Finding plagiarism with Counting Bloom Filters (CBFs)

Godson Ajodo

December 12, 2023

1 Counting Bloom Filters

1.1 Introduction

1.1.1 What is a Bloom Filter

A bloom filter is a space-efficient probabilistic data structure designed to test whether an element is a set member. It was conceived by Burton Howard Bloom in 1970 and is based on Hashing (Brilliant, n.d.). It is a "lightweight" of a hash table. Both hash tables and Bloom filters support efficient insertions and lookups. Bloom Filters are more space efficient than hash tables, but this comes at the cost of having "false positives" for entry lookup. False Positives in the sense that it would indicate an element has been inserted when it has, in fact not been.

Features of a Bloom Filter:

- Bit Array: An empty bloom filter is a bit array of m bits, all set to zero (GeeksforGeeks, n.d.). Each bit in the array represents a potential member of the set.
- Hash Functions: A bloom filter uses k hash functions to map input values to bit positions in the bit array (GeeksforGeeks, n.d.). These hash functions should be independent and uniformly distributed, with a range of 0 to m-1 (Brilliant, n.d.)
- Insertion: To add an element to the Bloom Filter, hash it k times and set the bits in the bit array at the index of those hashes to 1 (Limb, n.d.)
- Query: When querying the bloom filter, you can quickly check if an element is present in the set by looking at the corresponding bits in the bit array.

Working of Bloom Filters:

- Initialize the bloom filter: Create a bit array of length m, with all bits set to zero (Brilliant, n.d.)
- Add an element: To add an element to the bloom filter, perform the following steps for each hash function in the set:
  - Hash the element using the hash function which returns an index between 0 and m-1
  - Set the bit at the index identified by the hash function to 1 (SystemDesign.One, n.d.)
- Query the bloom filter: To check if an element is present in the set, perform the following steps for each hash function in the set:
  - Hash the element using the hash function, which returns an index between 0 and m-1
  - check the bit at the index identified by the hash function. If it is 1, the element is likely present in the set.

Differences between Bloom Filters and a Hash table: A Bloom filter and a hash table are both data structures used for efficient lookups, but they differ in several key aspects:

1. Purpose: Bloom Filters are used to test whether an element is a member of a set, while hash tables are used to store and retrieve key-value pairs. Bloom Filters are more focused on space efficiency, whereas hash tables prioritize fast access and accurate results
2. Space efficiency: Bloom Filters are more space-efficient than hash tables, as they require only a small amount of memory to represent a large set. In contrast, the hash table uses more memory to store the key-value pairs (Limb, n.d.)
3. Insertion: In a bloom filter, elements are added by hashing them and setting the corresponding bits in the bit array to 1. In a hash table, elements are stored in a bucket based on the hash function, and the value is associated with the key
4. Query: Bloom Filters provide a rapid and memory-efficient way to check if an element is present in the set (Limb, n.d.). Hash tables allow for faster lookups based on the hash function, but they do not provide the same level of space efficiency as Bloom Filters (Cloudflare, n.d.)
5. False positives: Bloom Filters have a small probability of false positives, meaning they may indicate that an element is present in the set when it is not. Hash tables, on the other hand, provide more accurate results with lower false positive rates
6. Updates: Hash tables support the insertion and deletion of elements, while bloom filters do not support element deletion.

1.1.2 Counting Bloom Filters (CBFs): A type of Bloom Filter

Counting Bloom Filters (CBFS) is a type of Bloom Filter that supports dynamic sets that can be updated via insertions and deletions (D'Angelo & Palmieri, n.d.). They use a bit array, where each bit can be set or cleared. However, instead of using a traditional Bloom Filter with binary values (0 or 1), a CBF allows multiple occurrences of an element by maintaining a counter at each bit position. This counter is incremented or decremented based on insertions or deletions. They implement fast-set representations to support membership queries with constant-time complexity (An, Ouyang, & Zhou, 2018).

CBFs use a hash table with k hash functions and m counters to store elements (Tarkoma, Rothenberg, & Lagerspetz, n.d.). The hash functions are applied multiple times as determined by the number of hash functions specified during CBF instantiation) to generate multiple indices for each element. A custom hash function is employed in the provided implementation of this paper.

The operations supported by CBFS include:

- Insertion: To insert an element x, CBF applies k hash functions to x and increments the corresponding counters in the hash table (Tarkoma, Rothenberg, & Lagerspetz, n.d.). The time complexity of insertion is O(k).

- Deletion: To delete an element x, CBF applies k hash functions to x and decrements the corresponding counters in the hash table (Tarkoma, Rothenberg, & Lagerspetz). The time complexity of deletion is also O(k).

- Search (or Membership query): To answer a membership query, CBF checks the k corresponding counters. If all of them are non-zero, CBF judges that the queried element is a member; otherwise, negative (An, Ouyang, & Zhou, 2018). The time complexity of the membership query or Search is O(k).

Practical Applications of CBFs: Counting Bloom Filters finds applications in scenarios where approximate set membership queries and dynamic datasets are prevalent, and also in cases where large datasets need to be processed efficiently. Some practical, real-world computational applications include:

- Caching Systems:
  - CBFs can be employed in Content Delivery Networks (CDNs) to cache and serve content efficiently. By storing the most frequently accessed content in the cache, CBFs can quickly serve requests, reducing the need to fetch content from the original source (Hal, n.d.).
  - it can also be used in service registries to manage a large number of services. These registries need to be scalable to handle the increasing number of services, and CBFs can help in efficiently managing and accessing the services (Cheng & Zhang, 2009).
  - "In low-power embedded processors, CBFs can be used to reduce the power consumption of the cache. In embedded systems, the cache can consume a significant portion of the overall power, so reducing its power consumption is essential." This helps the systems achieve lower power consumption while maintaining efficient cache functionality
- Weak Password detection: By maintaining a list of weak passwords in a Bloom Filter, systems can check if a new or updated password matches any of the weak passwords. If a match is detected, the user can be warned to choose a stronger password (OpenGenus IQ, n.d.)
  - Weak password finder tools, such as those for active directories, utilize counting Bloom Filters to analyze passwords against common dictionaries, check for password reuse across multiple accounts, and identify stored passwords using reversible encryption or legacy algorithms (Delinea, n.d.)
- DNA Sequence Analysis:
  - CBFs can be applied in genomics for filtering out known sequences, allowing researchers to focus computational resources on analyzing novel or rare genetic sequences.
  - It is widely used in DNA sequence analysis for efficient k-mer counting. They allow for the identification of k-mers that occur more than once in DNA sequence dataset while significantly reducing memory requirements (Melsted & Pritchard, 2011)
- Safe browsing in Google Chrome: Bloom Filters are employed in Google Chrome to enhance the security of the browser by detecting and blocking malicious websites.
  - Google Chrome's Enhanced Safe Browsing mode uses counting Bloom filters to detect and block malicious websites. When a user enables enhanced Safe Browsing, Chrome checks in real-time whether a site they are about to visit might be a phishing site. The CBF allows for the efficient storage and retrieval of information about potentially dangerous sites, providing a fast and effective means of identifying potential security risks.
- Wallet synchronization in Bitcoin: Bloom Filters are used in Bitcoin to detect if specific information will be deleted later, allowing the system to make informed decisions about data transfer and minimizing the risk of triggering DDoS attacks.
- Hash-based IP traceback: Bloom Filters can be used to perform cybersecurity tasks, such as tracking the origin of IP addresses in virus scanning.
  - Hash-based IP traceback is a technique that uses counting Bloom filters to trace the origin of a single IP packet delivered by the network in the recent past. The technique generates audit trails for traffic within the network, allowing for the identification of the source of an IP packet. By identifying the source of a packet it can help in identifying

  and mitigating potential security risks such as DDoS attacks.
  - The counting Bloom Filter allows for the efficient storage and retrieval of information about the packets, providing a fast and effective means of identifying the source of a packet (Snoeren, Partridge, & Sanchez, 2001)

Justification for their uses:

- CBFs provide a space-efficient way to approximate set membership queries with a controlled false positive rate.
- They are particularly useful in scenarios where memory constraints are a concern, and a small false positive rate is acceptable.
- The dynamic nature of CBFs, with support for insertions and deletions, makes them suitable for applications with dynamic datasets.

However, while they are useful, there are also some downsides to using CBFs. For example, in the Bitcoin wallet synchronization, CBFs may require the wallet to wait until a transaction is in a block, leading to slower synchronization times and potentially impacting the user experience (Blockchain Academy HS Mittweida, n.d.). Additionally, CBFS may have higher bandwidth requirements compared to other data structures like bloom filters and address-indexed servers.

2 Hash Functions

2.1 Definition and Purpose

Hashing functions are mathematical algorithms designed to convert variable-sized input data into a fixed-size string of characters, commonly known as a hash code. "The deterministic nature, computational efficiency, and desirable cryptographic properties of hash functions make them indispensable in various computational applications."

A hash function is a function that converts a given input into a fixed-size string of characters, typically used in various applications such as data integrity, digital signatures, and password storage. It is a fundamental tool in modern cryptography, ensuring data authenticity and integrity by producing a unique hash value for each input, making it useful for quickly confirming data tampering. There are different types of hash functions, each with its own unique features and applications, and the choice of a hash function depends on specific requirements such as speed, security, and memory usage. Cryptographic hash functions, in particular, are designed to be secure and withstand various cryptanalytic attacks. A good hash function should be efficiently computable and uniformly distribute the keys, with minimal collisions when placing records in the hash table (GeeksforGeeks, n.d.).

The ideal hash function has three main properties:

- It is extremely efficient to compute the hash value for any given input.
- It distributes the hash values uniformly across the hash table. This means that every possible output is equally likely, which helps in evenly distributing the keys.
- It minimizes collisions. A collision occurs when two different inputs produce the same output.

2.2 Hash Functions in Counting Bloom Filters

Hash functions play a crucial role in Counting Bloom Filters. They are used to map the set elements to counters in the filter. The choice of hash functions can greatly affect the performance of the Counting Bloom filter.

The hash functions used should have the properties of an ideal hash function as mentioned above. They should be efficient in computing, distribute values uniformly, and minimize collisions.

In addition, the hash functions should be independent of each other. This means that the output of one hash function should not affect the output of another hash function. This property ensures that the counters in the filter are incremented (or decremented) in a random and uniform manner.

2.2.1 Choice of Hash function:

I will be comparing two hash functions, one with minimum distribution and simple modular operations. The second hash function incorporates an extra layer of complexity where we try to consider the avalanche effect and try to minimize collisions as much as possible.

Modular Hash Functions: The `Modular_hash` function employs a modular hashing technique, which involves computing the hash value modulo a prime number (`prime_modulus`). This is done to ensure that the hash value remains within a specific range and helps reduce the likelihood of collisions.

- **Tokenization:** Before hashing, the input string is converted to lowercase using the `lower()` method. This ensures case-insensitive tokenization, meaning that strings with the same characters but different cases will produce the same hash value.
- **Looping:** The heart of the algorithm is the loop that iterates through each character of the lowercase string (`s_lower`). For each character, the hash value is updated using the formula:

  $$hash\_value = (hash\_value \times base + \text{ord}(char)) \mod prime\_modulus$$

  Additionally, a `base_power` variable is updated in each iteration, following a similar formula:

  $$base\_power = (base\_power \times base) \mod prime\_modulus$$

Strengths

- **Sensitivity to Order:** The hash function is sensitive to the order of characters in the input string. Even a small change in the order of characters will likely result in a significantly different hash value. This property is desirable for hash functions used in applications such as hash tables.
- **Prime Modulus:** The use of a prime modulus enhances the function's ability to distribute hash values uniformly. This helps reduce the occurrence of collisions, where different inputs produce the same hash value.
- **Customization:** The function allows for customization through the parameters `base` and `prime_modulus`. Users can experiment with different values to tailor the hash function to their specific needs, balancing performance and distribution characteristics.

Weaknesses

- **Limited Collision Resistance:** While the prime modulus enhances distribution, the function might still suffer from collisions, especially when dealing with a large number of inputs. Collisions occur when different inputs produce the same hash value, and they can have implications for the efficiency of hash-based data structures.
- **Sensitivity to String Length:** The hash value is influenced by the length of the input string. Strings of different lengths may produce different hash values even if their contents are similar. This could potentially lead to uneven distribution in hash tables.
- **Limited Security:** The hash function is designed for general-purpose hashing and is not suitable for cryptographic applications. It lacks the security features required to resist attacks such as collision attacks or pre-image attacks.

Statistical Insights

- **Distribution Uniformity:** The distribution of hash values is influenced by the prime modulus. Generally, the use of a prime modulus helps achieve a more uniform distribution, reducing the likelihood of clustering and improving the performance of hash tables.
- **Performance:** The performance of the hash function is influenced by factors such as the string length and the chosen values for base and prime modulus. While the function provides a reasonable trade-off between simplicity and performance, users should carefully select parameters based on their specific use cases.

Algorithm 1 Custom Hash Function (Modular Arithmetics)

```
function CustomHash($s, base, prime\_modulus$)
    $hash\_value \leftarrow 0$
    $base\_power \leftarrow 1$
    $s\_lower \leftarrow$ ToLower($s$)
    for all $char \in s\_lower$ do
        $hash\_value \leftarrow (hash\_value \times base + \text{Ord}(char)) \bmod prime\_modulus$
        $base\_power \leftarrow (base\_power \times base) \bmod prime\_modulus$
    end for
    return $hash\_value$
end function
```

Non-Cryptographic Custom Hash functions: To go about this, we would need to understand the way popular non-cryptographic hash functions work. Instead of feeding all the inputs the way they appear, we do some form of preprocessing. The preprocessing is necessary so that we can compress the input data into a n-bit block (but in our case, n = 32). If we don't do any form of preprocessing and use their corresponding ASCII value (each character is represented using 8 bits), the hash function would likely operate on a per-byte basis, which means the hash function processes the data in chunks of 8 bits at a time. For inputs like 'ABC', we would have something like '656667.' With more increments in the length, we would have exponential representations of the input, which is a very bad architecture.

So, we would have a container of 32-bit bit array space, and then we apply some form of cleverness (mixing in this case). The 'cleverness' here is a combination of XOR and multiplication operations with constants, designed to spread the influence of individual bits across the entire hash value. This cleverness helps our hash function to meet criteria like Uniform Distribution, the Avalanche effect (slight changes like 'ABC' and 'ABD' would have big differences in their hash values), deterministic, and efficiency.

Constants and Variables:

- **h (hashing key):** It represents the current hash value. The function updates this variable throughout the hashing process.
- **a, b, c,d (mixing constants):** These constants are used for mixing operations. Choosing appropriate constants is crucial for achieving good distribution and avalanche effect in the hash function.
  - Our constants are a mixture of prime numbers that are large and also numbers that have a good mix of 0s and 1s in their binary representation to help achieve a strong avalanche effect. Here are the four constants chosen for this purpose:
  -
    * a = 0xB7E15163 (3074997963): This is a constant used in the MurmurHash algorithm, known for its good distribution properties.
    * b = 0x8AED2A6B (2334415563): This large number also has a diverse bit pattern.
    * c = 0xD72A7A45 (3597765701): Another large number with a good mix of bits.
    * d = 0x9E3779B1 (2654435761): This is the golden ratio prime, often used in hash functions for its excellent distribution characteristics.
- **total_bytes:** It stores the total number of bytes in the input data.
- **remaining_bytes:** It calculates the number of bytes that are not divisible by 4. This is important for handling the last block of data that may not be a complete 32-bit block.
- **total_32bit_blocks:** It calculates the total number of complete 32-bit blocks in the data.

Mixing Function: The mix function takes a 32-bit chunk as input and performs a mixing operation on the current hash value h. The XOR and multiplication operations with mixing constants are designed to distribute the influence of individual input bits across the entire hash value.

Processing 32-bit Blocks: The function processes the input data in 32-bit blocks, converting each block to an integer and applying the mix function. This step ensures that the hash function considers each part of the input data while updating the hash value.

Processing Remaining Bytes: If the length of the input data is not a multiple of 4, there will be remaining bytes. The function pads the remaining bytes with zeros to create a complete 32-bit block and processes it using the mix function.

Test against Properties of Hash Functions:

- **Avalanche Effect:** The hash function is designed to ensure that a small change in the input (even a single bit) results in a significant and unpredictable change in the output hash. This is achieved through a combination of XOR, multiplication, and bit-shifting operations with the selected prime constant. These operations, especially in combination with large primes, help in dispersing the input bits across the entire hash value.
- **Uniform Distribution:** The constants and operations are designed to provide a uniform distribution of hash values. This is essential for preventing collisions and ensuring that the hash function is not biased toward certain inputs.

- **Deterministic:** The function is deterministic, meaning that the same input will always produce the same hash value.
- **Efficient:** The function processes data in 32-bit blocks, making it efficient for practical use.
- **Collision Resistance:** While the function seems well-designed, actual collision resistance would require extensive testing and analysis, and it's recommended to subject the function to various test cases to evaluate its collision resistance properties.

Time and Space Complexity Analysis

In this analysis, the following notation is used:

- n: The size of the bit array in the `bitarray` class.
- k: The size of the data (in bytes) being processed by the hash functions.
- m: The size of the bit array in the `CountingBloomFilter` class.
- h: The number of hash functions used in the `CountingBloomFilter`.
- w: The number of words (or items) being inserted or searched in the `CountingBloomFilter`.

We analyze the time and space complexity of the methods in the `CountingBloomFilter` and `bitarray` classes.

bitarray Class Methods

- **__init__**:
  - Time Complexity: $O(n)$
  - Space Complexity: $O(n)$
- **set_bit**:
  - Time Complexity: $O(1)$
  - Space Complexity: $O(1)$
- **get_bit**:
  - Time Complexity: $O(1)$
  - Space Complexity: $O(1)$
- **__repr__**:
  - Time Complexity: $O(n)$
  - Space Complexity: $O(n)$

CountingBloomFilter Class Methods

- **__init__**:
  - Time Complexity: $O(1)$
  - Space Complexity: $O(m)$
- **custom_hash**:

    - Time Complexity: $O(k)$
    - Space Complexity: $O(1)$
- **hash_cbf**:
    - Time Complexity: $O(k \cdot h)$
    - Space Complexity: $O(h)$
- **search**:
    - Time Complexity: $O(k \cdot h)$
    - Space Complexity: $O(h)$
- **insert**:
    - Time Complexity: $O(k \cdot h)$
    - Space Complexity: $O(h)$
- **delete**:
    - Time Complexity: $O(k \cdot h)$
    - Space Complexity: $O(h)$
- **insert_words**:
    - Time Complexity: $O(w \cdot k \cdot h)$
    - Space Complexity: $O(h)$
- **is_word_present**:
    - Time Complexity: $O(k \cdot h)$
    - Space Complexity: $O(h)$

Counting Bloom Filter Analysis

Relationship Between Parameters The false positive rate (FPR), memory size, number of hash functions, and the number of items inserted are interrelated.

Definitions: Let us define:

- n: Number of items to be inserted.
- m: Number of bits in the Bloom Filter (memory size).
- k: Number of hash functions.
- p: False positive rate.

Algorithm 2 Custom Hash Function (non-cryptographic hash function)

```
1: $h \leftarrow seed$
2: $a \leftarrow 3074997963$
3: $b \leftarrow 2334415563$
4: $c \leftarrow 3597765701$
5: $d \leftarrow 2654435761$
6: $total\_bytes \leftarrow \text{LENGTH}(data)$
7: $remaining\_bytes \leftarrow total\_bytes \bmod 4$
8: $total\_32bit\_blocks \leftarrow \frac{total\_bytes - remaining\_bytes}{4}$

9: function Mix($chunk$)
10:     $h \leftarrow h \oplus chunk$
11:     $h \leftarrow (h \times a) \oplus (h \gg 9)$
12:     $h \leftarrow (h \times b) \oplus (h \ll 11)$
13:     $h \leftarrow (h \times c) \oplus (h \gg 13)$
14:     $h \leftarrow (h \times d) \oplus (h \ll 15)$
15:     $h \leftarrow h$ AND $0xFFFFFFFF$
16: end function

17: for $block = 0$ to $total\_32bit\_blocks - 1$ do
18:     $chunk \leftarrow$ convert to integer from bytes($data[block \times 4 : (block + 1) \times 4]$, byteorder = ı$little$ȷ)
19:     Mix($chunk$)
20: end for
21: if $remaining\_bytes > 0$ then
22:     $remaining\_chunk \leftarrow$ convert to integer from bytes($data[-remaining\_bytes :] +$ “\x00” $\times$
    $(4 - remaining\_bytes)$, byteorder = ı$little$ȷ)
23:     Mix($remaining\_chunk$)
24: end if
25: $h \leftarrow h \oplus total\_bytes$
26: $h \leftarrow h \oplus (h \gg 16)$
27: $h \leftarrow (h \times 0x85EBCA6B)$ AND $0xFFFFFFFF$
28: $h \leftarrow h \oplus (h \gg 13)$
29: $h \leftarrow (h \times 0xC2B2AE35)$ AND $0xFFFFFFFF$
30: $h \leftarrow h \oplus (h \gg 16)$
        return $h$
```

Memory Size (m)

The memory size for a Counting Bloom Filter depends not only on the number of items and the desired false positive rate but also on the counter size. The approximate size is given by:

$$m = -\frac{n \ln p}{(\ln 2)^2} \cdot b \tag{1}$$

where b is the bit width of the counters.

Number of Hash Functions (k)

The optimal number of hash functions in a Counting Bloom Filter is similar to that in a standard Bloom Filter:

$$k = \frac{m}{n \cdot b} \ln 2 \tag{2}$$

False Positive Rate (p)

The false positive rate in a Counting Bloom Filter is more complex to calculate due to the counters. However, a simplified approximation can be:

$$p \approx \left(1 - e^{-\frac{kn}{m/b}}\right)^k \tag{3}$$

This equation shows the trade-off between memory usage (counters and their sizes) and accuracy.

Mathematical Proof of False Positive Rate for Counting Bloom Filter

The proof for the false positive rate in a Counting Bloom Filter is more complex than that of a standard Bloom Filter due to the introduction of counters. However, there is an increase in the probability of a counter being incremented with each insert operation.

For an element not in the set, the false positive occurs if all k hash functions point to counters that have been incremented by previous insertions. The probability of this occurring is affected by the bit width of the counters and the distribution of the hash functions.

2.2.2 Effectiveness of the CBF implementation

Memory Size Scaling with FPR and Number of Items

- **FPR:** The memory size of a Counting Bloom Filter is inversely proportional to the desired FPR. The lower the FPR, the more bits are needed per item to maintain this rate, leading to a larger memory size. Mathematically, the size m is often approximated as $m = \frac{-nln(p)}{(ln(2))^2}$, where n is the number of items and p is the FPR
- **Number of Items:** The memory size also scales linearly with the number of items. As more items are added, more bits are required to maintain the same level of FPR.

Actual FPR Scaling with the number of hash functions

- The actual FPR of a CBF is a complex function of the number of hash functions k, the number of bits m, and the number of items n. However since the actual FPR does not depend on the number of hash functions alone, we don't have a definite relationship between the two, especially since FPR can increase due to increased collisions.

Access Time Scaling with Number of Items Access time in a CBF, for both insertion and search operations, scales with the number of hash functions. As more items are added, the probability of collisions increases, which can increase the time it takes to check the confirm the presence of an item. The access time per item is expected to be relatively constant for a well-distributed hash function.

Initially, we planned to compare the CBF using two different hash functions.

We discover that with a simple modular hash function, which hashes the input in linear time, there are fewer complexities that are absent in the complex custom hash function. We see that there will be a higher case of collision and so the false positive rate tends to remain constant at some point(as seen in figure 1. Compared with the custom complex hash function whose collisions don't converge (figure 2), we have a better distribution of our counting bloom filter in terms of the presence and absence of values.

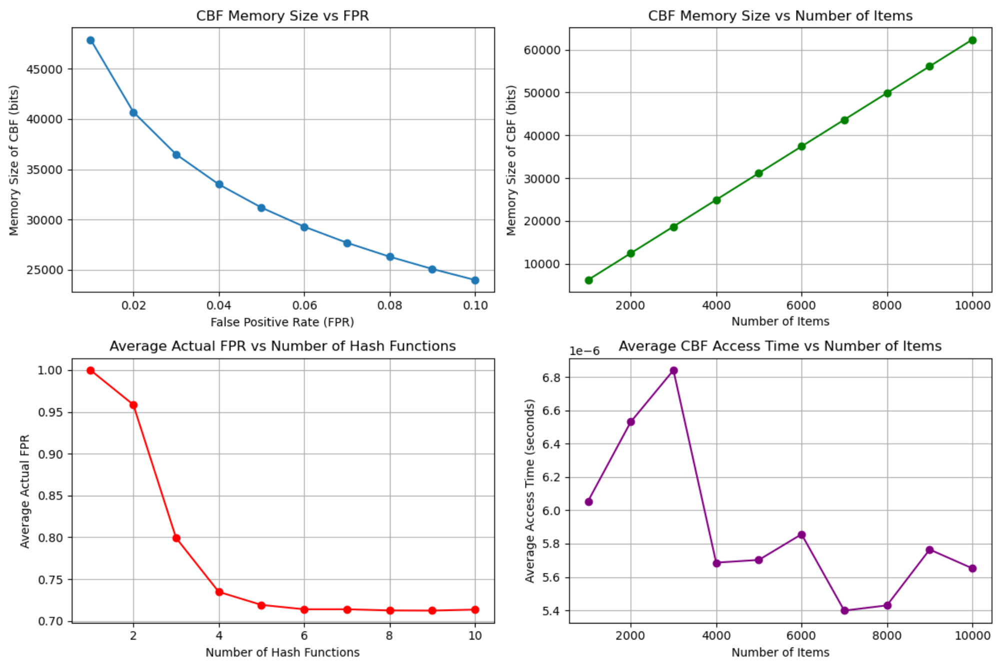

Figure 1: Graphs showing the relationship between the different parameters: Memory size, FPR, number of hash functions and number of items) when the simple modular arithmetic hash function is used

So because of the non-converging collision, we will proceed with using the implementation with the custom complex hash function for better distribution.

2.3 Practical Considerations

Real-World Data: Real-world data might not always follow theoretical models perfectly due to factors like non-ideal hash functions, data patterns, etc.

Hash Functions: The choice and quality of hash functions have a significant impact on both FPR and access times.

Counting Bloom Filter Specifics: In CBFs, each bit is replaced with a counter, which can affect

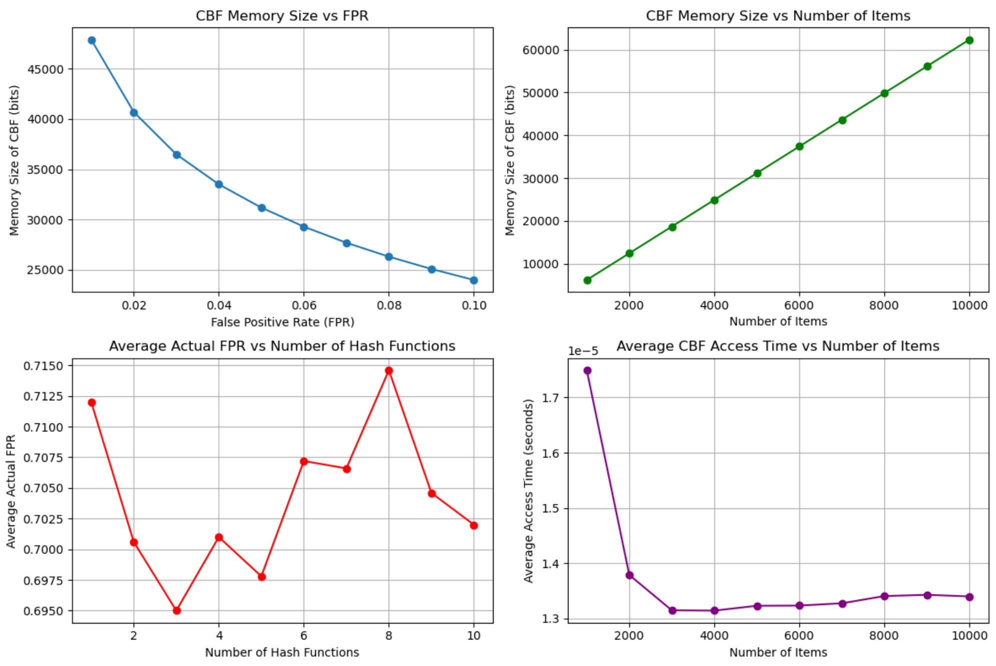

Figure 2: Graphs showing the relationship between the different parameters: Memory size, FPR, number of hash functions and number of items) when the complex non-cryptographic hash function is used

memory and access time analyses, especially if counters grow large enough to require more space.

3 Applications of Counting Bloom Filters in Plagiarism Detection

Counting Bloom Filters, in combination with hash functions, has proven to be a valuable tool for plagiarism detection. By efficiently encoding and storing large amounts of data, they enable the identification of similarities between documents. This has important implications for the field of plagiarism detection, as it allows for the detection of copied or paraphrased content. Counting Bloom Filters excel in this application due to their ability to provide probabilistic answers without the need to store the entire database (Geravand S. et. al. (2014)).

Additionally, counting Bloom Filters can be used in data leak protection applications, including internet plagiarism detection and filtering of web addresses (Academia (n.d.))

Their advantages include space efficiency and fast query processing, making them a suitable choice for large-scale plagiarism detection systems. However, it is essential to recognize their limitations, including false positives and the potential for information loss. Despite these drawbacks, the use of Counting Bloom Filters for plagiarism detection has shown promising results and continues to be an area of active research and development.

3.1 Implementation in Plagiarism Detection:

Defining plagiarism between two texts involves identifying significant similarities in content, structure, or expression that suggest one text may have been copied or closely derived from another without proper attribution. Plagiarism detection algorithms typically look for matching phrases, sentences, or structural elements to determine the likelihood that one text has been derived from another.

In my algorithm, plagiarism is defined as the occurrence of similar sets of words (defined by the window size) in different text versions. I first process version 1 of the text, using the Rolling Hash to generate hashes for each set of words and inserting these hashes into the CBF. Then, I process version 2 in a similar way, but instead of inserting the hashes into the CBF, I check if they are already present. If a hash from version 2 is found in the CBF, it indicates that the corresponding set of words also appears in version 1, suggesting potential plagiarism.

The plagiarism score is then calculated as the proportion of matching sets of words to the total number of sets in version 2.

3.1.1 Key Aspects of My Algorithm:

Window Size: Determines how many consecutive words are considered a set. A smaller window size may increase sensitivity but can lead to higher false positives.

Counting Bloom Filter: Allows for efficient storage and retrieval of hash values, while handling the possibility of hash collisions.

Rolling Hash: Provides an efficient way to generate hash values for consecutive sets of words.

Custom Hash Function: Ensures a unique and uniform distribution of hash values.

False Positive Rate (FPR): While the CBF reduces the likelihood of false positives, there's still a non-zero chance of them occurring due to the probabilistic nature of the Bloom Filter.

3.1.2 Document Representation:

CBFs can be employed to create compact representations of documents using hash functions.

Hash functions generate indices in the CBF, and the corresponding counters are incremented to represent the presence of specific features (n-grams, words, etc.) in the document. However, before generating indices in the CBF, some form of preprocessing would have to be done. This preprocessing could include removing stop words (like ‘and’, ‘or’, ‘to’, ‘in’, ‘on’, etc.), and punctuations and conversion of all entries into lowercase The reason for conversion into lowercase is to consider words like ‘dragon’ and ’Dragon’ as basically the same thing because the ASCII value of ’D’ differs from that of ’d,’ which would greatly affect our returns from our hash functions.

3.1.3 Similarity Comparison:

Rabin-Karp Algorithm The Rabin-Karp algorithm is a string-searching algorithm ideal for matching patterns. It uses a rolling hash to quickly filter out positions of the text that cannot match the pattern, thus being efficient for multiple pattern searches (Brilliant.org, n.d.).

Methodology

Algorithm Design For our plagiarism checker, I will be combining the counting bloom filter and a rolling hash mechanism inspired by the Rabin-Karp algorithm. The process involves two main phases: indexing and comparison.

1. **Building the Filter:** Each unique hash value generated from the rolling hash over the first text (representing a unique sequence of words) is stored in the Counting Bloom Filter.
2. **Checking for Plagiarism:** For the second text, the rolling hash is computed similarly. Each hash value is then checked against the Counting Bloom Filter. If a match is found, it suggests that a similar word sequence exists in both texts.

How Rolling Hash Works In the context of our plagiarism checker:

- **Initialization:** A window of a fixed size (number of words) is considered. This window size is crucial as it defines the granularity of comparison.
- **First Hash Computation:** The hash value for the initial window of text is computed. This hash is a function of all the characters (or words) within the window. In our case, it's a simple sum, but more complex functions can be used for better distribution.
- **Sliding the Window:**
  - **Append:** When the window slides one word to the right, the hash is updated by adding the value of the new word.
  - ***Remove:*** Simultaneously, the value of the word that's no longer in the window is subtracted.
- **Efficiency:** This method is efficient because it avoids recomputing the hash from scratch for every new window. Only incremental updates are done based on the entering and exiting elements.

Mathematical Description Let's denote the hash function as H, and the words in the window as $w_1, w_2, ..., w_3$. The base hash function could be as simple as summing the ASCII values of all characters in the words, but I consider words as atomic units on their own.

The initial hash value for a window is:

$H(w_1, w_2, ..., w_n)$

When the window slides to the right, incorporating a new word w_{new} and excluding the old word w_{old}, the updated hash value is:

$H' = H + H(w_{\text{new}}) - H(w_{\text{old}})$

This method allows for constant-time updates to the hash value, a significant improvement over recalculating the hash for a new window.

Plagiarism Scoring The plagiarism score is calculated as the percentage of matching word windows from the second text found in the first text. This score reflects not only the presence of similar words but also the sequence in which they appear.

Analysis

Tuning Parameters The choice of window size and the hash function is critical. A larger window size might miss finer-grained similarities, while a too-small window might lead to higher false positives. The hash function should balance between being fast to compute and providing a good distribution of hash values.

Strength

- **Sensitivity to Word Order:** This approach is sensitive to the order of words, a significant advantage in plagiarism detection where not just the words used but their sequence matters.
- **Scalability:** The CBF is space-efficient, making the algorithm scalable to large texts.
- **Speed:** The constant update of the hash value makes the algorithm fast, and particularly suitable for long texts.
- **Flexibility:** Parameters like window size and false positive rate can be adjusted based on the needs.
- **Dynamic:** The hash value changes as the window moves, even if the new window contains the same data but in a different order.

Limitations

- **False Positives:** Due to the probabilistic nature of the CBF, there's a non-zero chance of false positives, particularly if the window size is large or the hash function isn't well-distributed.
- **Hash Collisions:** The rolling hash may generate the same hash for different word windows, leading to potential inaccuracies.
- **Complexity:** The algorithm is more complex than simple text comparison methods, which might impact performance and require more computational resources.

3.1.4 Character-based or Word-based sliding window for the RollingHash function

Choosing between a character-based or word-based sliding window for the rolling function in a plagiarism detection context depends on the specific requirements and characteristics of your application. Both approaches have their advantages and disadvantages:

Character-Based Sliding Window

Advantages

- **Finer Granularity:** It offers more granularity and can detect even small sequences of copied text, making it more sensitive to minor similarities.
- **Language Agnostic:** Works well with those that don't use spaces to separate words or have complex word structures.
- **Flexibility:** More effective in detecting rearranged or slightly modified text sequences.

Disadvantages

- **Higher False Positives:** This may result in more false positives, especially in cases where common phrases or idioms are used.
- **Computational Intensity:** More computationally intensive due to processing at the character level.
- **Context Sensitivity:** It might miss the broader context or meaning of the text, focusing only on the sequence of characters.

Word-Based Sliding Window

Algorithm 3 Character-Based RollingHash

```
Class CharBasedRollingHash
  Method Initialize(base, window_size)
    base ← base
    window_size ← window_size
    hash_value ← 0
    base_pows ← [1]
    hash_mod ← large prime number (e.g., 1000000007)
    window ← empty list
  Method append(char)
    hash_value ← (hash_value × base + ASCII value of char) mod hash_mod
    Add char to window
    Add (last element of base_pows × base) mod hash_mod to base_pows
    if length of window > window_size then
      Call skip()
  Method skip()
    old_char ← remove first element from window
    old_char_val ← ASCII value of old_char × base_pows[window_size] mod hash_mod
    hash_value ← (hash_value - old_char_val + hash_mod) mod hash_mod
  Method current_hash()
    return hash_value
End Class
```

Advantages

- **Contextual Analysis:** Better at understanding the context and meaning of the text, as it considers whole words.
- **Efficiency:** Less computationally intensive compared to character-based analysis since there are fewer elements to process.
- **Reduced False Positives:** Lower chance of false positives, as it requires more substantial and meaningful matches.

Disadvantages

- **Less Granular:** Might miss small instances of plagiarism where only a few words are copied or slightly altered.
- **Language Limitations:** Not as effective for languages that don't clearly separate words or for texts with creative formatting.
- **Rigidity:** Less effective in detecting rearranged or subtly modified text.

If the goal is to detect even the slightest similarity or if the texts don't rely heavily on word separation, a character-based approach might be more suitable.

If the focus is more on the overall context and meaning, and computational efficiency is a concern, a word-based approach would be better.

In most academic or formal contexts, a word-based approach is often sufficient and more efficient, as it captures significant copying while reducing the noise of common phrases or structures. However, for a more nuanced or sensitive detection (like in legal or highly creative texts), a character-based

Algorithm 4 Word-Based Rolling Hash Function

```
Class RollingHash:
function $_{init(}$BASE, WINDOW_SIZE)    SELF.BASE $\leftarrow$ $base$
    $self.window\_size \leftarrow window\_size$
    $self.hash\_value \leftarrow 0$
    $self.base\_pow \leftarrow 1$
    $self.hash\_mod \leftarrow 1000000007$
    $self.window \leftarrow$ EMPTY LIST
    FOR _ IN RANGE $(window\_size - 1)$ DO
        $self.base\_pow \leftarrow (self.base\_pow \times self.base) \mod self.hash\_mod$
    END FOR

    FUNCTION APPEND($word$)
        $self.hash\_value \leftarrow (self.hash\_value \times self.base + \text{ORD}('|')) \mod self.hash\_mod$  ▷ Separator
        FOR $char$ IN $word$ DO
            $self.hash\_value \leftarrow (self.hash\_value \times self.base + \text{ORD}(char)) \mod self.hash\_mod$
        END FOR
        $self.window$.APPEND($word$)
    END FUNCTION
    FUNCTION SKIP
        IF $self.window$ IS NOT EMPTY THEN
            $old\_word \leftarrow self.window.\text{POP}(0)$
            $word\_pow \leftarrow 1$
            FOR $char$ IN $old\_word$ DO
                $self.hash\_value \leftarrow (self.hash\_value - \text{ORD}(char) \times word\_pow) \mod self.hash\_mod$
                $word\_pow \leftarrow (word\_pow \times self.base) \mod self.hash\_mod$
            END FOR
            $self.hash\_value \leftarrow (self.hash\_value - \text{ORD}('|') \times word\_pow) \mod self.hash\_mod$  ▷ Separator
        END IF
    END FUNCTION
    FUNCTION SLIDE($new\_word$)
        IF LENGTH OF $self.window \geq self.window\_size$ THEN
            SKIP
        END IF
        APPEND($new\_word$)
    END FUNCTION
    FUNCTION CURRENT_HASH
        RETURN $self.hash\_value$
    END FUNCTION
    FUNCTION RESET
        $self.hash\_value \leftarrow 0$
        $self.base\_pow \leftarrow 1$
        $self.window \leftarrow$ EMPTY LIST
    END FUNCTION
      =0
```

approach might be necessary despite its higher computational demands.

So for this paper, I will continue with the word-based approach.

3.2 Combination of the Counting Bloom Filter and the Rolling Hash Function for Plagiarism Checker

3.2.1 Custom Hash function in CountingBloomFilter

The custom hash function for my Counting Bloom Filter is a non-cryptographic hash function suitable for general hash-base lookup. it is used for its distribution properties, ensuring an even spread of hash values across the Bloom Filters. It's efficient for hashing entire, discrete pieces of data.

The rolling hash is optimized for scenarios where you need to continuously update hash values with minimal computational overhead, as in checking for matching sequences in text plagiarism.

The custom hash function treats the entire input as a single block, so rearranging the input data results in a completely different hash. This property is not an issue for the CBF. It is only concerned with presence or absence, but it becomes a problem for checking plagiarism since plagiarism doesn't entail the presence of words in a sentence but also how similar the texts' order is. The rolling hash helps in this case, as it is explicitly designed to be sensitive to the order of data. It allows us to detect not only whether a specific word is present but also if it appears in a particular sequence.

In summary, our custom hash function is used in the initial construction of the CBF to store hashes of individual words or chunks of text while rolling hash is used to efficiently process and compare large texts for similar sequences, making it highly suitable for detecting ordered sequences in text.

3.2.2 Workflow

Text Processing (Source Text):

- Use the Rolling Hash to generate hashes for each word window in the source text.
- Pass each Rolling Hash output through our custom hash function before inserting it into the Counting Bloom Filter.

Text Checking (Target Text):

- Similarly, generate hashes for each word window in the target text using the Rolling Hash.
- Use our custom hash function to process these hashes and then check them against the Counting Bloom Filter.

Plagiarism Detection:

- If the Bloom Filter indicates the presence of a hash from the target text (which represents a sequence of words), it suggests a potential match with the source text.
- Accumulate these matches to compute a plagiarism score.

3.2.3 Plagiarism Result for the shakespeares work

Parameters and Effects:

Algorithm 5 Plagiarism Check Complex

```
1: function PLAGIARISM_CHECK_COMPLEX(version1, version2, num_items, fpr, window_size)
2:     $cbf \leftarrow$ new CountingBloomFilter($num\_items, fpr$)
3:     $rh \leftarrow$ new RollingHash($256, window\_size$)
4:     for $i \leftarrow 0$ to length($version1$) $- \, window\_size$ do
5:         $window\_words \leftarrow$ list of words from $version1[i]$ to $version1[i + window\_size - 1]$
6:         rh.slide(join($window\_words$))
7:         $hash\_value \leftarrow$ rh.current_hash()
8:         cbf.insert(str($hash\_value$))
9:         rh.reset()
10:    end for
11:    $match\_count \leftarrow 0$
12:    $plagiarized\_sequences \leftarrow$ new list
13:    for $i \leftarrow 0$ to length($version2$) $- \, window\_size$ do
14:        $window\_words \leftarrow$ list of words from $version2[i]$ to $version2[i + window\_size - 1]$
15:        rh.slide(join($window\_words$))
16:        $hash\_value \leftarrow$ rh.current_hash()
17:        if cbf.is_word_present(str($hash\_value$)) then
18:            $match\_count \leftarrow match\_count + 1$
19:            Add join($window\_words$) to $plagiarized\_sequences$
20:        end if
21:        rh.reset()
22:    end for
23:    $plagiarism\_score \leftarrow (match\_count / \max(1, \text{length}(version2) - window\_size)) \times 100$
24:    return $plagiarism\_score$
25: end function=0
```

Increased Window Size (3): The window size is 3, meaning the algorithm compares sets of three consecutive words between texts.

Increasing the window size generally reduces sensitivity to common phrases and short coincidences, leading to a decrease in plagiarism scores unless there are exact matches of longer phrases.

The scores are much lower than with a window size of 2, indicating that exact matches of three-word sequences are less frequent in the texts.

Higher False Positive Rate (0.01): The false positive rate 1%. This higher rate means there's a greater chance of the algorithm incorrectly identifying a word triplet as plagiarized when it's not.

However, the low scores suggest that even with this higher false positive rate, the instances of three-word sequence overlaps are quite rare in the datasets.

Interpreting the Scores: Version 1 and 2 (0.437%): This score is very low, suggesting minimal similarity. Only about 0.4% of the three-word sequences in Version 2 are also in Version 1.

Version 2 and 3 (0.854%): This is slightly higher but still indicates a very low level of similarity.

Version 1 and 3 (0.444%): Similar to Version 1 and 2, indicating minimal overlap.

Version 3 and 2 (0.697%), Version 3 and 1 (0.544%), Version 2 and 1 (0.579%): These scores are all under 1%, pointing to very low levels of similarity.

With a threshold of 60%, we can conclude that the versions are not plagiarised

Potential Influences:

Distinct Content: The low scores across all comparisons suggest that the texts are quite distinct from each other, especially in terms of three-word sequences.

Specificity of Matches: With a window size of 3, only specific and longer matching sequences are identified, which reduces the likelihood of coincidental matches.

Effect of FPR: Despite the increased false positive rate, the impact seems minimal, as indicated by the low scores.

Side Notes: The result from the plagiarism check with the modular arithmetic hash function implemented shows a higher propensity for false positives and collisions. This hints at a preference of the complex non-cryptographic hashing. [1] For instance, the plagiarism scores are as follows:

- Plagiarism Score between Version 1 and 2: 1.7023288804823264%
- Plagiarism Score between Version 2 and 3: 1.9761459307764264%
- Plagiarism Score between Version 1 and 3: 1.7539756782039289%
- Plagiarism Score between Version 3 and 2: 2.0096938172360796%
- Plagiarism Score between Version 3 and 1: 1.997872088899397%
- Plagiarism Score between Version 2 and 1: 1.7732592505024236%

[1]Higher false positive rates and collision rates are a common issue with hash-based plagiarism checks, especially when using modular arithmetic hash functions. The program is in the appendix

3.2.4 Plagiarism Detection Strategies

Plagiarism detection is a complex task that employs various algorithmic strategies, each with unique strengths and limitations. These strategies are essential in identifying different types of plagiarism across diverse contexts. The following are the primary strategies used in plagiarism detection:

String Matching Algorithms

Description: These algorithms are designed to find exact string matches in texts. They are particularly effective in academic settings where verbatim plagiarism is prevalent (Dham, 2023).

Key Algorithms:

- KMP (Knuth-Morris-Pratt)
- Boyer-Moore
- Rabin-Karp (which I combined with the CBFs)

Experimental Approach: The efficiency and accuracy of these algorithms are assessed by comparing their performance in detecting direct copies in extensive documents. Metrics such as time complexity and false positive rates are crucial in this evaluation.

Natural Language Processing (NLP) Techniques

Description: NLP techniques are utilized to detect paraphrased plagiarism by identifying semantic similarities in texts, going beyond mere word-for-word matches (Gomede, E. (2023)).

Key Algorithms:

- Cosine similarity
- Jaccard similarity
- Bag-of-Words
- TF-IDF (Term Frequency-Inverse Document Frequency)

Experimental Approach: The effectiveness of these techniques is analyzed by testing their ability to identify semantic similarities in paraphrased documents, with a focus on balancing sensitivity and specificity.

Sequence Alignment Algorithms

Description: Originally developed for bioinformatics, these algorithms can be adapted for textual analysis to find similar sequences with minor edits (John & Benos, 2015).

Key Algorithms:

- Smith-Waterman
- Needleman-Wunsch (A dynamic programming approach which finds LCS)

Experimental Approach: These algorithms are tested on texts with slight modifications, such as the use of synonyms or rearranged sentences, to gauge their ability to identify closely related content.

Machine Learning Models

Description: Machine learning models are capable of learning from examples to identify complex patterns of plagiarism, adaptable to various forms of plagiarism.

Key Algorithms:

- SVM (Support Vector Machine)
- Neural Networks
- Decision Trees

Experimental Approach: Models are trained on datasets labeled with plagiarized and non-plagiarized content and evaluated using metrics like accuracy, precision, recall, and F1-score.

Combination Approaches

Description: This combines multiple methods, such as NLP with machine learning or sequence alignment with string matching, to increase robustness and accuracy.

Experimental Approach: A system integrating multiple algorithms is implemented and evaluated against systems using a single method.

Experimental Analysis

Dataset Preparation: A diverse set of documents, including academic papers, articles, and books with varying levels of plagiarism, is used.

Algorithm Implementation: Each strategy is optimized for the type of text being analyzed.

Performance Metrics: Accuracy, precision, recall, false positive rate, computational efficiency, and scalability are key metrics.

Cross-Validation: Cross-validation techniques ensure the reliability of results.

4 LOs and HCs Applications

4.1 LOs Application

#cs110-AlgoStratDataStruct: In this paper, I carefully chose custom hash functions over standard hash tables to address collision frequency and ensure efficient input distribution and the avalanche effect. My algorithmic strategy for the plagiarism checker emphasized a sliding window and word-based analysis for enhanced text comparison accuracy. This approach provides precise plagiarism detection and robustness against false positives, striking a balance between accuracy, efficiency, and scalability in text analysis (68 words)

#**cs110-ComputationalCritique:** I extended my analysis to cover a broad spectrum of test scenarios, including high false positive rates in hash functions. My critique now incorporates a comparative study of my plagiarism checker against established algorithms such as NLP-based, string-matching techniques, etc. Key decisions like choosing complex non-cryptographic hash functions and a word-based approach are thoroughly justified, highlighting their impact on algorithm efficiency (62 Words)

#**cs110-ComplexityAnalysis:** Conducted a detailed analysis of algorithm behavior under varying conditions, focusing on the effect of different false positive rates on Counting Bloom Filter performance. Comparative plots for key parameters like num_items and memory_size, especially between modular and complex non-cryptographic hashing, underscored their impact on performance. I made sure to include captions and labels clearly showing the x and y axes (61 words).

#**cs110-CodeReadability:** Demonstrated a strong commitment to code readability through precise naming of classes and variables, such as RollingHash, bitArrays, PlagiarismChecker, and CBF, and comprehensive docstrings giving insights on the purpose of the classes, methods and functions. Strategic comments throughout the code enhance understanding without causing distraction, significantly improving code clarity (50 Words).

#**professionalism:** Adhered strictly to APA referencing, ensuring accurate citations. Report formatting and presentation align with standards of authoritative sources like Wikipedia and Cormen et al., utilizing LaTeX for document preparation to emulate their style (for both pseudocode, programs, mathematical formulas, and other academic writing contents) and ensure clear, concise communication (50 Words).

#**pythonprogramming:** Focused on developing and refining key Python classes like RollingHash and Counting Bloom Filter for optimal performance. Emphasized on efficiency in RollingHash for sliding window computations and effective large dataset handling in CBF. Rigorous testing with edge cases ensured reliability and accuracy, showcasing a deep understanding of Python programming and commitment to quality. (54 words)

4.2 HCs Application

#**dataviz:** My data visualization focused on key metrics like memory size versus false positive rates and the number of hash functions. I examined memory scales with FPR, item count at fixed FPR, the variation of actual FPR with hash functions, and access time in a constant FPR Counting Bloom Filter. Each plot, aligned with theoretical analysis, included clear captions and labels, explaining trends and implications, enhancing understanding of the data's practical validity (72 Words)

#**audience:** My paper considers individuals ranging from beginners to professionals on the topic while balancing academic rigor with clarity, akin to Cormen's style and Wikipedia's expository approach. It includes detailed explanations about Counting Bloom Filters, covering their definitions, properties, and key elements like hash functions. To enhance comprehension, particularly for novices, I've included pseudocode for each concept and algorithm, designed to be straightforward and easy to follow. This approach not only elucidates how these ideas help in plagiarism detection but also ensures accessibility and understanding across all knowledge levels (89 Words)

#evidencebased In my project, I adopted an evidence-based approach to evaluate various hashing techniques and plagiarism detection methods. I critically analyzed methods like cosine similarity, Jaccard similarity, Needleman-Wunsch algorithm, and NLP models. Cosine and Jaccard similarities were found inadequate due to their neglect of word sequence order. Needleman-Wunsch, focus on subsequences, didn't align with plagiarism's emphasis on substrings. NLP models, though powerful, require extensive resources. Consequently, I chose the rolling hash function with the Rabin-Karp algorithm for its efficiency and sequence sensitivity, making it an ideal fit for plagiarism detection that balances practicality with computational effectiveness (95 words).

4.3 AI Tools Usage

In my project, I extensively utilized AI tools to streamline my workflow and enhance the quality of my research. For instance, I frequently used ChatGPT to debug and refine my LaTeX commands, especially when presenting complex pseudocode in an algorithmic format. An example includes troubleshooting issues with the `\begin{algorithmic}` command, which was crucial for correctly formatting pseudocode for my analysis. Also I was able to discover the usage of `\begin{lstlisting}` command, used for formatting code for smooth and well indented codes.

For gathering information on various plagiarism checking approaches, I turned to Perplexity.ai, which provided a wealth of sources and insights. This tool was instrumental in exploring different methods such as cosine similarity, Jaccard similarity, Needleman-Wunsch algorithm, and NLP models. It helped me access valuable resources like:

- An overview of the Rabin-Karp Algorithm from Brilliant.org.
- A detailed explanation of String Matching Algorithms from PrepBytes.
- An article by Everton Gomede on Medium discussing textual similarity in NLP for plagiarism detection, available at Medium.
- Lecture notes on Sequence Alignment from Carnegie Mellon University by John and Benos, found at Carnegie Mellon University.

These resources, retrieved through Perplexity.ai, were pivotal in deepening my understanding of various algorithms and their applicability to plagiarism detection, enabling me to make informed decisions about the methodologies I chose for my project. The use of these AI tools not only streamlined my research process but also ensured that the information I gathered was comprehensive and relevant to my study.

5 Appendix

5.1 Plagiarism Checker Implementation with Modular Hash Function

```
# Python code starts here
import math

# Defining my bitarray class.
# Basically, it's a list of 0s and 1s.
```

```python
# But for our CBF, we increment the bit positions with every number of appearance.
class bitarray:
    def __init__(self, size):
        """Initialize the bit array with a specified size, setting all bits to 0.
    """
        self.bits = [0] * size

    def set_bit(self, index, value):
        """Increment the bit at a specified index by the given value."""
        self.bits[index] += value

    def get_bit(self, index):
        """Retrieve the value of the bit at the specified index."""
        return self.bits[index]

    def __repr__(self):
        """Represent the bit array as a string for easy visualization."""
        return "".join(str(bit) if bit else "0" for bit in self.bits)

class CountingBloomFilter:
    """
    Implementation of a Counting Bloom Filter.
    It uses multiple hash functions to map elements to bit positions.
    """

    def __init__(self, num_items, fpr):
        """
        Initialize the Counting Bloom Filter.

        :param num_items: Estimated number of items to store.
        :param fpr: Desired false positive rate.
        """
        self.fpr = fpr
        self.size = -num_items * math.log(fpr) / (math.log(2) ** 2)
        self.num_hashfn = int(self.size * math.log(2) / num_items)
        self.bit_array = bitarray(int(self.size))

    @staticmethod
    def custom_hash(s, base=256, prime_modulus=10**9 + 7):
        """
        Custom hash function for hashing data.

        :param s: Data to be hashed.
        :param base: Base value for the hash function.
        :param prime_modulus: Prime modulus for calculating the hash.
        :return: Hashed value.
        """
        hash_value = 0
        base_power = 1

        # Convert the string to lowercase for case-insensitive tokenization
        s_lower = s.lower()

        for char in s_lower:
            hash_value = (hash_value * base + char) % prime_modulus
            base_power = (base_power * base) % prime_modulus

        return hash_value

    def hash_cbf(self, item):
        """
        Generate multiple hash values for an item using different seeds.

        :param item: Item to hash.
        :return: List of hash values.
        """
        hash_values = []
        for i in range(self.num_hashfn):
            hash_value = CountingBloomFilter.custom_hash(item, i) % int(self.size)
            hash_values.append(hash_value)
        return hash_values

    def search(self, item):
        """
        Check if an item is possibly in the bloom filter.

        :param item: Item to search for.
        :return: Boolean indicating if item is possibly present.
        """
        item = item.encode("utf-8")
        hash_values = self.hash_cbf(item)
        return all(self.bit_array.get_bit(hash_val) for hash_val in hash_values)

    def insert(self, item):
        """
        Insert an item into the bloom filter.

        :param item: Item to insert.
        """
        item = item.encode("utf-8")
        hash_values = self.hash_cbf(item)
        for hash_val in hash_values:
            self.bit_array.set_bit(hash_val, 1)

    def delete(self, item):
        """
        Delete an item from the bloom filter, if present.

        :param item: Item to delete.
        """
        item = item.encode("utf-8")
        hash_values = self.hash_cbf(item)
        for hash_val in hash_values:
            if self.bit_array.get_bit(hash_val) > 0:
                self.bit_array.set_bit(hash_val, self.bit_array.get_bit(hash_val)
    - 1)

    def insert_words(self, words):
        """
        Insert multiple words into the bloom filter.

        :param words: List of words to insert.
        """
        for word in words:
            self.insert(word)

    def is_word_present(self, word):
        """
        Check if a word is present in the bloom filter.

        :param word: Word to check.
        :return: Boolean indicating if word is present.
        """
        return self.search(word)

class RollingHash:
    """
    Implementation of a rolling hash for efficient text hashing,
    particularly useful for algorithms like Rabin-Karp.
    """

    def __init__(self, base, window_size):
        """
        Initialize the rolling hash.

        :param base: Base value for the hash function.
        :param window_size: Size of the window (number of words) to hash.
        """
        self.base = base
        self.window_size = window_size
        self.hash_value = 0
        self.base_pow = 1
        self.hash_mod = 1000000007
        self.window = []
        for _ in range(window_size - 1):
            self.base_pow = (self.base_pow * self.base) % self.hash_mod

    def append(self, word):
        """
        Append a word to the rolling hash.

        :param word: Word to append.
        """
        self.hash_value = (self.hash_value * self.base + ord('|')) % self.hash_mod  # Separator
        for char in word:
            self.hash_value = (self.hash_value * self.base + ord(char)) % self.hash_mod
        self.window.append(word)

    def skip(self):
        """
        Skip the oldest word from the rolling hash window.
        """
        if self.window:
            old_word = self.window.pop(0)
            word_pow = 1
            for char in old_word:
                self.hash_value = (self.hash_value - ord(char) * word_pow) % self.hash_mod
                word_pow = (word_pow * self.base) % self.hash_mod
            self.hash_value = (self.hash_value - ord('|') * word_pow) % self.hash_mod  # Separator

    def slide(self, new_word):
        """
        Slide the rolling hash window to include a new word.
        :param new_word: New word to include in the window.
        """
        if len(self.window) >= self.window_size:
            self.skip()
        self.append(new_word)

    def current_hash(self):
        """
        Get the current hash value of the rolling hash.

        :return: Current hash value.
        """
        return self.hash_value

    def reset(self):
        """
        Reset the rolling hash to its initial state.
        """
        self.hash_value = 0
        self.base_pow = 1
        self.window = []

def plagiarism_check_complex(version1, version2, num_items, fpr, window_size=2):
    """
    Check for plagiarism between two versions of a text using a complex algorithm.

    :param version1: List of words in the first version.
    :param version2: List of words in the second version.
    :param num_items: Number of items in the Counting Bloom Filter.
    :param fpr: False positive rate for the Counting Bloom Filter.
    :param window_size: Size of the window for the Rolling Hash.
    :return: Plagiarism score as a percentage.
    """
    cbf = CountingBloomFilter(num_items, fpr)
    rh = RollingHash(256, window_size)

    # Process version1
    for i in range(len(version1) - window_size + 1):
        window_words = version1[i:i + window_size]
        rh.slide(' '.join(window_words))
        hash_value = rh.current_hash()
        cbf.insert(str(hash_value))
        rh.reset()

    # Process version2 and check plagiarism
    match_count = 0
    plagiarized_sequences = []
    for i in range(len(version2) - window_size + 1):
        window_words = version2[i:i + window_size]
        rh.slide(' '.join(window_words))
        hash_value = rh.current_hash()
        if cbf.is_word_present(str(hash_value)):
            match_count += 1
            plagiarized_sequences.append(' '.join(window_words))
        rh.reset()

    plagiarism_score = (match_count / max(1, len(version2) - window_size + 1)) *
   100
```

```

    return plagiarism_score


url_version_1 = "https://bit.ly/39MurYb"
url_version_2 = "https://bit.ly/3we1QCp"
url_version_3 = "https://bit.ly/3vUecRn"
from requests import get


def get_txt_into_list_of_words(url):
    """Cleans the text data
    Input
    ----------
    url : string
    The URL for the txt file.
    Returns
    -------
    data_just_words_lower_case: list
    List of "cleaned-up" words sorted by the order they appear in the original file.
    """
    bad_chars = [";", ",", ".", "?", "!", "_", "[", "]", "(", ")", "*"]
    data = get(url).text
    data = "".join(c for c in data if c not in bad_chars)
    data_without_newlines = "".join(
        c if (c not in ["\n", "\r", "\t"]) else " " for c in data
    )
    data_just_words = [word for word in data_without_newlines.split(" ") if word != ""]
    data_just_words_lower_case = [word.lower() for word in data_just_words]
    return data_just_words_lower_case


version_1 = get_txt_into_list_of_words(url_version_1)
version_2 = get_txt_into_list_of_words(url_version_2)
version_3 = get_txt_into_list_of_words(url_version_3)
num_items = 10000
false_positive_rate = 0.01
# Example usage
complex_score_1_2 = plagiarism_check_complex(version_1, version_2, num_items,
    false_positive_rate, window_size=3)
print(f"Plagiarism Score between Version 1 and 2: {complex_score_1_2}%")

complex_score_2_3 = plagiarism_check_complex(version_2, version_3, num_items,
    false_positive_rate, window_size=3)
print(f"Plagiarism Score between Version 2 and 3: {complex_score_2_3}%")

complex_score_1_3 = plagiarism_check_complex(version_1, version_3, num_items,
    false_positive_rate, window_size=3)
print(f"Plagiarism Score between Version 1 and 3: {complex_score_1_3}%")

complex_score_1_4 = plagiarism_check_complex(version_3, version_2, num_items,
    false_positive_rate, window_size=3)
print(f"Plagiarism Score between Version 3 and 2: {complex_score_1_4}%")

complex_score_1_5 = plagiarism_check_complex(version_3, version_1, num_items,
    false_positive_rate, window_size=3)
print(f"Plagiarism Score between Version 3 and 1: {complex_score_1_5}%")
```

```

complex_score_1_6 = plagiarism_check_complex(version_2, version_1, num_items,
    false_positive_rate, window_size=3)
print(f"Plagiarism Score between Version 2 and 1: {complex_score_1_6}%")
```

5.1.1 Graph plots

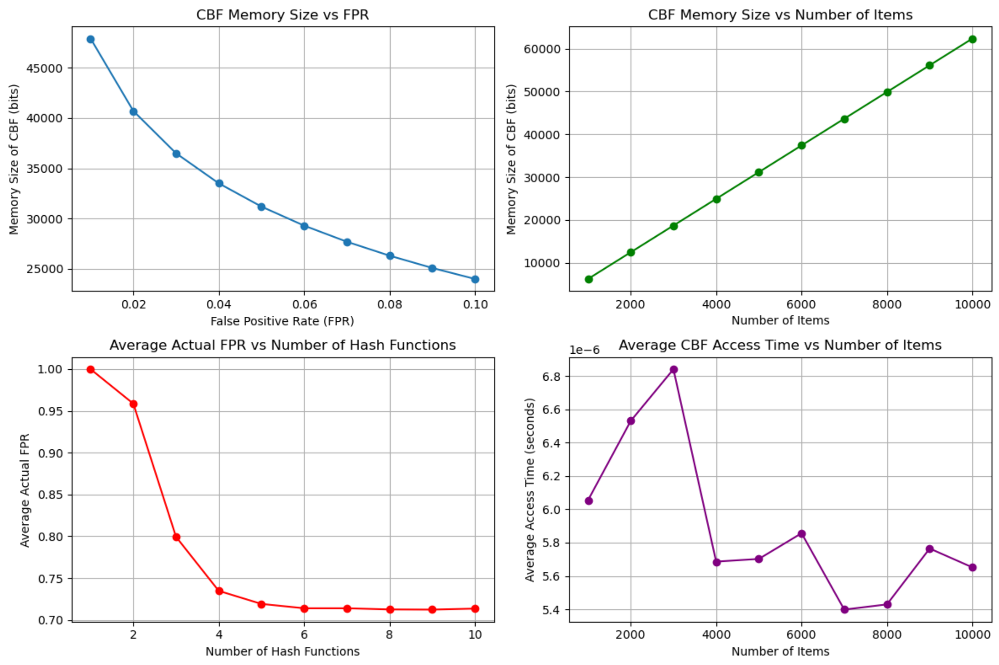

Figure 3: Graphs showing the relationship between the different parameters: Memory size, FPR, number of hash functions and number of items) when the simple modular arithmetic hash function is used

5.2 Plagiarism Checker Implementation with Non-cryptographic Complex Hash Function

```
import math


class bitarray:
    """
    A custom bit array class to manage individual bits efficiently.
    Each element in the bit array can be incremented to count occurrences.
    """

    def __init__(self, size):
        """Initialize the bit array with a specified size, setting all bits to 0.
    """
        self.bits = [0] * size

    def set_bit(self, index, value):
        """Increment the bit at a specified index by the given value."""
        self.bits[index] += value

    def get_bit(self, index):
```

```python
        """Retrieve the value of the bit at the specified index."""
        return self.bits[index]

    def __repr__(self):
        """Represent the bit array as a string for easy visualization."""
        return "".join(str(bit) for bit in self.bits)


class CountingBloomFilter:
    """
    Implementation of a Counting Bloom Filter.
    It uses multiple hash functions to map elements to bit positions.
    """

    def __init__(self, num_items, fpr):
        """
        Initialize the Counting Bloom Filter.

        :param num_items: Estimated number of items to store.
        :param fpr: Desired false positive rate.
        """
        self.fpr = fpr
        self.size = -num_items * math.log(fpr) / (math.log(2) ** 2)
        self.num_hashfn = int(self.size * math.log(2) / num_items)
        self.bit_array = bitarray(int(self.size))

    @staticmethod
    def custom_hash(data, seed):
        """
        Custom hash function for hashing data.

        :param data: Data to be hashed.
        :param seed: Seed value for the hash function.
        :return: Hashed value.
        """
        h = seed
        a = 0xB7E15163  # 3074997963
        b = 0x8AED2A6B  # 2334415563
        c = 0xD72A7A45  # 3597765701
        d = 0x9E3779B1  # 2654435761

        total_bytes = len(data)
        remaining_bytes = total_bytes % 4
        total_32bit_blocks = (total_bytes - remaining_bytes) // 4

        def mix(chunk):
            nonlocal h
            h ^= chunk
            h = (h * a) ^ (h >> 9)
            h = (h * b) ^ (h << 11)
            h = (h * c) ^ (h >> 13)
            h = (h * d) ^ (h << 15)
            h &= 0xFFFFFFFF

        for block in range(total_32bit_blocks):
            chunk = int.from_bytes(data[block * 4: block * 4 + 4], byteorder="
    little")
            mix(chunk)

        if remaining_bytes > 0:
            remaining_chunk = int.from_bytes(
                data[-remaining_bytes:] + b"\x00" * (4 - remaining_bytes),
                byteorder="little",
            )
            mix(remaining_chunk)

        h ^= total_bytes
        h ^= h >> 16
        h = (h * 0x85EBCA6B) & 0xFFFFFFFF
        h ^= h >> 13
        h = (h * 0xC2B2AE35) & 0xFFFFFFFF
        h ^= h >> 16

        return h




    def hash_cbf(self, item):
        """
        Generate multiple hash values for an item using different seeds.

        :param item: Item to hash.
        :return: List of hash values.
        """
        hash_values = []
        for i in range(self.num_hashfn):
            hash_value = CountingBloomFilter.custom_hash(item, i) % int(self.size)
            hash_values.append(hash_value)
        return hash_values

    def search(self, item):
        """
        Check if an item is possibly in the bloom filter.

        :param item: Item to search for.
        :return: Boolean indicating if item is possibly present.
        """
        item = item.encode("utf-8")
        hash_values = self.hash_cbf(item)
        return all(self.bit_array.get_bit(hash_val) for hash_val in hash_values)

    def insert(self, item):
        """
        Insert an item into the bloom filter.

        :param item: Item to insert.
        """
        item = item.encode("utf-8")
        hash_values = self.hash_cbf(item)
        for hash_val in hash_values:
            self.bit_array.set_bit(hash_val, 1)

    def delete(self, item):
        """
        Delete an item from the bloom filter, if present.

        :param item: Item to delete.
        """
        item = item.encode("utf-8")
        hash_values = self.hash_cbf(item)
        for hash_val in hash_values:
            if self.bit_array.get_bit(hash_val) > 0:
                self.bit_array.set_bit(hash_val, self.bit_array.get_bit(hash_val)
    - 1)

    def insert_words(self, words):
        """
        Insert multiple words into the bloom filter.

        :param words: List of words to insert.
        """
        for word in words:
            self.insert(word)

    def is_word_present(self, word):
        """
        Check if a word is present in the bloom filter.

        :param word: Word to check.
        :return: Boolean indicating if word is present.
        """
        return self.search(word)
class RollingHash:
    """
    Implementation of a rolling hash for efficient text hashing,
    particularly useful for algorithms like Rabin-Karp.
    """

    def __init__(self, base, window_size):
        """
        Initialize the rolling hash.

        :param base: Base value for the hash function.
        :param window_size: Size of the window (number of words) to hash.
        """
        self.base = base
        self.window_size = window_size
        self.hash_value = 0
        self.base_pow = 1
        self.hash_mod = 1000000007
        self.window = []
        for _ in range(window_size - 1):
            self.base_pow = (self.base_pow * self.base) % self.hash_mod

    def append(self, word):
        """
        Append a word to the rolling hash.

        :param word: Word to append.
        """
        self.hash_value = (self.hash_value * self.base + ord('|')) % self.hash_mod
      # Separator
        for char in word:
            self.hash_value = (self.hash_value * self.base + ord(char)) % self.
    hash_mod
        self.window.append(word)

    def skip(self):
        """
        Skip the oldest word from the rolling hash window.
        """
        if self.window:
            old_word = self.window.pop(0)
            word_pow = 1
            for char in old_word:
                self.hash_value = (self.hash_value - ord(char) * word_pow) % self.hash_mod
                word_pow = (word_pow * self.base) % self.hash_mod
            self.hash_value = (self.hash_value - ord('|') * word_pow) % self.hash_mod  # Separator

    def slide(self, new_word):
        """
        Slide the rolling hash window to include a new word.

        :param new_word: New word to include in the window.
        """
        if len(self.window) >= self.window_size:
            self.skip()
        self.append(new_word)

    def current_hash(self):
        """
        Get the current hash value of the rolling hash.

        :return: Current hash value.
        """
        return self.hash_value

    def reset(self):
        """
        Reset the rolling hash to its initial state.
        """
        self.hash_value = 0
        self.base_pow = 1
        self.window = []


def plagiarism_check_complex(version1, version2, num_items, fpr, window_size=2):
    """
    Check for plagiarism between two versions of a text using a complex algorithm.

    :param version1: List of words in the first version.
    :param version2: List of words in the second version.

    return: plagiarism score
    """
    cbf = CountingBloomFilter(num_items, fpr)
    rh = RollingHash(256, window_size)

    # Process version1
    for i in range(len(version1) - window_size + 1):
        window_words = version1[i:i + window_size]
        rh.slide(' '.join(window_words))
        hash_value = rh.current_hash()
        cbf.insert(str(hash_value))
        rh.reset()

    # Process version2 and check plagiarism
    match_count = 0
    plagiarized_sequences = []
    for i in range(len(version2) - window_size + 1):
        window_words = version2[i:i + window_size]
        rh.slide(' '.join(window_words))
        hash_value = rh.current_hash()
        if cbf.is_word_present(str(hash_value)):
            match_count += 1
            plagiarized_sequences.append(' '.join(window_words))
        rh.reset()

    plagiarism_score = (match_count / max(1, len(version2) - window_size + 1)) *
    100

    return plagiarism_score


url_version_1 = "https://bit.ly/39MurYb"
url_version_2 = "https://bit.ly/3we1QCp"
url_version_3 = "https://bit.ly/3vUecRn"
from requests import get


def get_txt_into_list_of_words(url):
    """Cleans the text data
    Input
    ----------
    url : string
    The URL for the txt file.
    Returns
    -------
    data_just_words_lower_case: list
    List of "cleaned-up" words sorted by the order they appear in the original
    file.
    """
    bad_chars = [";", ",", ".", "?", "!", "_", "[", "]", "(", ")", "*"]
    data = get(url).text
    data = "".join(c for c in data if c not in bad_chars)
    data_without_newlines = "".join(
        c if (c not in ["\n", "\r", "\t"]) else " " for c in data
    )
    data_just_words = [word for word in data_without_newlines.split(" ") if word
    != ""]
    data_just_words_lower_case = [word.lower() for word in data_just_words]
    return data_just_words_lower_case


version_1 = get_txt_into_list_of_words(url_version_1)
version_2 = get_txt_into_list_of_words(url_version_2)
version_3 = get_txt_into_list_of_words(url_version_3)
num_items = 10000
false_positive_rate = 0.01
# Example usage
complex_score_1_2 = plagiarism_check_complex(version_1, version_2, num_items,
    false_positive_rate, window_size=3)
```

```
print(f"Complex Plagiarism Score between Version 1 and 2: {complex_score_1_2}%")

complex_score_2_3 = plagiarism_check_complex(version_2, version_3, num_items,
    false_positive_rate, window_size=3)
print(f"Complex Plagiarism Score between Version 2 and 3: {complex_score_2_3}%")

complex_score_1_3 = plagiarism_check_complex(version_1, version_3, num_items,
    false_positive_rate, window_size=3)
print(f"Complex Plagiarism Score between Version 1 and 3: {complex_score_1_3}%")

complex_score_1_4 = plagiarism_check_complex(version_3, version_2, num_items,
    false_positive_rate, window_size=3)
print(f"Complex Plagiarism Score between Version 3 and 2: {complex_score_1_4}%")

complex_score_1_5 = plagiarism_check_complex(version_3, version_1, num_items,
    false_positive_rate, window_size=3)
print(f"Complex Plagiarism Score between Version 3 and 1: {complex_score_1_5}%")

complex_score_1_6 = plagiarism_check_complex(version_2, version_1, num_items,
    false_positive_rate, window_size=3)
print(f"Complex Plagiarism Score between Version 2 and 1: {complex_score_1_6}%")
```

5.2.1 Plots

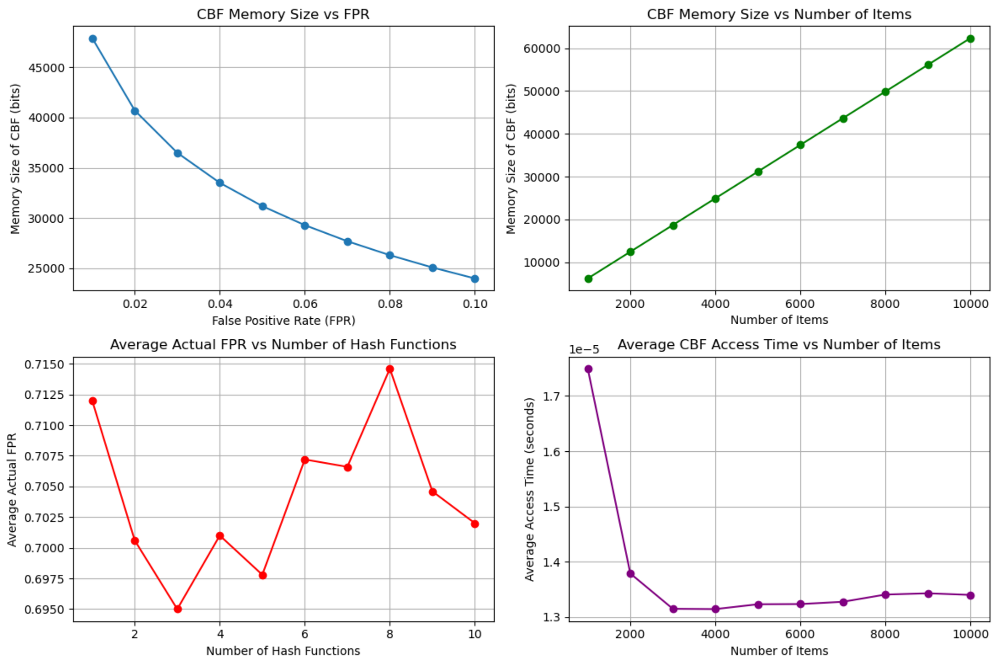

Figure 4: Graphs showing the relationship between the different parameters: Memory size, FPR, number of hash functions and number of items) when the complex non-cryptographic hash function is used

5.3 Programs for all plots used

5.3.1 Functions for every comparisons

```python
import matplotlib.pyplot as plt
import time
import numpy as np

# Analysis 1: Memory Size Scaling with FPR

def analyze_cbf_scaling_with_fpr(shakespeare_words, fpr_values, num_items):
    sizes = []
    for fpr in fpr_values:
        cbf = CountingBloomFilter(num_items, fpr)
        sizes.append(len(cbf.bit_array.bits))
    return sizes

# Analysis 2: Memory Size Scaling with Number of Items

def analyze_cbf_scaling_with_items(shakespeare_words, fixed_fpr, num_items_values)
    :
    sizes = []
    for num_items in num_items_values:
        cbf = CountingBloomFilter(num_items, fixed_fpr)
        sizes.append(len(cbf.bit_array.bits))
    return sizes

# Analysis 3: Actual FPR Scaling with Number of Hash Functions

def actual_fpr_analysis(shakespeare_words, fpr, num_items, num_hash_functions,
    num_runs=10):
    """
    Analyze how the actual FPR scales with the number of hash functions.
    This function performs multiple runs for each hash function count and averages
     the results.

    Parameters:
    shakespeare_words (list): The list of words from Shakespeare's works.
    fpr (float): The desired false positive rate.
    num_items (int): The number of items to insert into the CBF.
    num_hash_functions (range): A range of numbers of hash functions to test.
    num_runs (int): The number of runs to average over for each hash function
    count.

    Returns:
    list: A list of average actual FPRs corresponding to each hash function count.
    """
    avg_actual_fprs = []
    for num_hash_fn in num_hash_functions:
        total_fpr = 0
        for _ in range(num_runs):
            cbf = CountingBloomFilter(num_items, fpr)
            cbf.num_hashfn = num_hash_fn  # Manually setting number of hash
    functions
            # Inserting items
            for word in shakespeare_words[:num_items]:
                cbf.insert(word)
            # Checking FPR
            false_positives = 0
            for word in shakespeare_words[num_items:num_items * 2]:
                if cbf.is_word_present(word):
                    false_positives += 1
            total_fpr += false_positives / num_items
        avg_actual_fprs.append(total_fpr / num_runs)
    return avg_actual_fprs


# Analysis 4: Access Time Scaling with Number of Items

def cbf_access_time_analysis(shakespeare_words, fpr, num_items_values, num_runs
    =10):
    """
    Analyze how access time to hashed values scales with the number of items
    stored in a CBF at a constant FPR.
    This function performs multiple runs for each number of items and averages the
     results.

    Parameters:
    shakespeare_words (list): The list of words from Shakespeare's works.
    fpr (float): The desired false positive rate.
    num_items_values (range): A range of numbers of items to test.
    num_runs (int): The number of runs to average over for each number of items.

    Returns:
    list: A list of average access times corresponding to each number of items.
    """
    avg_access_times = []
    for num_items in num_items_values:
        total_access_time = 0
        for _ in range(num_runs):
            cbf = CountingBloomFilter(num_items, fpr)
            # Inserting items
            for word in shakespeare_words[:num_items]:
                cbf.insert(word)
            # Measuring access time
            start_time = time.time()
            for word in shakespeare_words[:num_items]:
                cbf.is_word_present(word)
            access_time = (time.time() - start_time) / num_items
            total_access_time += access_time
        avg_access_times.append(total_access_time / num_runs)
    return avg_access_times


# Parameters for analysis
import requests


def download_shakespeare_works(url):
    response = requests.get(url)
    text = response.text
    return text.split()

# Replace with the actual URL of the complete works of Shakespeare
shakespeare_url = "https://gist.githubusercontent.com/raquelhr/78
    f66877813825dc344efefdc684a5d6/raw/361a40e4cd22cb6025e1fb2baca3bf7e166b2ec6/"
    # Replace with the actual URL
```

```python
shakespeare_words = download_shakespeare_works(shakespeare_url)

fpr_values = np.linspace(0.01, 0.1, 10)  # Different FPR values
num_items_values = range(1000, 10001, 1000)  # Different number of items
num_hash_functions = range(1, 11)  # Different number of hash functions
fixed_fpr = 0.05  # Fixed FPR for some analyses

# Execute analyses
size_with_fpr = analyze_cbf_scaling_with_fpr(shakespeare_words, fpr_values, 5000)
size_with_items = analyze_cbf_scaling_with_items(shakespeare_words, fixed_fpr,
    num_items_values)
actual_fpr_results = actual_fpr_analysis(shakespeare_words, fixed_fpr, 5000,
    num_hash_functions, num_runs = 10)
access_times = cbf_access_time_analysis(shakespeare_words, fixed_fpr,
    num_items_values, num_runs = 10)
```

5.3.2 Visualizing the plots

```python
# Plotting the results of the analyses


plt.figure(figsize=(12, 8))

# Plot 1: CBF Memory Size vs FPR
plt.subplot(2, 2, 1)
plt.plot(fpr_values, size_with_fpr, marker='o')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('Memory Size of CBF (bits)')
plt.title('CBF Memory Size vs FPR')
plt.grid(True)

# Plot 2: CBF Memory Size vs Number of Items
plt.subplot(2, 2, 2)
plt.plot(num_items_values, size_with_items, marker='o', color='green')
plt.xlabel('Number of Items')
plt.ylabel('Memory Size of CBF (bits)')
plt.title('CBF Memory Size vs Number of Items')
plt.grid(True)

# Plot 3: Actual FPR vs Number of Hash Functions
plt.subplot(2, 2, 3)
plt.plot(num_hash_functions, actual_fpr_results, marker='o', color='red')
plt.xlabel('Number of Hash Functions')
plt.ylabel('Average Actual FPR')
plt.title('Average Actual FPR vs Number of Hash Functions')
plt.grid(True)

# Plot 4: CBF Access Time vs Number of Items
plt.subplot(2, 2, 4)
plt.plot(num_items_values, access_times, marker='o', color='purple')
plt.xlabel('Number of Items')
plt.ylabel('Average Access Time (seconds)')
plt.title('Average CBF Access Time vs Number of Items')
plt.grid(True)

plt.tight_layout()
plt.show()
```

References

[1] An, L., Ouyang, Y., & Zhou, Y. (2018). *Cuckoo Filter: Practically Better Than Bloom*. arXiv. Retrieved from https://arxiv.org/pdf/1804.04777.pdf

[2] Academia.edu. (n.d.). *An efficient and scalable plagiarism checking system using Bloom Filters*. Retrieved from https://www.academia.edu/10707796/An_efficient_and_scalable_plagiarism_checking_system_using_Bloom_filters

[3] Blockchain Academy HS Mittweida. (n.d.). *Lesson 12: Client-Side Block Filtering - Compact Block Filtering (CBF) in Bitcoin*. Retrieved from https://blockchainacademy.hs-mittweida.de/courses/blockchain-introduction-technical-beginner-to-intermediate/lessons/lesson-12-client-side-block-filtering-compact-block-filtering-cbf-in-bitcoin/topic/downsides-of-cbf/

[4] Brilliant. (n.d.). *Bloom Filter*. Retrieved from https://brilliant.org/wiki/bloom-filter/

[5] Brilliant.org. (n.d.). *Rabin-Karp Algorithm*. Retrieved from https://brilliant.org/wiki/rabin-karp-algorithm/

[6] Cloudflare. (n.d.). *When Bloom Filters don't bloom*. Retrieved from https://blog.cloudflare.com/when-bloom-filters-dont-bloom/

[7] Cheng, Shuxing, & Zhang, Liang-Jie. (2009.). *CBF-based Storage Unit for Service Group S-1*. ResearchGate. Retrieved from https://www.researchgate.net/figure/CBF-based-Storage-Unit-for-Service-Group-S-1_fig5_221586903

[8] D'Angelo, G., & Palmieri, F. (n.d.). *Accurate Counting Bloom Filters for Large-Scale Data Processing*. ResearchGate. Retrieved from https://www.researchgate.net/publication/258394929_Accurate_Counting_Bloom_Filters_for_Large-Scale_Data_Processing

[9] Delinea. (n.d.). *Weak Password Finder Tool for Active Directory*. Retrieved from https://delinea.com/resources/weak-password-finder-tool-active-directory

[10] Dham, M. (2023, June 29). *String Matching Algorithm*. PrepBytes. Retrieved from https://www.prepbytes.com/blog/strings/string-matching-algorithm/

[11] Geravand S. et. al. (2014). *ScienceDirect*. Retrieved from https://www.sciencedirect.com/science/article/abs/pii/S0045790614001712

[12] GeeksforGeeks. (n.d.). *Bloom Filters – Introduction and Python Implementation*. Retrieved from https://www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation

[13] GeeksforGeeks. (n.d.). *Hash Functions and List Types of Hash Functions*. Retrieved from https://www.geeksforgeeks.org/hash-functions-and-list-types-of-hash-functions/

[14] Gomede, E. (2023). *Textual similarity in natural language processing for plagiarism detection*. Medium. Retrieved from https://medium.com/@evertongomede/textual-similarity-in-natural-language-processing-for-plagiarism-detection-411eb64564c6

[15] Hal. (n.d.). Retrieved from https://hal.science/hal-01054040/document

[16] John, B., & Benos, T. (2015). *Sequence Alignment [Lecture notes]*. Carnegie Mellon University. Retrieved from https://www.cs.cmu.edu/~02710/Lectures/SeqAlign2015.pdf

[17] Limb, B. (n.d.). *Bloomfilter Tutorial*. Retrieved from https://llimllib.github.io/bloomfilter-tutorial/

[18] Melsted, P., & Pritchard, J. K. (2011). *Efficient counting of k-mers in DNA sequences using a Bloom Filter*. BMC Bioinformatics, 12, 333. Retrieved from https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-12-333

[19] OpenGenus IQ. (n.d.). *Applications of Bloom Filter*. Retrieved from https://iq.opengenus.org/applications-of-bloom-filter/

[20] Snoeren, A. C., Partridge, C., & Sanchez, L. A. (2001). *Hash-Based IP Traceback*. SIGCOMM '01. Retrieved from https://conferences.sigcomm.org/sigcomm/2001/p1-snoeren.pdf

[21] SystemDesign.One. (n.d.). *Bloom Filters Explained*. Retrieved from https://systemdesign.one/bloom-filters-explained/

[22] Tarkoma, S., Rothenberg, C. E., & Lagerspetz, E. (n.d.). *The Variable-Increment Counting Bloom Filter*. ResearchGate. Retrieved from https://www.researchgate.net/publication/254031690_The_Variable-Increment_Counting_Bloom_Filter