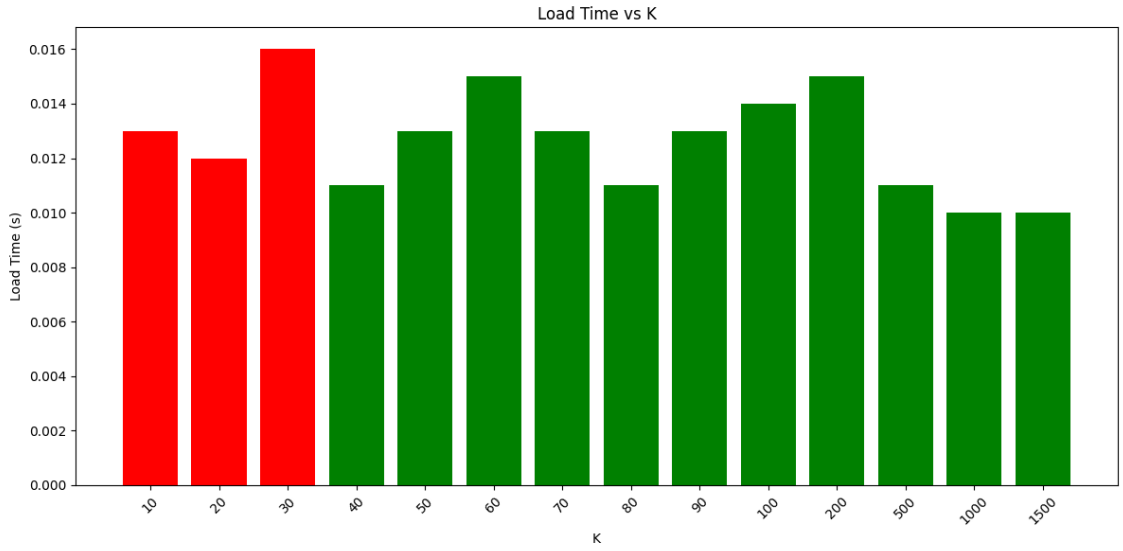
In order to evaluate the performance of our program, we have tested three examples of datasets with differents lengths of characters (nucleotides) and sequences to look for:

- synthetic_genome_medium_example1: length in between $10^2 - 10^4$
- synthetic_genome_medium_example2: length in between $10^4 - 10^6$
- synthetic_genome_medium_example3: length in between $10^5 - 10^7$

For each dataset, we have also tested the program with a range of different K-values, which is the length of sequences in which the genome is divided so as to insert the data in the buckets and compare with the sequence we are looking for. For the first example we have used the following K-values: {10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 500, 1000, 1500}. For the next two examples, due to their extended length compared to the former one, we also thought that it would be interesting to explore the K-values {2000, 2500} in addition to the others.

Then, after the evaluation, these are the graphs regarding the results:

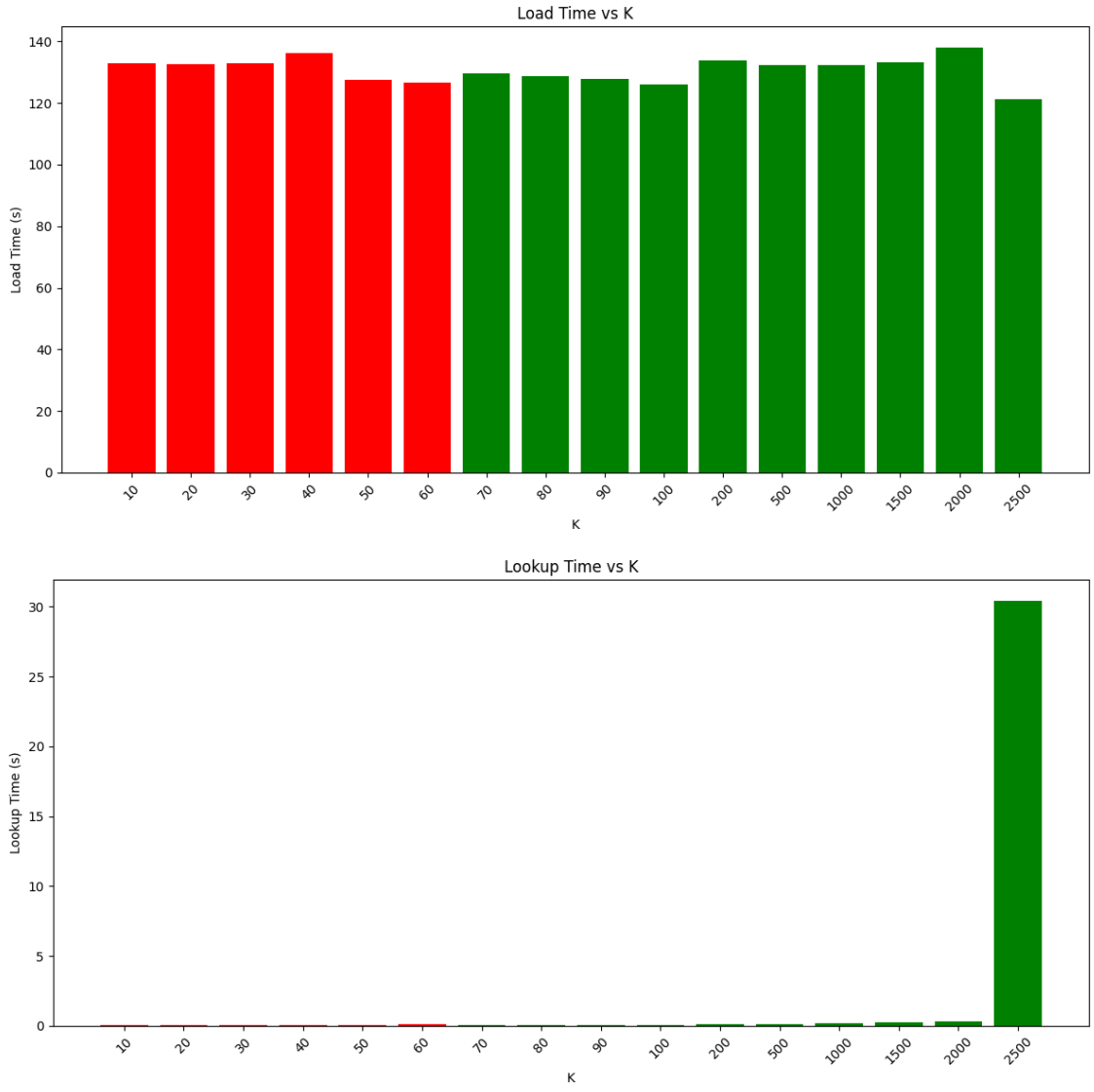
- **synthetic_genome_medium_example1: length in between $10^2 - 10^4$**



For comparison purposes, we have classified the results in **red** if the sequence could not be found, and **green** if they could be found. The reason of this is that the sequence that we were looking for in this case had a length of 40 characters, which is why our program only finds the sequence with K-values equal to or higher than 40 (as we expected it to be).

Regarding the time values scored in each K-value case, we see that the **load-time** is very low as the volume of data in this dataset is really low, and more even the **lookup time**. We also do not appreciate any specific change of time required for loading/lookin for the data with higher values of K.

- **synthetic_genome_medium_example2: length in between $10^4 - 10^6$**



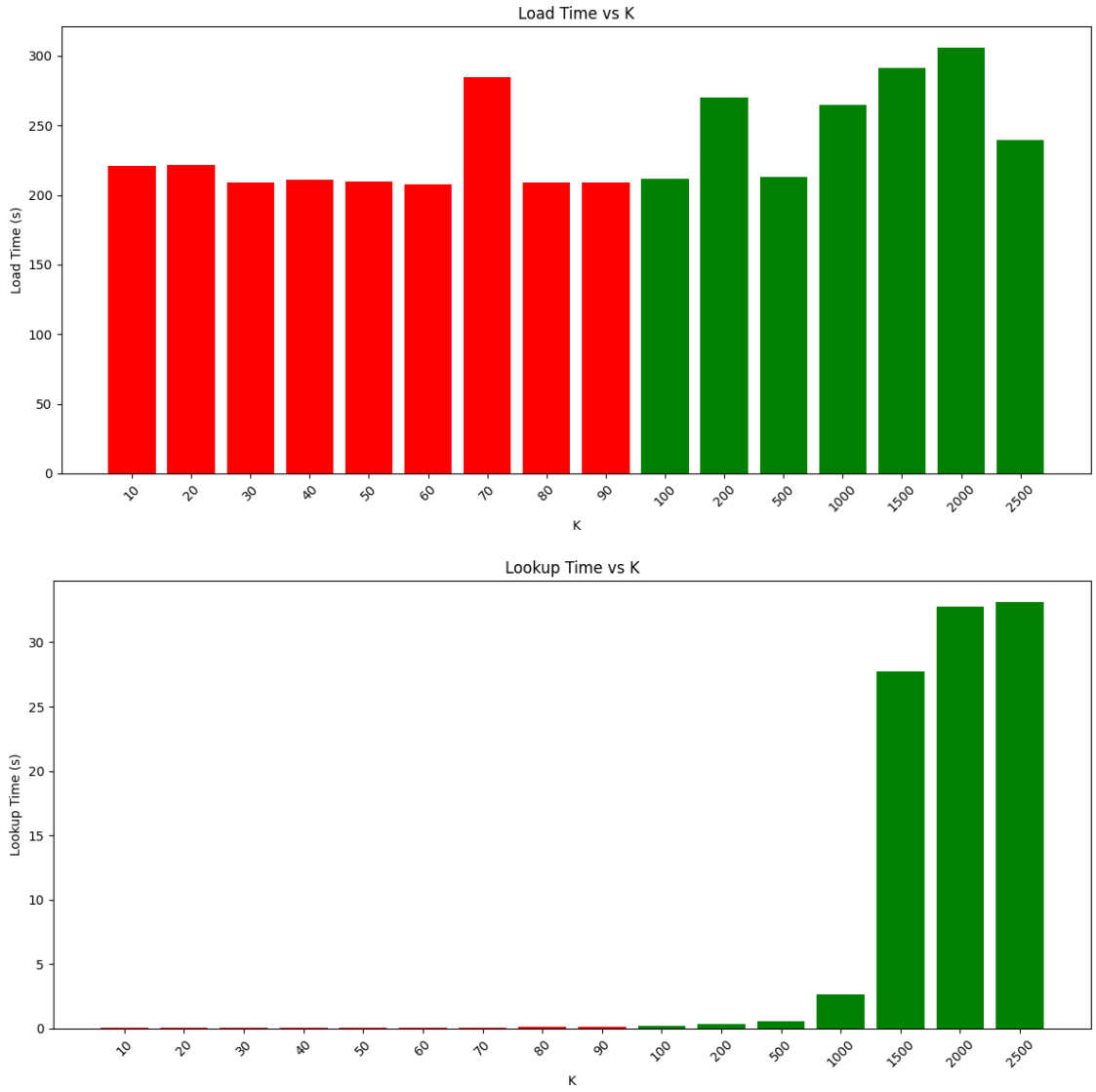
Given the prior results, we can see that again the **color** remain acording to our logic, as the sequence is only found when the K-values are equal to or higher than its length, which is of 70 this time.

In this case, we can see that the **load time** is significantly higher than the last one due to the length of the dataset, scoring times in between {120, 140} seconds.

Now, we can also appreciate a phenomenon that changes drastically in comparison to the former experiment: there is a case in the **lookup time** that suddenly changes drastically to 30 seconds, when for all the prior values of K we observe time values of {0.05, 0.4} seconds. We haven't been able to determine the reason behind this unexpected change

from 2000 to 2500 K-values, as it is an isolated case, but it always produced around the same result when we repeated the process.

- **synthetic_genome_medium_example3: length in between $10^5 - 10^7$**



In regard to the **color** of the bars, they are again correct as the sequence we were looking for in this case had a length of 100 characters, which is why it was not found in the prior cases.

About the **loading time**, it is scored between a range of {200,300} seconds, which makes sense as this dataset is longer than the second example. However, we can observe in the **lookup time** that there are significantly more cases when the time changes drastically comparing it to lower k-values. Then, we can conclude that there is a pattern in which for datasets with longer extension of data, the time that takes searching a sequence increases when approaching higher values of K.

As a **conclusion** of the previous evaluations, we concluded that the most efficient way of looking for a sequence is setting a K-value in the program of the same size than the

sequence that is being searched, as it is implied that lower K-values result in lower expected times for the search. Therefore, our implementation before the evaluation using different K-values actually had this characteristic, using the size of the sequence as the only K-value for performing the execution of the program.

Finally, before finishing this evaluation part we would like to point out that the official genome of the E. Coli was not used for performing these results due to its extension, which made it not possible for us to perform further experiments as the time it took was significantly longer than the previous synthetic generated genomes.

License

Installation

Clone the repository to your local machine:

```
git clone https://github.com/your-username/ldcf.git
cd ldcf
```

Contact

For any questions or suggestions, please open an issue or contact the repository owner. Thank you for using the Logarithmic Dynamic Cuckoo Filter!