Hindawi
Applied Computational Intelligence and Soft Computing
Volume 2023, Article ID 3278505, 16 pages
https://doi.org/10.1155/2023/3278505

Research Article

An Improved Hashing Approach for Biological Sequence to Solve Exact Pattern Matching Problems

Prince Mahmud, Anisur Rahman, and Kamrul Hasan Talukder

Computer Science and Engineering Discipline, Khulna University, Khulna 9208, Bangladesh

Correspondence should be addressed to Prince Mahmud; m.princecse@gmail.com

Received 17 April 2023; Revised 30 October 2023; Accepted 10 November 2023; Published 20 November 2023

Academic Editor: Anandakumar Haldorai

Pattern matching algorithms have gained a lot of importance in computer science, primarily because they are used in various domains such as computational biology, video retrieval, intrusion detection systems, and fraud detection. Finding one or more patterns in a given text is known as pattern matching. Two important things that are used to judge how well exact pattern matching algorithms work are the total number of attempts and the character comparisons that are made during the matching process. The primary focus of our proposed method is reducing the size of both components wherever possible. Despite sprinting, hash-based pattern matching algorithms may have hash collisions. The Efficient Hashing Method (EHM) algorithm is improved in this research. Despite the EHM algorithm's effectiveness, it takes a lot of time in the preprocessing phase, and some hash collisions are generated. A novel hashing method has been proposed, which has reduced the preprocessing time and hash collision of the EHM algorithm. We devised the Hashing Approach for Pattern Matching (HAPM) algorithm by taking the best parts of the EHM and Quick Search (QS) algorithms and adding a way to avoid hash collisions. The preprocessing step of this algorithm combines the bad character table from the QS algorithm, the hashing strategy from the EHM algorithm, and the collision-reducing mechanism. To analyze the performance of our HAPM algorithm, we have used three types of datasets: *E. coli*, DNA sequences, and protein sequences. We looked at six algorithms discussed in the literature and compared our proposed method. The Hash-q with Unique FNG (HqUF) algorithm was only compared with *E. coli* and DNA datasets because it creates unique bits for DNA sequences. Our proposed HAPM algorithm also overcomes the problems of the HqUF algorithm. The new method beats older ones regarding average runtime, number of attempts, and character comparisons for long and short text patterns, though it did worse on some short patterns.

1. Introduction

Pattern matching is one of the most significant tasks in computer science. Finding a specific pattern within a large pattern or text is known as pattern matching [1]. Problems of this type arise in many areas of the fourth industrial revolution, including networking, signal processing, data recovery, language processing, artificial intelligence, and many more [2]. Pattern matching is also known as pattern searching or string matching.

String-matching algorithms make up a significant subclass of string algorithms. These algorithms look for instances in a lengthy string or text where a single string or multiple strings, collectively called patterns, appear. String-matching techniques look for strings in text strings (where strings are collections of characters) that fit a predefined pattern (finite set) [3]. Let a text t, which has a length of n, and the pattern be p, which has a length of m, where m is less than or equal to n. The sequence and the search window are compared, character by character, to find the pattern in the text string. The term "search window" refers to the area of a text string compared to a pattern in which the search box's length equals the pattern's length [4].

To comprehend biological data, mainly when the datasets are enormous and complicated, the interdisciplinary discipline of bioinformatics develops techniques and software tools [5]. Pattern matching issues appear in many computational bioinformatics tasks, including basic local synchronization search, biomarker discovery, sequence matching, homologous sequence identification, and proteogenomic mapping [6, 7]. Pattern matching can be used in biotechnology, forensics, medical, and agricultural research to look into probable disease or anomaly diagnoses [8]. Hashing's effectiveness in storage and search has made it a popular choice for colossal pattern matching [9]. Hashing methods generate a unique hash value that helps search for a specific text pattern [10].

Pattern matching can be broken down into numerous categories. Exact and approximate pattern matching are the two primary types of pattern matching in terms of accuracy [11, 12]. While approximate pattern matching results in erroneous searching, exact matching effectively searches for the precise occurrence of the pattern within the text [13, 14]. Based on the number of patterns, pattern matching can be divided into two categories: single and multiple patterns [14, 15]. Single pattern matching searches for a single pattern in the text, whereas multiple patterns are searched in multiple pattern matching [16]. The concept of exact pattern matching can be classified into two distinct groups: online and offline. Online string matching involves preprocessing only the pattern, while offline string matching involves preprocessing the text while keeping the pattern unchanged [17, 18]. Figure 1 displays the classification of string matching.

Although many algorithms were proposed to solve the pattern matching problem, here we propose an approach based on the hashing technique. The hashing method notably improves efficiency and effectiveness when applied to pattern-matching challenges [20]. The success of this algorithm can be attributed to its versatile design, which allows it to perform well across various domains and facilitates single and multiple searches [21, 22]. The main objective of this paper is to design and construct an efficient algorithm for better performance in exact pattern matching. The algorithm is applied within bioinformatics, specifically for analyzing biological data such as DNA and protein sequences. The performance of the algorithm is evaluated by comparing it with various methods, including traditional, hybrid, and hashing-based pattern matching techniques. These techniques include Quick Search (QS), Maximum Shift (MS), Minimum Attempts and Comparisons (MAC), Back and Forth Matching (BFM), Efficient Hashing Method (EHM), and Hash-q Unique FNG (HqUF). The evaluation is conducted on publicly available datasets, utilizing identical parameter settings for all methods. The contributions of this research can be summarized as follows:

(1) We propose a hashing approach that improves the EHM algorithm, which includes two phases: preprocessing and searching
(2) The preprocessing phase integrates the EHM algorithm's hashing technique, QS bad character table, and hash collision-reducing technique, whereas the searching phase follows our own approach

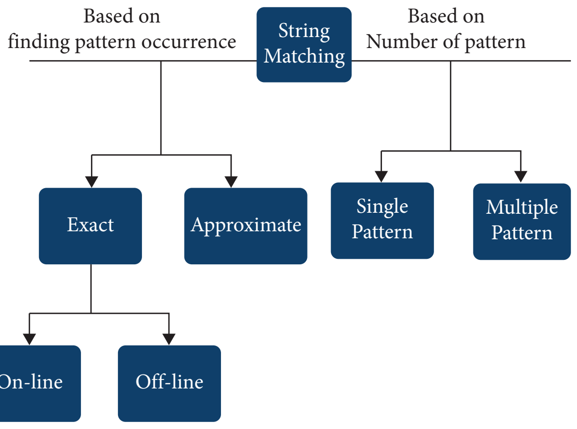


Figure 1: Taxonomy of string matching [11, 19].

(3) Experiments were undertaken to assess the efficacy of the proposed approach, which was then compared with several algorithms across three distinct datasets
(4) Whether the accuracy is outstanding compared to other tasks is investigated

The subsequent sections of this work are structured in the following manner. The related works that have been recently developed are outlined in Section 2. Section 3 of the document comprehensively explains the proposed technique for addressing string-matching problems. This explanation also includes a working example, an analysis of the hashing method, and the time complexity. In addition, graphical representations enhance the clarity and understanding of the presented information. Section 4 represents the outcomes of the conducted tests, accompanied by appropriate graphical and tabular illustrations. The conclusions of our study are explained in Section 5.

2. Literature Review

To address the pattern matching problem, a large number of algorithms employ their technique or a hybrid approach. Hash-based string-matching algorithms have become very popular recently for solving string-matching problems. However, the algorithm that first comes to mind when thinking about how to solve the pattern-finding problem is probably Brute Force (BF) [23]. Brute Force is a straightforward algorithm that searches the pattern character by character and right-shifts the pattern by one position [24]. The Boyer–Moore (BM) method introduces three ideas for searching for patterns: the right-to-left comparison, the good suffix rule, and the bad character rule [24, 25]. According to the pattern of the selected alphabet, the final two observations are subject to the preprocessing stage [26]. Aligning the pattern from left to right and using a search strategy that checks characters in right-to-left order character by character, like the Brute-Force algorithm, makes the string-matching method work better [27]. The BM algorithm has a variation known as the Quick Search algorithm [28]. The bad character table value of the next character in the search window determines the shifted value of the QS algorithm [29]. The Back and Forth Matching algorithm employs index-based techniques [30]. When the first and last characters of the search window match the pattern, the second and rightmost characters are checked simultaneously, and the remaining characters are also checked in the above manner [19, 31]. An efficient string-matching algorithm known as Maximum Shift was designed and implemented in 2014 [32]. Quick Search, Zhu–Takaoka, and Horspool algorithms are combined to form the MS algorithm. Quick Search and Zuh–Takaoka bad character rules are constructed during the preparation stage of the MS algorithm. The name of this algorithm is based on the maximum shift feature in the search phase, which takes the maximum value from two tables. The Berry–Ravindran (BR) and index-based methods are combined to create the modern and effective exact string-matching method known as the Minimum Attempts and Comparisons method [33]. The MAC algorithm initially creates the BR table from the BR equation and the index table (IT) for preprocessing. Based on the initial character of the pattern, the shifting value table (IbSv) is subsequently produced from the index table. A hashing-based string-matching algorithm known as Rabin–Karp (RK) was developed in 1987 [34]. This algorithm uses the hashing approach to identify patterns within a text [35]. Lecroq introduced the Hash-q algorithm, which calculates a hash value between 0 and 255 for each q-gram in the pattern p [36, 37]. This algorithm computes a shift for each hash value and generates q values from 3 to 8. The hash function is

$$\text{Hash} - q = 4p[k-2] + 2p[k-1] + p[k]. \tag{1}$$

Here, many hash collisions are generated for different q values. In a recent study, researchers proposed an efficient hashing approach with an enhanced form of Hash-q known as Hash-q with Unique FNG [38]. This algorithm converts ASCII codes, represented as follows, into a hash with a bit representation. A: 01000001, C: 01000011, T: 01010100, G: 01000111. The basic idea of this algorithm is to use two unique bits, which are A: 00, C: 01, G: 11, and T: 10, instead of eight bits. These two bits are the rightmost second and third bits from the ASCII code of the corresponding character. The following is the hash function for this algorithm:

$$\begin{aligned}\text{HqUF} = &\left(\sum_{k=0}^{q-2}(x[k]\&0b110) \ll 2(q-k-1) - 1\right)\\ &+ (x[q-1]\&0b110) \gg 1.\end{aligned} \tag{2}$$

If the last q characters of the text and the pattern do not have equal hash values, the pattern will be moved by $m - q + 1$; otherwise, compare the text and pattern from start to before the last q characters.

Precise string-matching algorithms are widely used in many disciplines, including bioinformatics, computational biology, text processing, and intrusion detection [39]. These algorithms are widely used to solve various problems related to string matching and pattern recognition. Exact string-matching methods are used in bioinformatics for a wide range of tasks, including genome assembly, sequence alignment, and gene prediction [40]. For DNA and protein sequence alignment, methods like Quick Search and Boyer–Moore are frequently used. These algorithms can quickly match and compare sequences to identify similarities and differences, which is critical for understanding the structure and function of genes and proteins. Exact string-matching algorithms are also used in computational biology to find patterns in DNA and protein sequences. For example, the Boyer–Moore algorithm can search for specific patterns in DNA sequences [41], which is very important for finding genes, regulatory elements, and other functional parts. In text processing, exact string-matching algorithms are used for tasks such as spell-checking, plagiarism detection, and natural language processing [11]. For example, algorithms such as the Berry–Ravindran algorithm can be used to identify the difference between two strings, which helps spell-check and identify plagiarism. Intrusion detection systems use a string-matching method to identify data packets with intrusion-related keywords. Whenever new data is received, it is compared to the database, which contains all the dangerous code. If a match is detected, an alert will be sent. Every intruded packet must be captured and identified using exact string-matching methods.

The essential requirement of the abovementioned methods is to reduce the search time. These methods were usually accomplished by creating a single function or a hybrid approach combining the advantageous features of several single algorithms. Performance can be affected by various factors, including the speed of the processor, the operating system, and the database, the length of the string or pattern, the frequency with which it occurs, and the size of the alphabet. Although the time complexity of the BF algorithm is high, it applies to all fields; nonetheless, it performs slowly when dealing with lengthy patterns and text. The BM algorithm is superior to the BF algorithm, although the performance of the BM algorithm is variable depending on the length of the pattern and the alphabet set. Using QS to differentiate between short and long sequences in real-world applications is a process that is both quick and uncomplicated to carry out. Despite this, the amount of time required for QS preprocessing grows longer when the letter size of the pattern is increased. Compared to previous algorithms, BFM shows a considerable boost in its ability to locate strings inside substantial text files. However, because a preprocessing phase must be finished before searching can begin, BFM's performance will improve if the text and the pattern are brief. This is because of the nature of the search. In experimental settings, the MS algorithm yields superior results for English text, DNA sequences, and protein sequences; nevertheless, its performance suffers when dealing with limited alphabet sets. Even though the MAC algorithm only makes a limited number of attempts and comparisons, it nonetheless needs the maximum amount of time to execute because of the index-based approach. The Rabin–Karp algorithm offers a rapid means of determining the presence of a pattern inside a given text, obviating the need to examine all potential positions within the text exhaustively. However, it is essential to note that this technique may exhibit suboptimal temporal complexity in scenarios where numerous hash collisions arise. The Hash-q algorithm works fast for small patterns in small-sized alphabets but shows the worst output for large patterns due to having to calculate a hash value between 0 and 255 for each q-gram. The HqUF technique effectively mitigates hash collisions in the context of DNA sequences. The system offers optimal hashing capabilities and efficiently generates hash values. The main problem with the HqUF algorithm is that it only produces unique bits for DNA sequences, and it is complicated to create individual bits for proteins or other arrangements that contain more than four characters.

The HqUF method has been identified as a highly effective hash-based text matching technique. This algorithm efficiently adapts the Hash-q algorithm and demonstrates suitability for DNA sequences. Nevertheless, a limitation of this technique is its exclusive ability to generate distinct hash values solely for DNA sequences. This algorithm cannot generate separate bits if a dataset contains more than four characters. Research has also demonstrated that a single algorithm with hashing approaches capable of effectively processing all types of data is yet to be identified as the optimal option. In light of the shortcomings of existing algorithms, this research aims to introduce a novel and effective approach that generates distinct hash values across all datasets to prevent hash collisions. Therefore, if hash collisions can be eliminated, runtime, character comparisons, and hash comparisons can all be decreased.

3. Proposed Approach

In the challenge of string-matching problems, our observation concludes that some existing algorithms consider the act of shifts, comparisons, and execution time. We concentrated our research on the string-matching algorithm and proposed one that may decrease the number of shifts and comparisons for sequential pattern matching. We have improved the hashing method used in the Efficient Hashing Method (EHM) [19]. Our proposed algorithm is divided into two phases: one is preprocessing, and the other is searching.

3.1. The Proposed Hash Function.

The hash function is mainly used for the ASCII value of its corresponding character. The basic idea of the hashing method is to sum up all the ASCII values of a particular string and modulate it with a specific prime number to get a remainder. The following equation is used to generate the hash function:

$$h(S) = [\mathrm{ASCII}(S1) + \mathrm{ASCII}(S2) + \mathrm{ASCII}(S3) + \cdots + \mathrm{ASCII}(Sn)] \bmod q. \tag{3}$$

Here, n is the pattern length or substring length, and q denotes a predefined prime number, where prime numbers are a subset of natural numbers divisible solely by one and the number itself. The likelihood of producing distinct and nonrepetitive values increases when using prime numbers in the hashing process. This characteristic is inherent to the field of mathematics. As an illustration, consider the given string "ResearchTopic." A different hash value may be obtained by assigning a prime number to each letter and summing these values.

There is some hash collision that occurs in this equation. Let a pattern $p1 = \mathrm{ACGTTGA}$ and $p2 = \mathrm{TAGCACG}$ of a DNA sequence and prime numbers 17. The hash function or value is computed by the following equation:

$$\begin{aligned} h(p1) &= (\mathrm{ASCII}(A) + \mathrm{ASCII}(C) + \mathrm{ASCII}(G) + \mathrm{ASCII}(T) + \mathrm{ASCII}(T) + \mathrm{ASCII}(G) + \mathrm{ASCII}(A)) \bmod 17 = 14, \\ h(p2) &= (\mathrm{ASCII}(T) + \mathrm{ASCII}(A) + \mathrm{ASCII}(G) + \mathrm{ASCII}(C) + \mathrm{ASCII}(A) + \mathrm{ASCII}(C) + \mathrm{ASCII}(G)) \bmod 17 = 14. \end{aligned} \tag{4}$$

These two patterns generate the same hash values, but these are different patterns. This is known as a hash collision. We improve the hash collision by dividing the pattern with a specific prime number for getting a quotient. The following equation is used to generate the quotient:

$$r(S) = [\mathrm{ASCII}(S1) + \mathrm{ASCII}(S2) + \mathrm{ASCII}(S3) + \cdots + \mathrm{ASCII}(\mathrm{Sn})] \,\mathrm{divide}\, q. \tag{5}$$

Here, n is the pattern or substring length, and q denotes a predefined prime number.

Let the above pattern $p1 = \text{ACGTTGA}$ and $p2 = \text{TAGCACG}$ of a DNA sequence and prime numbers 17. The quotient is computed by the following equation:

$$\begin{aligned} r(p1) &= (\text{ASCII}(A) + \text{ASCII}(C) + \text{ASCII}(G) + \text{ASCII}(T) + \text{ASCII}(T) + \text{ASCII}(G) + \text{ASCII}(A))\,\text{divide}\,17 = 29, \\ r(p2) &= (\text{ASCII}(T) + \text{ASCII}(A) + \text{ASCII}(G) + \text{ASCII}(C) + \text{ASCII}(A) + \text{ASCII}(C) + \text{ASCII}(G))\,\text{divide}\,17 = 28. \end{aligned} \tag{6}$$

These two patterns generate different values, although their hash function generates the same ones.

3.2. Preprocessing Phase. Our proposed algorithm is online-based, which preprocesses the pattern and keeps the text intact. First of all, our algorithm calculates the hash of the pattern using the following equation:

$$h(S) = [\text{ASCII}(S1) + \text{ASCII}(S2) + \text{ASCII}(S3) + \cdots + \text{ASCII}(\text{Sn})]\,\text{mod}\,q. \tag{7}$$

Then, the algorithm calculates the quotient of the pattern using the following equation:

$$r(S) = [\text{ASCII}(S1) + \text{ASCII}(S2) + \text{ASCII}(S3) + \cdots + \text{ASCII}(\text{Sn})]\,\text{divide}\,q. \tag{8}$$

For shifting the pattern within the text, our proposed algorithm uses the good properties of the QS algorithm. The shifting value of the QS algorithm depends on the next character of the search window. The following function below generates the next shifting distance to skip character comparisons.

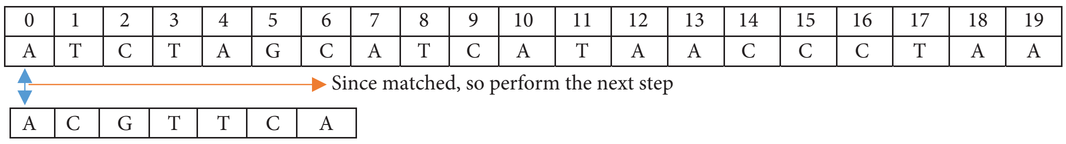
$$\text{qsBc}(x) = \begin{cases} (i\colon\ 0 \le i < m \text{ and } p[m-i] = x), & \text{if } x \text{ occurs, in } p, \\ m+1, & \text{otherwise.} \end{cases} \tag{9}$$

Here, m is the length of pattern P, and i denotes the pattern index from 0 to $m-1$. Character x serves as the text's definition for each character.

Algorithm 1 presents the preprocessing stage's pseudocode:

3.3. Searching Phase. A window of size m glides along with the text during the searching phase, starting at position 0. After each try, the window is shifted to the right until the text's conclusion is reached. First, compare the pattern's first character with the search window. If the match occurs, create a substring based on the pattern length and compute the hash value and quotient using the above hash value and quotient equation. The hash and quotient of the pattern will be compared with the substring hash value and quotient. If the substring hash value and quotient match the pattern hash value and quotient, then the final character of the pattern and substring will be compared. If matched, then the substring and pattern's leftmost second and rightmost second characters are checked concurrently. If there are any differences, the algorithm moves the pattern based on the QS table values saved during preprocessing. If no differences exist, the substring and pattern's leftmost and rightmost third characters are checked simultaneously. The rest of the characters are compared in the same way.

Algorithm 2 presents the searching stage's pseudocode:


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
A T C T A G C A T C A T A A C C C T A A
Since matched, so perform the next step
A C G T T C A

```
(1) //preprocess only pattern characters and take any prime number
(2) $q \longleftarrow$ prime number
(3) for ($i = 0$ to m) do
(4)     sum $\longleftarrow$ ASCII ($p_i$)
(5) end for loop
(6) //Generate hash value
(7) $h$ ($p$) $\longleftarrow$ sum mod $q$
(8) //Generate quotient value using predefined prime number
(9) $r$ ($p$) $\longleftarrow$ sum divide $q$
(10) //Generate QS table for the pattern
(11) set <char> alphabet set, map < char, int > QsBc
(12) set <char>:: iterator $i$
(13) for ($i$ = alphabet_set.begin() to $i$! = alphabet_set.end()) do
(14)     QsBc[$*i$] $\longleftarrow m + 1$
(15) end for loop
(16) for ($i = 0$ to pattern size) do
(17)     QsBc [pattern[$i$]] $\longleftarrow m - i$
(18) end for loop
(19) Searching ($t$, $p$)
```

ALGORITHM 1: Pre-processing of HAPM (P).

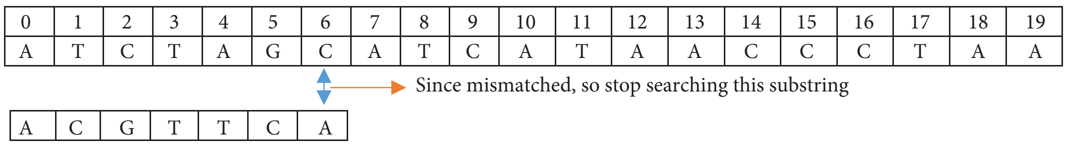
```
(1) start_ind $\longleftarrow$ 0, $d \longleftarrow m/2$, end_ind $\longleftarrow m - 1$
(2) while (text_size! = 0)
(3)     if $t$ [start_ind] = $p[0]$
(4)         //Create substring based on pattern length
(5)         for ($i = 0$ to $m$) do
(6)             sum $\longleftarrow$ ASCII ($s_i$)
(7)         end for loop
(8)         //Create hash value using predefined prime number
(9)         $h$ ($s$) $\longleftarrow$ sum mod $q$
(10)        //Create quotient value using predefined prime number
(11)        $r$ ($s$) $\longleftarrow$ sum divide $q$
(12)        if $h(p) = h(s)$ and $r(p) = r(s)$
(13)            if $t$ [start_ind + $m - 1$] = $p[m - 1]$
(14)                for ($i$ = start_ind + 1 to $d$) do
(15)                    if $t[i]! = p[i]$ or $t[m - i - 1]! = p[m - i]$
(16)                        break
(17)                    end if
(18)                end for loop
(19)            end if
(20)        end if
(21)        if every character is matched
(22)            then the pattern found occurs
(23)        end if
(24)    shift_val $\longleftarrow$ QsBc [text [end_ind + 1]]
(25)    start_ind $\longleftarrow$ start_ind + shift_val
(26)    end_ind $\longleftarrow$ start_ind + $m - 1$
(27) end while
```

ALGORITHM 2: Searching of HAPM (T, P).

Now, generate the substring and calculate the hash and quotient value of that substring:

$$
\begin{aligned}
h(s) &= (\text{ASCII}(A) + \text{ASCII}(T) + \text{ASCII}(C) + \text{ASCII}(T) + \text{ASCII}(A) + \text{ASCII}(G) + \text{ASCII}(C))\bmod 13 = 9,\\
r(s) &= (\text{ASCII}(A) + \text{ASCII}(T) + \text{ASCII}(C) + \text{ASCII}(T) + \text{ASCII}(A) + \text{ASCII}(G) + \text{ASCII}(C))\,\text{divide}\ 13 = 38.
\end{aligned}
\tag{10}
$$

Since the substring hash and quotient value are equal to the pattern hash and quotient, respectively, the following comparison is performed according to the search technique:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
A	T	C	T	A	G	C	A	T	C	A	T	A	A	C	C	C	T	A	A

A	C	G	T	T	C	A

A diagram depicting the proposed algorithm for the string-matching problem is presented in Figure 2. This diagram mainly represents the data flow of our proposed algorithm.

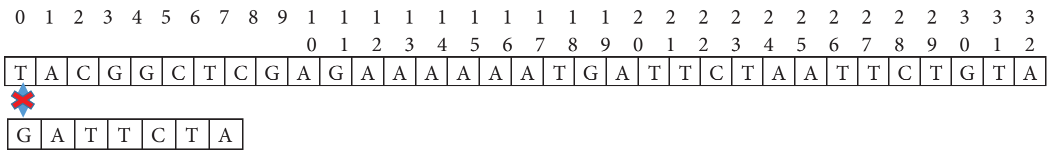
3.4. Working Example. The aloe vera plant is a succulent that retains water as a gel in its leaves. This moisturising gel is perfect for sunburns, insect bites, minor cuts and wounds, and other skin problems. The Aloe vera voucher Aloe vera chloroplast nucleotide sequence was used to test our proposed approach. According to the FASTA format, we selected a small portion of the gene's nucleotide sequence from index 4970 to 5002 (just 33 characters) [42]. The wording of the DNA sequence under consideration is as follows:

Text is t = TACGGCTCGAGAAAAAATGATTCTAAT TCTGTA, pattern is p = GATTCTA, and the prime number is 17.

$$
\begin{aligned}
h(p) &= [\text{ASCII}(G) + \text{ASCII}(A) + \text{ASCII}(T) + \text{ASCII}(T) + \text{ASCII}(C) + \text{ASCII}(T) + \text{ASCII}(A)]\bmod 17 = 10,\\
r(p) &= [\text{ASCII}(G) + \text{ASCII}(A) + \text{ASCII}(T) + \text{ASCII}(T) + \text{ASCII}(C) + \text{ASCII}(T) + \text{ASCII}(A)]\,\text{divide}\ 17 = 30.
\end{aligned}
\tag{11}
$$

The Quick Search table is shown in Table 1 for pattern p.

1st attempt:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
T	A	C	G	G	C	T	C	G	A	G	A	A	A	A	A	A	T	G	A	T	T	C	T	A	A	T	T	C	T	G	T	A

G	A	T	T	C	T	A

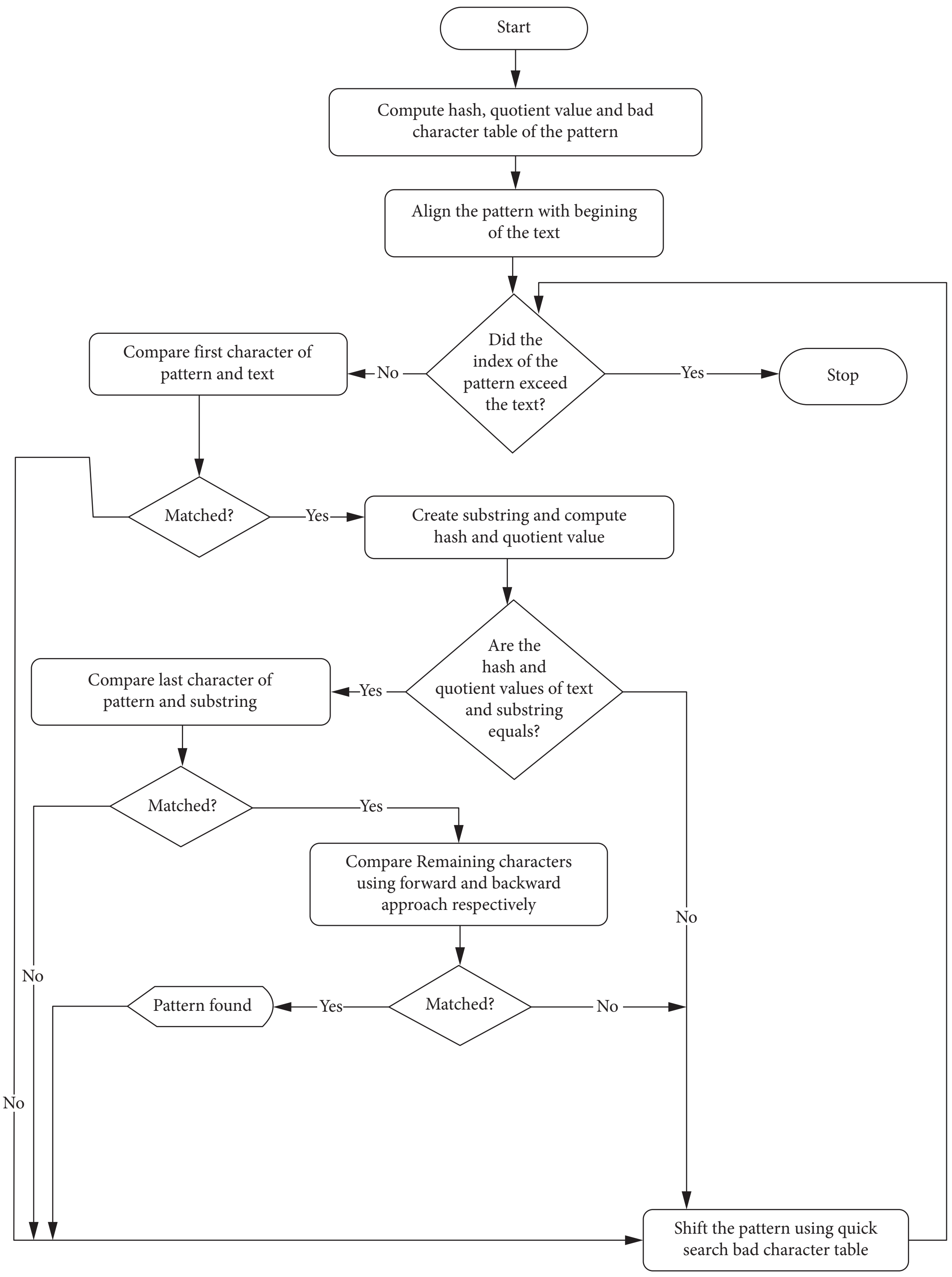
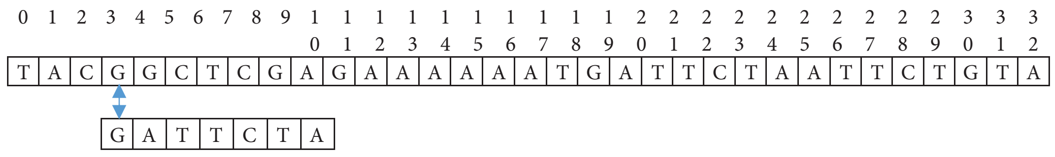

Start
Compute hash, quotient value and bad character table of the pattern
Align the pattern with begining of the text
Did the index of the pattern exceed the text?
Yes
Stop
No
Compare first character of pattern and text
Matched?
Yes
Create substring and compute hash and quotient value
Are the hash and quotient values of text and substring equals?
Yes
Compare last character of pattern and substring
Matched?
Yes
Compare Remaining characters using forward and backward approach respectively
Matched?
Yes
Pattern found
No
No
No
No
Shift the pattern using quick search bad character table

Figure 2: Flowchart of the proposed method.

Table 1: Quick Search table.

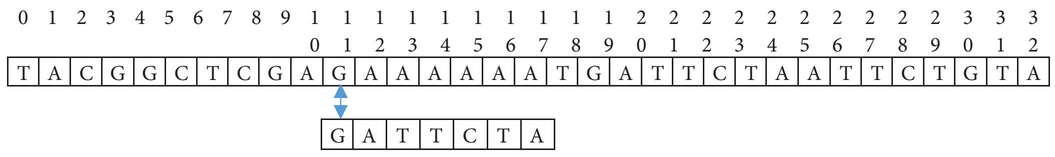
x	A	C	G	T
qsBc (x)	6	3	7	5

Shift by qsBc $[C] = 3$.
2nd attempt:



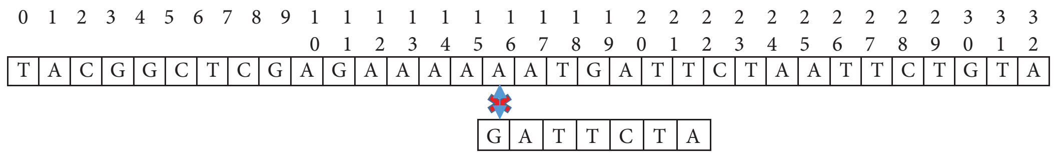
$$h(s) = [\text{ASCII}(G) + \text{ASCII}(G) + \text{ASCII}(C) + \text{ASCII}(T) + \text{ASCII}(C) + \text{ASCII}(G) + \text{ASCII}(A)] \bmod 17 = 3,$$
$$r(s) = [\text{ASCII}(G) + \text{ASCII}(G) + \text{ASCII}(C) + \text{ASCII}(T) + \text{ASCII}(C) + \text{ASCII}(G) + \text{ASCII}(A)] \text{divide } 17 = 29. \quad (12)$$

Here, $h(p)! = h(s)$ so shift by qsBc $[G] = 7$.
3rd attempt:

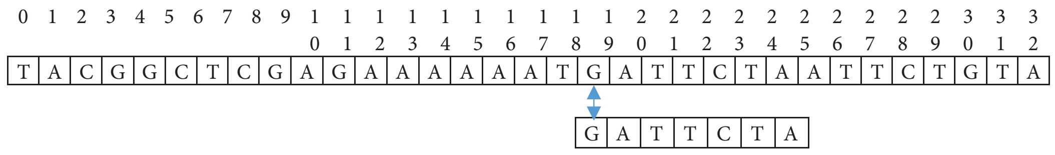


$$h(s) = [\text{ASCII}(G) + \text{ASCII}(A) + \text{ASCII}(A) + \text{ASCII}(A) + \text{ASCII}(A) + \text{ASCII}(A) + \text{ASCII}(A)] \bmod 17 = 2,$$
$$r(s) = [\text{ASCII}(G) + \text{ASCII}(A) + \text{ASCII}(A) + \text{ASCII}(A) + \text{ASCII}(A) + \text{ASCII}(A) + \text{ASCII}(A)] \text{divide } 17 = 27. \quad (13)$$

Here, $h(p)! = h(s)$ so shift by qsBc $[T] = 5$.
4th attempt:

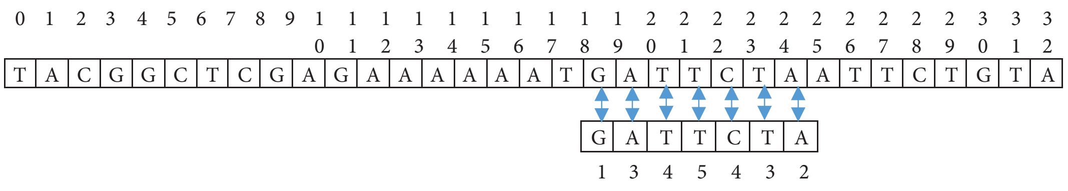


Shift by qsBc $[C] = 3$.
5th attempt:

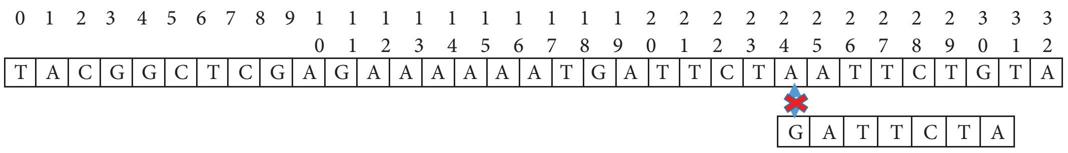


$$h(s) = [\text{ASCII}(G) + \text{ASCII}(A) + \text{ASCII}(T) + \text{ASCII}(T) + \text{ASCII}(C) + \text{ASCII}(T) + \text{ASCII}(A)] \bmod 17 = 10,$$
$$r(s) = [\text{ASCII}(G) + \text{ASCII}(A) + \text{ASCII}(T) + \text{ASCII}(T) + \text{ASCII}(C) + \text{ASCII}(T) + \text{ASCII}(A)] \text{divide } 17 = 30. \quad (14)$$

As $h(p) = h(s)$ and $r(p) = r(s)$, so perform the next step:



Match the pattern and shift by qsBc $[A] = 6$.
6th attempt:



Shift by qsBc $[T] = 5$, which exceeds the text, so stop searching.

Our proposed algorithm needs six shifts and thirteen character comparisons to find the pattern within the text.

3.5. Hashing Method Analysis. We have used only 200 MB of DNA sequence to test the effectiveness of the hashing technique in our proposed approach against various prime values [43]. Our approach divides the pattern by a predetermined prime number to determine the hash value by calculating the remainder of the patterns. The hash value is computed by the following equation:

$$h(T) = [\text{ASCII}(T1) + \text{ASCII}(T2) + \text{ASCII}(T3) + \cdots + \text{ASCII}(\text{Tn})] \bmod q. \tag{15}$$

Here, n is the pattern or substring length, and q denotes a predefined prime number.

Let a pattern $p = \text{ACGTA}$ of a DNA sequence and prime numbers 3 and 229, which were chosen randomly. The hash function or value is computed by the following equation:

$$\begin{aligned} h(p) &= (\text{ASCII}(A) + \text{ASCII}(C) + \text{ASCII}(G) + \text{ASCII}(T) + \text{ASCII}(A)) \bmod \text{ prime number} \\ &= (65 + 67 + 71 + 84 + 65) \bmod \text{ prime number} \\ &= 352 \bmod \text{ prime number}. \end{aligned} \tag{16}$$

Here, 352 mod $3 = 1$ and 352 mod $229 = 123$.

When we divide the sum by a smaller prime number (3), the remainder value gets smaller (1). For this reason, the substring hash value is becoming more and more equal to the pattern hash value. As a result, the number of attempts and comparisons increases, but when we divide the sum by a more significant prime number (229), the remainder value gets larger (123). Due to this reason, the substring hash value and pattern hash value are getting less equal. As a result, the attempts and comparisons are decreasing.

After analyzing the hashing process, it can be concluded that a more extensive prime number results in fewer attempts and comparisons, leading to favourable output. However, the prime number must be less than the sum of the ASCII values of the letters.

Figures 3–5 present a graphical representation of the experimental findings about the tallying of attempts, comparisons of characters, and length of time required for the task's completion, respectively. We have used the prime numbers 3, 7, 11, 13, 17, 39, 47, 73, 97, and 229, which were

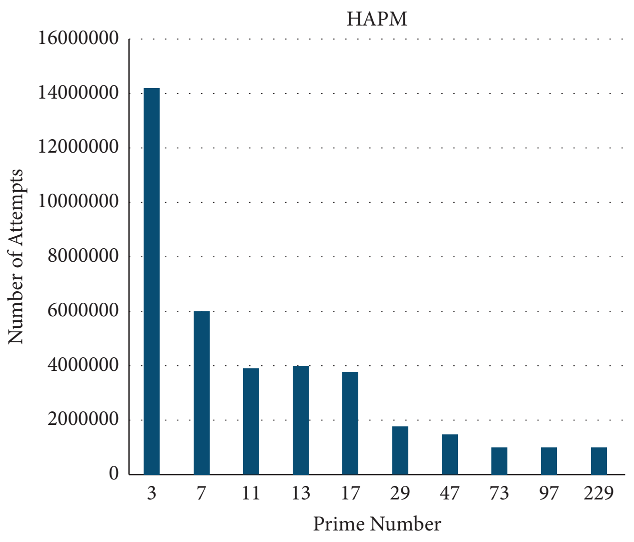


Figure 3: Number of attempts for prime numbers using DNA sequence.

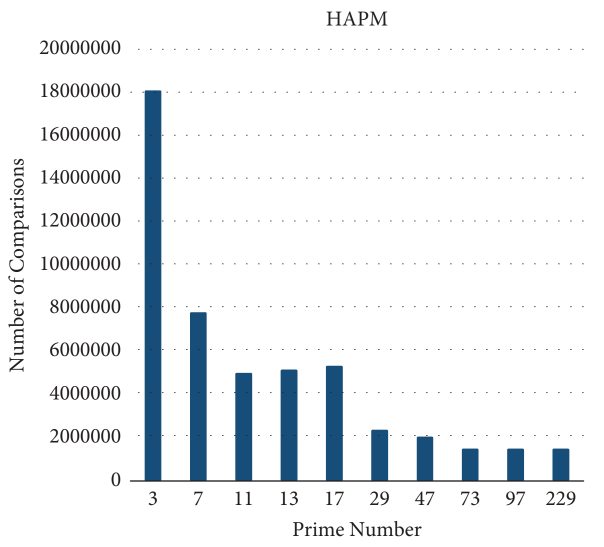


Figure 4: Number of comparisons for prime numbers using DNA sequence.

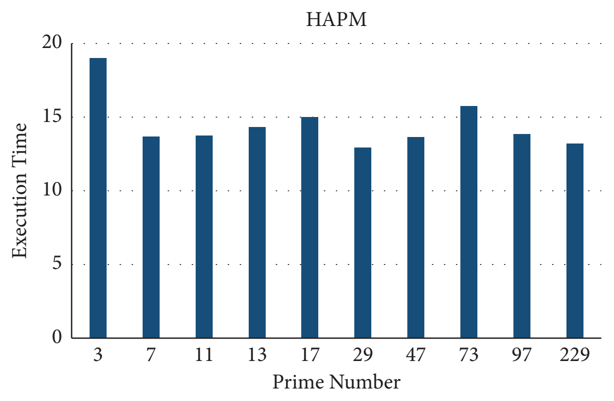


Figure 5: Number of execution time using DNA sequence.

chosen based on randomness or pseudorandomness. We ran our algorithm ten times for a single prime number to get a more efficient and accurate result. We used the pattern length of ten taken from the DNA text. Based on the obtained findings, it can be inferred that there exists an inverse relationship between the magnitude of the prime number and the number of attempts and character comparisons required. The larger the prime number, the fewer attempts and character comparisons. This is because the more significant the prime number, the fewer times the substring hash value and the pattern hash value will match. However, the impact on execution time is very insignificant. Whether the prime number is large or small, execution time is random because the residual value depends on the sum of the ASCII value and the prime number. To compute the hashing method, we employ the ASCII character and then determine the summation of its corresponding ASCII value, resulting in the generation of a significant value.

3.6. Time Complexities in Perspective and Comparison to the Proposed Algorithm.

The amount of time a statement will take to execute depends on its complexity. The preparation stage of the proposed algorithm's complexity is $O(m)$. However, the temporal complexity of the search phase can be broken down into two scenarios as follows:

Case I: In the worst-case scenario, each character in the text would appear to fit into a specific pattern. Throughout the procedure, the worst-case scenario frequently happens if the characters in the pattern match those in the following text. For example, it is stated that the worst-case complexity for the text $t =$ "LLLLLLLLLLLLLLLLLLLLLL," and the pattern $p =$ "LLLLL" is $O(nm)$.

Case II: The pattern is $p =$ "FFFFF," and the text $t =$ "SSSSSSSSSSSS" is used to evaluate the best case of the proposed search phase. The preparation stage of the proposed algorithm is directly influenced by the searching stage, where the maximum shift value is always denoted by the QS shift value, which is $(m + 1)$. To determine the best case, follow the following formula: $O(n/(m + 1))$.

Table 2 shows the time complexity of different string-matching algorithms.

4. Results and Discussion

We have used three different types of data to examine the performance of our proposed algorithm. These are the *E. coli* dataset, DNA sequence, and protein sequence. *Escherichia coli* (*E. coli*) is a small dataset with a length of 4,686,137. This dataset contains only DNA sequence letters (A, C, T, and G). Text files containing the *E. coli* dataset were acquired from the NCBI website [38, 44]. We have used a large dataset of 200 MB for DNA (the alphabet's set of 4 characters) and protein (the alphabet's set of 20 characters) sequences taken

Table 2: The time complexity of different string-matching algorithms.

Algorithms	Time complexity	Data
BF [23]	$O(mn)$	All datasets
BM [24]	$O(mn)$	All datasets
QS [29]	$O(mn)$	All datasets
BFM [30]	$O(mn)$	All datasets
MS [32]	Best case: $O(n/(m+1))$ Worst case: $O(nm)$	All datasets
MAC [33]	$O(nm)$	All datasets
EHM [19]	$O(nm)$	All datasets
Hash-q [36]	Best case: $O(n/(m-q))$ Worst case: $O(nm)$	DNA datasets
HqUF [38]	Best case: $O(n/(m-q))$ Worst case: $O(n(m-q))$	DNA datasets
HAPM	Best case: $O(n/(m+1))$ Worst case: $O(nm)$	All datasets

from the "Pizza & Chili Corpus" website [43]. For the *E. coli* dataset, pattern lengths 4, 8, 16, 32, 64, 128, 256, 512, and 1024 were chosen randomly using the HqUF algorithm. The pattern lengths for DNA and protein sequences are 3, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, and 200, which are randomly selected from the text file. To determine the performance of our proposed HAPM algorithm, 20 distinct patterns were chosen randomly for each pattern length and executed 20 times each. The C++ programming language is employed to execute the proposed algorithm. The algorithm was run utilizing code blocks version 17.12. The computer is equipped with an Intel Core i5-3210M processor operating at a frequency of 2.50 GHz. It features 8 GB of random access memory (RAM), HD Graphics 3000, and a 750 GB hard drive.

4.1. The E. coli Dataset Outcomes. Table 3 displays the average run time. From this result, it can be seen that there is a 73.04% speed increase for the pattern length 64. The lowest speed increase is found for pattern length 512, which is 15.38%. It can also be seen from the table that for small patterns (pattern length 4 to 64) the execution time of our algorithm is much less than HqUF algorithm, which means that the speed increases a lot. This is because the shorter pattern matches the text more often, so it takes longer to find the more succinct pattern.

Table 4 displays the average number of shifts or attempts. It can be seen from the table that 31.04% improvement has occurred for pattern length 256, and the lowest improvement is found for pattern length 32, which is 02.43%. Improvements vary for all other patterns. Results are often suitable for small patterns, and sometimes results are good for large patterns. The attempt depends on the shifting value of the pattern.

Table 5 displays the average number of character comparisons. This result shows that an improvement of 28.69% has been made for pattern length 1024, while the lowest improvement, 11.43%, is observed for pattern length 32. For the other patterns, the upgrades are varied. Results are frequently favourable for large patterns but can also be favorable for large and small patterns. The character comparison depends on the hashing value of the pattern.

However, our HAPM algorithm outperforms the HqUF algorithm for each pattern length regarding execution time, attempts, and character comparisons.

Table 3: Average run time using *E. coli*.

Pattern length	HqUF	HAPM	Speed up (%)
4	16.13	8.05	50.00
8	13.12	6.16	53.05
16	5.06	2.01	60.27
32	3.28	1.56	52.43
64	1.15	0.31	73.04
128	0.33	0.21	36.36
256	0.15	0.12	20.00
512	0.13	0.11	15.38
1024	0.17	0.12	29.41

Table 4: Average number of attempts using *E. coli* dataset.

Pattern length	HqUF	HAPM	Improvement (%)
4	34412.56	30124.76	12.46
8	32156.94	29901.67	07.01
16	5123.87	4524.67	11.69
32	501.76	489.56	02.43
64	204.81	190.34	07.07
128	159.34	128.39	19.43
256	178.34	122.98	31.04
512	512.89	411.67	19.74
1024	671.57	510.39	24.01

Table 5: Average number of character comparisons using *E. coli* dataset.

Pattern length	HqUF	HAPM	Improvement (%)
4	45454.24	40198.56	11.56
8	42145.23	31145.67	26.10
16	12437.89	10104.98	18.76
32	578.87	512.65	11.43
64	280.42	244.56	12.78
128	234.89	178.58	23.97
256	252.52	201.57	20.18
512	911.98	789.56	13.43
1024	1139.89	812.87	28.69

4.2. The DNA Dataset Outcomes. The efficiency of the BFM algorithm, in terms of execution time, number of attempts, and comparisons, is notably superior when compared to the Knuth–Morris–Pratt (KMP) and Boyer–Moore–Hoorspool (BMH) algorithms [30]. It is worth noting that the BMH algorithm is a simplified and enhanced version of the BM algorithm. When comparing the results of the EHM algorithm with those of the BFM, MAC, MS, and QS algorithms in terms of gaining lesser numbers of attempts and comparisons, the EHM algorithm demonstrated superior performance for both the short and long patterns [19]. The HqUF method performs better than the Hash-q method, but it works only for DNA sequences [38].

Figure 6 displays the average number of attempts. From these outcomes, it is clear that the EHM algorithm performs better than the BFM algorithm for all patterns. The HqUF algorithm performs better than the EHM algorithm for all large patterns, but the EHM algorithm performs better for small patterns. This is because the HqUF algorithm uses 2 bits for each character of DNA and matches the last q character of the pattern with the q character of the text most of the time in case of small patterns. Our HAPM algorithm shows superior results to the HqUF algorithm for all small and large patterns. It also shows better than the EHM and BFM algorithms.

Figure 7 displays the average number of character comparisons. This outcome demonstrates that the EHM algorithm is better than the BFM algorithm for all patterns, where the BFM algorithm displays an unstable behaviour. The HqUF algorithm displays better results than the EHM algorithm for all large patterns but shows the worst outcomes for small patterns such as pattern length 3. The use of the HqUF technique stems from implementing a 2-bit encoding scheme for each DNA letter. This encoding scheme facilitates the matching process between the last q characters of the pattern and the corresponding q characters in the text, mainly when dealing with smaller patterns. Our proposed HAPM as well as the HqUF algorithms show stable behaviour. However, our HAPM algorithm performs better than the HqUF, EHM, and BFM algorithms for all pattern lengths.

Figure 8 displays the average number of execution times. These findings show that the EHM algorithm's execution time is most significant for all considerable pattern lengths. However, the algorithm's performance with tiny pattern lengths reveals some encouraging signs. The EHM algorithm is a substring-based approach that takes so long to run. The time required to hash text substrings increases as the pattern length increases. The BFM algorithm shows better results than the EHM algorithm for all patterns. The HqUF approach outperforms the BFM algorithm for large patterns but underperforms for short pattern lengths due to the 2-bit encoding time required by the HqUF methodology. Our proposed algorithm shows stable behaviour and better results than the HqUF algorithm for all pattern lengths, but worse results than the EHM and BFM algorithms for short pattern lengths. The poor outcomes for small patterns are because our HAPM algorithm produces fewer shifted values for small patterns and takes longer to hash them.

4.3. The Protein Dataset Outcomes. The MS algorithm is more efficient than four other string-matching algorithms: Quick Search, Horspool, Smith, and Berry–Ravindran [32]. In addition, the MAC algorithm outperforms the MS and IBS (index-based shift) algorithms regarding the number of attempts and the total number of character comparisons [33].

Figure 9 displays the average number of attempts. Both the MS and QS algorithms exhibit unstable behaviour, with the MS algorithm doing significantly better than the QS algorithm across the board. These findings make it abundantly evident that the MS algorithm is superior to the QS algorithm. The MAC algorithm exhibits stable behaviour for all pattern lengths and outperforms the MS and QS algorithms. The BFM algorithm performs better than MAC, MS, and QS algorithms, but for some patterns (such as

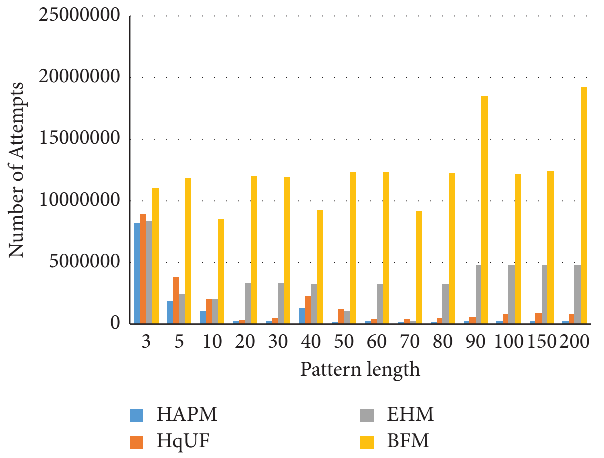


Figure 6: Average number of attempts using DNA sequence.

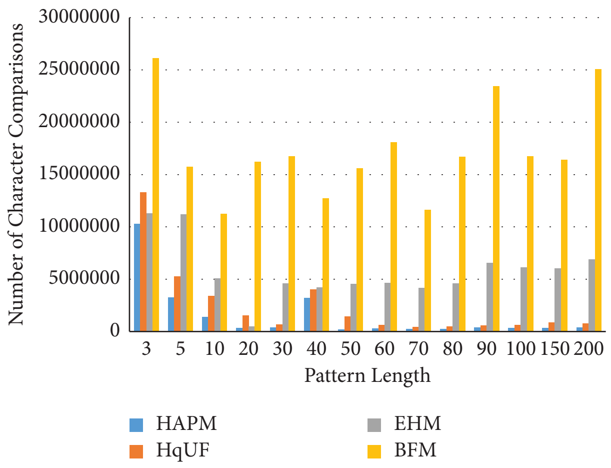


Figure 7: Average number of character comparisons using DNA sequence.

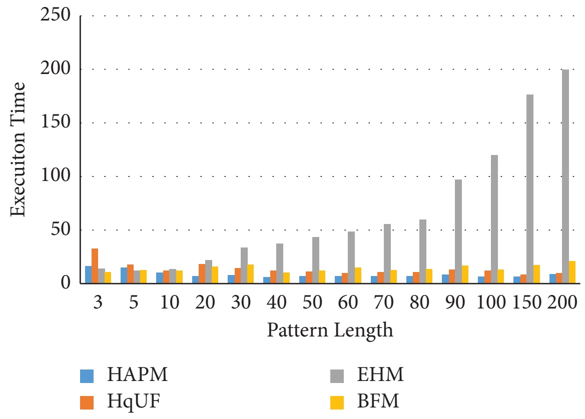


Figure 8: Average execution time using DNA sequence.

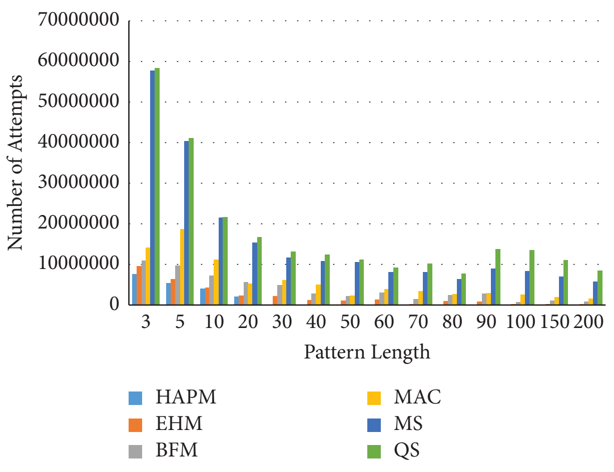


Figure 9: Average number of attempts using protein sequence.

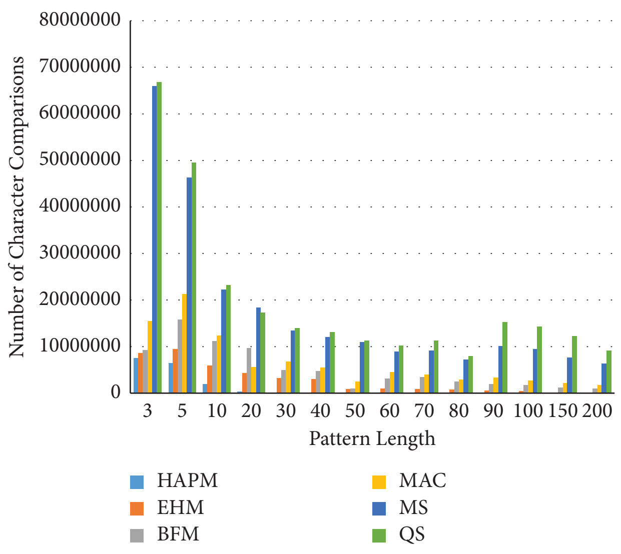


Figure 10: Average number of character comparisons using protein sequence.

pattern lengths 20, 50, and 90), it shows results almost close to the MAC algorithm. The reason for this is that the MAC algorithm is index-based. If the first character of patterns is less frequent in the text, then it requires less number of searches. The EHM algorithm offers better performance than the BFM algorithm. Our proposed HAPM algorithm outperforms EHM, BFM, MAC, MS, and QS algorithms for all pattern lengths.

Figure 10 displays the average number of comparisons. The results indicate that the performance of the MS algorithm surpasses that of the QS algorithm, while the MAC algorithm exceeds both the MS and QS algorithms across all pattern lengths. The BFM method performs better than the MAC algorithm, except for pattern length 20. In the context of the MAC algorithm, specific patterns exhibit fewer attempts when utilizing the indexing strategy. This phenomenon is the underlying cause of the reduced character comparisons required in particular patterns. Conversely, the EHM algorithm shows superior outcomes to the BFM algorithm across all pattern lengths. Our suggested HAPM algorithm outperforms the EHM, BFM, MAC, MS, and QS algorithms across all pattern lengths.

Figure 11 displays the average number of execution time. From these results, it is seen that the execution time of the QS algorithm is highest for all pattern lengths. The MS algorithm performs better than QS and MAC algorithms. Although the execution time and character comparison of the MAC algorithms are lower than the MS algorithm, the execution time is higher because the MAC algorithm is an index-based method. The preprocessing step of the MAC algorithm takes more time to index the alphabet. The execution time of the BFM algorithm is lower than MAC, MS, and QS algorithms for all patterns and lower than the EHM algorithm for all extensive pattern lengths but higher or almost equal to the EHM algorithm in execution time for some small patterns. The EHM algorithm is a substring-based hashing method whose execution time is higher than the BFM algorithm despite the small number of attempts and character comparisons. Hashing of text substrings takes longer for longer pattern lengths. Our proposed HAPM

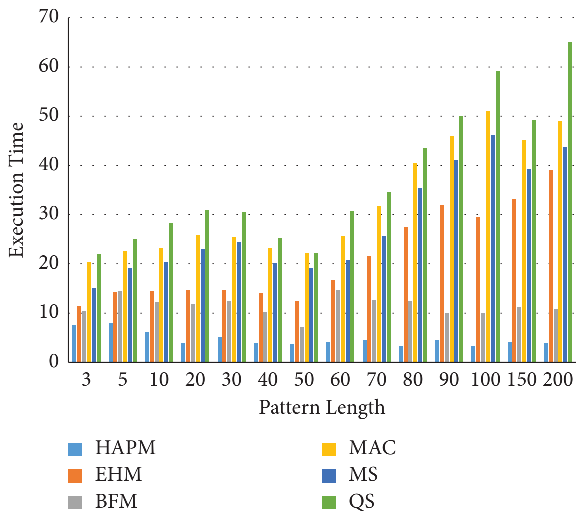


Figure 11: Average execution time using protein sequence.

algorithm shows stable behaviour and better results than all algorithms for all pattern lengths.

5. Conclusion

In computer science, string matching has grown significantly in popularity and will be crucial to future technology development. Hashing-based string-matching algorithms are increasing daily, but the most vital objective is reducing hash collisions. We have proposed a hashing-based algorithm that has reduced hash collisions. Three different data types are used to test our proposed algorithm's performance, and 20 distinct patterns for each pattern length are randomly selected from the dataset. Their average value is taken after executing each of them 20 times. We implemented six alternative algorithms to evaluate the performance of our approach and tested them on the dataset. 73.04% speed-up, 31.04%, and 28.69% improvement have been achieved for average run time, the average number of shifts, and comparisons, respectively, comparing our proposed HAPM algorithm with the HqUF method on the *E. coli* dataset. Our algorithm performs better for DNA and protein datasets than the previous algorithms in terms of an average number of attempts and comparisons. Still, some cases show worse results for some short patterns regarding an average number of execution times. In future research, we will create fresh strategies based on the suggested hash function to speed up the execution of short patterns.

Data Availability

All the data used to support the findings of this study are made available online.

Conflicts of Interest

The authors declare that there are no conflicts of interest.

Acknowledgments

This work was funded by the division of Information and Communication Technology (ICT), Ministry of Posts, Telecommunications and Information Technology, Government of People's Republic of Bangladesh.

References

[1] K. Vayadande, M. Ram, K. Paralkar, D. Pawal, S. Deshpande, and V. Sonkusale, "Pattern matching in file system," *International Journal of Computer Application*, vol. 975, no. 2022, p. 8887.

[2] P. Neamatollahi, M. Hadi, and M. Naghibzadeh, "Simple and efficient pattern matching algorithms for biological sequences," *IEEE Access*, vol. 8, pp. 23838–23846, 2020.

[3] P. Niroula and Y. Nam, "A quantum algorithm for string matching," *Npj Quantum Information*, vol. 7, no. 1, p. 37, 2021.

[4] F. Mohammed and N. H. Al-Kumaim, "A survey of the hybrid exact string matching algorithms," *Advances on Intelligent Informatics and Computing: Health Informatics, Intelligent Systems, Data Science and Smart Computing*, vol. 127, p. 173, 2022.

[5] A. D. Baxevanis, "Biological sequence databases," *Bioinformatics*, John Wiley & Sons, New York, NY, USApp. 1–18, 2020.

[6] O. A. S. Ibrahim, B. A. Hamed, and T. A. El-Hafeez, "A new fast technique for pattern matching in biological sequences," *The Journal of Supercomputing*, vol. 79, no. 1, pp. 367–388, 2023.

[7] V. Y. Gudur and A. Acharyya, "Hardware-software codesign based accelerated and reconfigurable methodology for string matching in computational bioinformatics applications," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 17, no. 4, pp. 1198–1210, 2020.

[8] B. A. Hamed, O. A. S. Ibrahim, and T. Abd El-Hafeez, "A survey on improving pattern matching algorithms for biological sequences," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 26, Article ID e7292, 2022.

[9] Q.-Y. Jiang and W.-J. Li, "Asymmetric deep supervised hashing," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[10] O. Jafari, P. Maurya, P. Nagarkar, K. Mushfiqul Islam, and C. Crushev, "A survey on locality sensitive hashing algorithms and their applications," 2021, https://arxiv.org/abs/2102.08942.

[11] S. I. Hakak, A. Kamsin, P. Shivakumara et al., "Exact string matching algorithms: survey, issues, and future research directions," *IEEE Access*, vol. 7, pp. 69614–69637, 2019.

[12] I. Markić, M. Štula, M. Zorić, and D. Stipaničev, "Entropy-based approach in selection exact string-matching algorithms," *Entropy*, vol. 23, no. 1, p. 31, 2020.

[13] S. Faro and S. Scafiti, "A weak approach to suffix automata simulation for exact and approximate string matching," *Theoretical Computer Science*, vol. 933, no. 2022, pp. 88–103, 2022.

[14] J. Rekha, "Approximate multiple string matching algorithm," *Journal of Theoretical and Applied Information Technology*, vol. 98, no. 11, 2020.

[15] C. Someswara Rao and K. Butchi Raju, "Single and multiple pattern string matching algorithm," *Indian Journal of Science and Technology*, vol. 10, p. 3, 2017.

[16] S. Song, G. Gu, C. Ryu, S. Faro, T. Lecroq, and K. Park, "Fast algorithms for single and multiple pattern Cartesian tree matching," *Theoretical Computer Science*, vol. 849, no. 2021, pp. 47–63, 2021.

[17] A. Cinti, F. M. Bianchi, A. Martino, and A. Rizzi, "A novel algorithm for online inexact string matching and its FPGA implementation," *Cognitive Computation*, vol. 12, no. 2, pp. 369–387, 2020.

[18] M. Waga, I. Hasuo, and K. Suenaga, "Efficient online timed pattern matching by automata-based skipping," in *Proceedings of the International Conference on Formal Modeling and Analysis of Timed Systems*, pp. 224–243, Berlin, Germany, September 2017.

[19] P. Mahmud, A. Rahman, and K. H. Talukder, "An efficient hashing method for exact string matching problems," in *Proceedings of the Data Intelligence and Cognitive Informatics: Proceedings of ICDICI 2021*, pp. 289–301, Tirunelveli, India, July 2022.

[20] T. Fukač, J. Matoušek, K. Jan, and L. Kekely, "Increasing memory efficiency of hash-based pattern matching for high-speed networks," in *Proceedings of the 2021 International Conference on Field-Programmable Technology (ICFPT)*, pp. 1–9, IEEE, Auckland, New Zealand, December 2021.

[21] A. M. Al-Ssulami, A. M. Azmi, H. Mathkour, H. Aboalsamh, and H. Aboalsamh, "LsHASHq: a string matching algorithm exploiting longer q-gram shifting," *Information Processing and Management*, vol. 59, no. 5, Article ID 103057, 2022.

[22] A. A. Karcioglu and H. Bulut, "The WM-q multiple exact string matching algorithm for DNA sequences," *Computers in Biology and Medicine*, vol. 136, no. 2021, Article ID 104656, 2021.

[23] C. Baturu and Naufal abdi, "Brute force algorithm implementation of dictionary search," *Jurnal Info Sains: Informatika dan Sains*, vol. 10, no. 1, pp. 24–30, 2020.

[24] Layustira, V. Ardelia, and W. Istiono, "Comparative analysis of brute force and boyer moore algorithms in word suggestion search," *International Journal*, vol. 9, no. 8, 2021.

[25] D. Purba, Z. Matondang, H. Manalu, L. Sitorus, and M. Sagala, "The application Boyer Moore algorithm to answered crossword puzzle," *AIP Conference Proceedings*, vol. 2798, no. 1, 2023.

[26] A. A. Ojugo, D. A. Oyemade, and D. A. Oyemade, "Boyer moore string-match framework for a hybrid short message service spam filtering technique," *IAES International Journal of Artificial Intelligence*, vol. 10, no. 3, p. 519, 2021.

[27] Y. Duan, H. Long, and Q. Yu, "Application of improved BM algorithm in string approximate matching," *Procedia Computer Science*, vol. 166, pp. 576–581, 2020.

[28] R. K. Pandey and S. Taruna, "Prevalent exact string-matching algorithms in natural language processing: a review," *Journal of Physics: Conference Series*, vol. 1854, no. 1, Article ID 12042, 2021.

[29] A. A. Almazroi, F. Mohammed, M. A. Qureshi et al., "A hybrid algorithm for pattern matching: an integration of berry-ravindran and raita algorithms," in *Proceedings of the International Conference of Reliable Information and Communication Technology*, pp. 160–172, Jaipur, India, September 2021.

[30] M. O. Al-Faruk, K. M. A. Hussain, M. A. Shahriar, S. M. Tonni, and S. Mahjabin Tonni, "BFM: a forward backward string matching algorithm with improved shifting for information retrieval," *International Journal on Information Technology*, vol. 12, no. 2, pp. 479–483, 2020.

[31] Al-Dabbagh, S. S. Mahmood, and Y. M. Abdal, "Parallel hybrid string matching algorithm using CUDA API function," in *Proceedings of the 2021 International Conference on Computing and Communications Applications and Technologies (I3CAT)*, pp. 66–70, IEEE, Ipswich, UK, September 2021.

[32] Kadhim, H. Adil, and N. A. AbdulRashidx, "Maximum-shift string matching algorithms," in *Proceedings of the 2014 International Conference on Computer and Information Sciences (ICCOINS)*, pp. 1–6, IEEE, Kuala Lumpur, Malaysia, June 2014.

[33] P. Mahmud, M. S. Rana, and K. Hasan Talukder, "An efficient hybrid exact string matching algorithm to minimize the number of attempts and character comparisons," in *Proceedings of the 2018 21st International Conference of Computer and Information Technology (ICCIT)*, pp. 1–6, IEEE, Dhaka, Bangladesh, December 2018.

[34] A. Dilobar and S. J. Karimovich, "RABIN-KARP algorithm in algorithms," *Journal of Integrated Education and Research*, vol. 2, no. 9, pp. 94–99, 2023.

[35] A. Putera Utama Siahaan, S. Aryza, E. Hariyanto, A. Hasudungan Lubis, A. Ikhwan, and P. Len Eh Kan, "Combination of levenshtein distance and rabin-karp to improve the accuracy of document equivalence level," *International Journal of Engineering & Technology*, vol. 7, no. 2.27, pp. 17–21, 2018.

[36] C. Ryu, T. Lecroq, and K. Park, "Fast string matching for DNA sequences," *Theoretical Computer Science*, vol. 812, pp. 137–148, 2020.

[37] A. A. Karcioglu and H. Bulut, "q-frame hash comparison based exact string matching algorithms for DNA sequences," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 9, Article ID e6505, 2022.

[38] A. A. Karcioglu and H. Bulut, "Improving hash-q exact string matching algorithm with perfect hashing for DNA sequences," *Computers in Biology and Medicine*, vol. 131, Article ID 104292, 2021.

[39] A. R. Chayapathi, "Survey and comparison of string matching algorithms," *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, vol. 12, no. 12, pp. 1471–1491, 2021.

[40] B. Branchini, S. Breschi, A. Zeni, and M. D. Santambrogio, "Fast genome analysis leveraging exact string matching," in *Proceedings of the 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 136–139, IEEE, Lyon, France, June 2022.

[41] K. Duvvuri, P. N. Reddy, D. R. Harshitha Kanisettypalli, and T. V. Nidhin Prabhakar, "Comparative analysis of pattern matching algorithms using DNA sequences," in *Proceedings of the 2022 IEEE 2nd Mysore Sub Section International Conference (MysuruCon)*, pp. 1–5, IEEE, Mysuru, India, October 2022.

[42] Ncbi, "National center for biotechnology information," 2023, https://www.ncbi.nlm.nih.gov/genome/?term=Aloe+vera+chloroplast.

[43] Pizzachili, "The Text Collection," 2023, http://pizzachili.dcc.uchile.cl/texts.html.

[44] G. Plunkett, "*Escherichia coli* str. K-12 substr. DH10B, complete sequence dataset, NCBI data," 2023, https://www.ncbi.nlm.nih.gov/nuccore/NC_010473.1.