



Research Article

Lightweight Pattern Matching Method for DNA Sequencing in Internet of Medical Things

J. A. M. Rexie ¹, Kumudha Raimond,¹ Mythily Murugaaboopathy,¹ D. Brindha,¹
and Henock Mulugeta ²

¹Department of Computer Science and Engineering, Karunya Institute of Technology and Sciences, Coimbatore, India

²Computer Engineering, Cybersecurity and AI School of Information Technology and Engineering (SiTE),
Addis Ababa Institute of Technology, Addis Ababa, Ethiopia

Correspondence should be addressed to J. A. M. Rexie; rexievimalphd@gmail.com and Henock Mulugeta; henock.mulugeta@aait.edu.et

Received 23 May 2022; Revised 28 June 2022; Accepted 29 July 2022; Published 8 September 2022

Academic Editor: Vijay Kumar

Copyright © 2022 J. A. M. Rexie et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

An area of medical science, that is, gaining prominence, is DNA sequencing. Genetic mutations responsible for the disease have been detected using DNA sequencing. The research is focusing on pattern identification methodologies for dealing with DNA-sequencing problems relating to various applications. A few examples of such problems are alignment and assembly of short reads from next generation sequencing (NGS), comparing DNA sequences, and determining the frequency of a pattern in a sequence. The approximate matching of DNA sequences is also well suited for many applications equivalent to the exact matching of the sequence since the DNA sequences are often subject to mutation. Consequently, recognizing pattern similarity becomes necessary. Furthermore, it can also be used in virtually every application that calls for pattern matching, for example, spell-checking, spam filtering, and search engines. According to the traditional approach, finding a similar pattern in the case where the sequence length is l_s and the pattern length is l_p occurs in $O(l_s * l_p)$. This heavy processing is caused by comparing every character of the sequence repeatedly with the pattern. The research intended to reduce the time complexity of the pattern matching by introducing an approach named “optimized pattern similarity identification” (OPSI). This methodology constructs a table, entitled “shift beyond for avoiding redundant comparison” (SBARC), to bypass the characters in the texts that are already compared with the pattern. The table pertains to the information about the character distance to be skipped in the matching. OPSI discovers at most spots of similar patterns occur in the sequence (by ignoring ϵ mismatches). The experiment resulted in the time complexity identified as $O(l_s, \epsilon)$. In comparison to the size of the pattern, the allowed number of mismatches will be much smaller. Aspects such as scalability, generalizability, and performance of the OPSI algorithm are discussed. In comparison with the hamming distance-based approximate pattern matching algorithm, the proposed algorithm is found to be 69% more efficient.

1. Introduction

Pattern matching is of two kinds based on the scenario for which it is applied: exact pattern matching (EPM) and approximate pattern matching (APM). An EPM is highly needed for the scenario in which the accuracy expected is 100%. For instance, when there is a search of a record in a database using a key value, exact matching is mandatory. Equally, APM finds its application in the fields like Bioinformatics, web search engines, text mining, intrusion detection system [1], and spam filtering. One of the interesting applications of string matching

is text mining and which is discussed in [2] for extracting health-related information from Twitter messages.

In Bioinformatics, identifying similar patterns has a major part in sequence alignment, sequence assembly, a search of patterns in a DNA sequence, sequence comparison, and many more. For detecting a similar pattern, EPM has to be modified to ignore a few mismatches. The mismatch may be an insertion of a new character or deletion of a character or substitution by another character.

The problem of finding EPM was approached in different aspects and given solutions. The objective of the different

tries on the same problem is to reduce the computation time taken for the solution. The time complexity for the worst case is in the order of the product of the length of the sequence and the pattern. Among the many solutions, the algorithm given by Knuth–Morris–Pratt (KMP) [3] proved to be having a linear time complexity, i.e., in the order of the length of the sequence alone. The proposed approach, optimized pattern similarity identification (OPSI), aims at finding the pattern similarity which needs to allow a few permissible counts of mismatches. The algorithm applies a similar computation of shift value as the KMP algorithm in the preprocessing stage. Then, in the proceeding stages, the algorithm identifies the pattern similarity with less than ϵ mismatches.

In medicine, a term called next-generation sequencing (NGS) refers to a high throughput method of sequencing. The sequencing technologies have the fundamental characteristic of extensively parallelizing DNA molecule sequencing in flow cells [4]. After the alignment or assembly of short or long reads, DNA sequences can be utilized for any application. The short read is a part of the DNA sequence of length 100 to 300 base pairs produced by NGS technology [5]; and, long read lengths range from 500 bp to 800 bp [6]. NGS technologies have advanced to generate genomes that are longer than these lengths. The reads are mapped to the reference genome using bioinformatics analysis [7]. There are many DNA alignment algorithms proposed for merging the short reads into a single read and these algorithms are mainly divided into assembly and alignment of NGS data. Assembly of the short reads will be done in the absence of reference sequence [8, 9]; alignment will be done with the reference sequence as given in [10–13].

Read alignment is affected by some sequencing technology parameters, including the read length and error rate. It is essential to determine the actual location of each read, according to the reference. The exact location cannot be determined in advance, so the matching has to be done approximately [14]; the match must take into account a few mismatches and missing pieces within the reference sequence. The use of approximate matching would be unnecessary if there were no repeats in a genome and no errors introduced by a sequencing experiment. As long as there are sufficient read lengths relative to the genome size, exact matching can be applied to determine where the true locations are. Unfortunately, both assumptions are inaccurate: Eukaryotic genomes contain repeated elements, and each sequencing process has errors inherent in it [15]. The true location of a read is not accessible if it lies entirely within a perfectly identical repeat sequence. There are exceptions to this. Moreover, errors can occur in the reads, so the string matching needs to be approximated. There are two types of error models commonly used: 1. The hamming distance manages the mismatches between sequence reads and the genomic region of interest, and 2. The edit distance takes into account indels and mismatches [16].

An alternate option to applying the edit distance, which handles all mismatches, insertions, and deletions equally, is to employ the weighted edit distance, which allows for the differentiation between mismatches and insertions, by

assigning weights to errors. Even position-specific errors can be weighted differently when a weighted edit distance is developed by taking quality values into account. The quality value represents the probability that the base call will be mistaken [17]. Apart from these error models, affine gap can also be applied to longer indels, but at the expense of increased computational costs [18].

1.1. Motivation. There are similarities in genetic sequences among organisms (but different from one another). Because of point mutations, only a few positions are different between the organisms. These mutations can be nucleotide substitutions (another letter in the sequence) or insertions or deletions of some nucleotides. These sequence comparisons have been used to answer many biological questions, including the study of new Coronavirus strains, the prediction of genetic diseases, and the identification of cancerous cells. In order to do this, algorithms must be applied to align the sequences. For two sequences, this may not be a problem, but for hundreds or thousands, it may involve substantial computational effort.

Genome sequencing is a big reason for the current progress in genetics. Alternatively, it is known as a massively parallel sequencer. These technologies involve breaking the genetic sequence (that has a few billion nucleotides in an individual) into hundreds of nucleotides-long pieces and simultaneously sequencing them. Then, one has to reassemble the sequences, either by comparing the reads with a sequence for an organism, that is, similar (sequence mapping) or by assembling the reads based on similarity (sequence assembly). It is possible to achieve this through many sophisticated bioinformatic tools, including BWA [19], bowtie2 [20], TopHat [21], and Megahit [21], for example.

Blasting is the process of finding out what sequence belongs to a particular gene of a given organism by comparing the sequence to a large database. The BLAST [22] program is one of the many tools available to do this.

The crucial component of the algorithm in these entire DNA sequencing applications is pattern matching. Exact matches will not always yield the best results. Because of the source material and the process by which the sequences were created, there may be errors in the sequences. Even if the sequence and pattern are not identical, it will still be efficient to compare them, as long as the count of mismatched characters does not exceed a given threshold [23, 24]. The performance of the alignment algorithms is influenced by the performance of the pattern matching process. The impact on time motivated us to propose a time-optimized algorithm for approximate pattern matching.

The applications, which are depending on the similarity between the DNA sequences, greatly rely on the sequence alignment algorithms. Among these applications is the manufacture of drugs against Coronavirus. [25] Developed a method for aligning two Coronavirus sequences and identifying where mutations may have been occurring in the viral DNA. In the process of producing medicines to treat diseases, this information can be used. Furthermore, alignment algorithms are used for the correction of reading errors

[26, 27]. KMP algorithm for pattern recognition, it helps to reduce the worst-case complexity, thus it helps to produce better alignment and better accuracy classification in the field of DNA sequence classifications [28]. To lower the make span rate, it is thought that the Internet of Health Things (IoHT) schedule must be balanced. For the optimum hybrid moth flame optimization (HMFO) job scheduling for cloud computing integrated with the IoHT environment over e-healthcare systems, we created a smart model technique in this study. Due to the anomalous changes in node energy levels, mobile nodes offer unreliable communication among themselves. It is so challenging to control such node actions that target node fails to gather packets [29–38].

The paper is organized as follows: Section 2 contains a literature survey; the EPM and OPSI algorithm are presented and demonstrated with examples in Section 3; Section 4 presents the result and discussion of OPSI algorithm, and Section 5 concludes the work presented.

2. Related Work

The focus of inventions of new pattern matching algorithms is to optimize the count of comparisons between the characters of the text and the pattern. Pattern matching can be mainly of two varieties, such as EPM and approximate pattern matching. The EPM problem is to find the occurrences of a pattern of length l_p in a text of length l_s . Let the substring of length l_p in the text be called a window.

In the Brute Force approach [39], the pattern is aligned with the first window (i.e., the first l_p characters of text). Each character in the window is compared with the respective characters of the pattern. When a mismatch occurs, the window shifts by one position, i.e., the window starts at the second character and ends at the $l_p + 1$ st character of the text. The character comparison continues between the window and the pattern. Since there is no preprocessing of the text or the pattern, this approach has the time complexity of $O(nm)$ for the worst-case combination of text and pattern.

There are pattern matching algorithms using the hash-based method. One of the methods proposed earlier using hashing is the Rabin–Karp algorithm [40]. The advantage of hash-based methods is that it encodes the characters into integers. Since the comparison has to be done only on the hash code and not on the strings, the comparison of numbers is time-efficient. However, the time complexity of the algorithm is $O(mn)$; also, the computation of the hash code is complex in the Rabin–Karp algorithm. To improve efficiency, Zhao and Liu [41] suggested an algorithm that considers a partial number of bits in the binary representation of the character to compute the hash code.

Levenshtein algorithm is incorporated into the Rabin–Karp algorithm and was proposed as revised Rabin–Karp algorithm [42]. Hash distances are calculated for both patterns and text, resulting in an improved level of accuracy. Pattern and text are input parameters into the algorithm, and the Rabin–Karp hash value is used as the output parameter. The Levenshtein method is applied to determine whether the sequence is similar. A similarity level is calculated by Levenshtein by comparing two input reads. And

also, it is used to calculate the minimum effort needed to transform one input data to another string by doing minimum modifications. Using the combination of improved Rabin–Karp algorithm, a Levenshtein algorithm, and additional check of quotient, it leads to improved accuracy and efficiency. However, the issue faced by hash-based algorithms is the collisions, i.e., more than one string is generating the same hash code.

Bloom filter supports a space-efficient data structure based on hashing. Bloom filters were used for representing the de Bruijn graph in less amount of space compared to the space originally required for storing the graph [43]. De Bruijn graph is applied for the assembly of DNA sequences produced by NGS. Najam et al. [44] proposed a pattern matching algorithm using multiple bloom filters. The method assures finding all positions of the pattern in the text in the compressed text itself.

To improve the performance of the Brute Force approach of pattern matching, improvised algorithms were proposed by preprocessing the pattern to take an optimized shift of the window. The KMP algorithm [3] and Boyer–Moore (BM) algorithm [45] preprocessed the pattern to compute the shift position to be used to move the window when there is a character mismatch. Boyer–Moore combines two approaches, bad character heuristic and good suffix heuristic. Each of these heuristics can be used independently of the other in order to find a pattern in a text. The pattern is processed and the two heuristics are mapped into different arrays. The pattern is shifted every time by the maximum suggested by each of the two heuristics. Each step consists of calculating the greatest offset suggested by the two heuristics. In contrast to other pattern matching algorithms, the Boyer–Moore algorithm initiates the matching from the last character of the pattern.

Bad character heuristics are easy to understand. Assume that there is a character in a text that never occurs in a pattern. The pattern can be altered by changing the “bad character” to begin matching from substrings next to this character when it does not match (i.e., when a mismatch occurs at this character). As an alternative, it is possible that a bad character exists in the pattern; in this case, choose among bad characters in the text. As a result, the shift is likely to be higher than one.

Suffixes that have matched successfully are good suffixes. A mismatch that has a negative shift in bad character heuristics leads to an onward jump equal to the length of the suffix found in the substring of the pattern matched until the bad character. The average case complexity of both algorithms is $O(l_s)$. BM algorithm executes faster for an alphabet set of average size and lengthy patterns. But for the pattern of long length, preprocessing time is getting increased.

Reference [46] modified KMP algorithm to suit for searching an encoded pattern in Huffman encoded text. Since encoded contents are handled, there are chances for false matches in encoded text, i.e., the match found may not be an original match of the pattern in the text. He modified the KMP algorithm to be adaptable for handling binary strings. Also, since the BM algorithm is advisable for a larger alphabet set, KMP is chosen for the binary string.

As an alternate to calculate shift positions for the pattern, suffix trees were introduced and widely used in pattern matching algorithms [47–49]. The suffix tree is a data structure that will have the starting positions of each suffix of length one to the length of the text. An example is shown in Figure 1(b) depicting the suffix tree for the text, $t = \text{“CCACTGG.”}$ The suffix tree has to be created for the whole text in which the pattern has to be searched. The amount of space needed for the suffix tree representation may be around 10 to 20 bytes per character [48]. Since the DNA sequence is very large in the range of billions, the space needed for the suffix tree will be more. Hence, there are many algorithms proposed for the compressed representation of suffix trees. Also, suffix trees were converted to suffix arrays to overcome the space issue faced by suffix trees [50–52]; but at the same time, efficiency was retained. The suffix array represents similar index information as the suffix tree but in the format of an array. The computation of the suffix array is exhibited in Figures 1(c) and 1(d).

The suffix array of the text (t), “CCACTGG,” i.e., SA (text) will be $\{7, 0, 5, 3, 1, 6, 4, 2\}$. Any pattern to be searched in the text will be a prefix of any of the suffixes if the pattern is present in the text. Hence, to search a pattern, the binary search is applied, and the comparison of the pattern begins with the suffix at the middle position of SA. For example, if the pattern (p) to be searched is “GTA,” p is compared with t [SA [3]], which is “GTGTA.” Since the comparison fails and p is lesser than “GTGTA,” the binary search will be continued in the first half of the sorted array. The complexity of the search is $\log I_s$ since the binary search is applied. However, since the suffix array has to be sorted, the complexity of the method is $O(I_s \log I_s)$.

The research on pattern matching is not only into different data structures but also into suggesting hardware-related solutions to improvise performance. Özcan and Ünsal [53] proposed hardware-based solution for bitwise string matching and proved that it performs well.

The character-based algorithm, KMP, is having better time efficiency. The key idea of this efficiency is the computation of the shift value for the pattern. The same is applied for preprocessing the pattern and is utilized for identifying pattern similarity.

3. Research Methods

3.1. Exact Pattern Matching. Knuth et al. [3], (KMP), proposed an algorithm for EPM, which scans the characters of the given sequence only once from left to right, and the characters of the pattern were aligned to a character in the text according to the previous comparison. Hence, the time complexity will be in the linear order of the length of the sequence in which the pattern is searched for.

The algorithm makes sure that the characters of the sequence are scanned once. It is assured by preprocessing the pattern to calculate the shift position. The shift position indicates the next window for further comparison in case of any mismatch. The shift position is assigned such that the sequence of characters compared already with the pattern in

the current window of the sequence will not be compared again if a mismatch occurs.

3.2. Preprocessing the Pattern. Let DNA sequence be of length l_s and the pattern of length l_p . The straightforward algorithm for EPM makes a greater number of comparisons since there is an absence of observing the pattern before the comparison. Directly, the pattern is aligned at the left end of the sequence and character comparison starts. To enhance the solution, the pattern is scanned from left to right and observed for the repetition of any prefixes in the substrings. This observation is recorded in the shift beyond for avoiding redundant comparison (SBARC) table. The table consists of the shift value for the characters of the pattern. This assists in escaping from the redundant comparison in the occurrences of recurring characters.

Let DNA sequence be of length l_s and the pattern of length l_p . The straightforward algorithm for EPM makes a greater number of comparisons since there is an absence of observing the pattern before the comparison. Hence, the time complexity of this solution is $O(l_s * l_p)$. For small l_p , this is affordable. This becomes problematic as l_p increases. There is a lot of information available once it is checked whether $S[i, \dots, i + l_p - 1]$ matches P or not, so we can determine whether $S[i + 1, \dots, i + l_p]$ fits P . There is one way to deal with this problem at a high level. This is to design a deterministic finite automaton (DFA) that depends on the pattern [54, 55]. The state diagram comprises $l_p = |P|$ states. This automaton receives the sequence S . If a sequence of j characters from the pattern has already matched at the location, where they are currently in the sequence, the DFA guarantees that we will be in the j^{th} state. Now, the next two characters will be examined. The next state is reached if a match occurs, i.e., $j + 1$ characters are matched. An error can be resolved by going back to some earlier state. It is needed to make an intelligent decision about that earlier state. It will be an incorrect choice if the initial state in the DFA is selected. As based on the information about all possible states, the possible furthest back state has to be chosen. Suppose the pattern, $P = \text{AAC}$, then the DFA will be as shown in Figure 2.

As soon as we see the pattern, the final state (S_3) would be reached. As a result, the location at which the pattern P appears the first time in sequence S . When finding all instances of P , the DFA would need to be adjusted as shown in Figure 3.

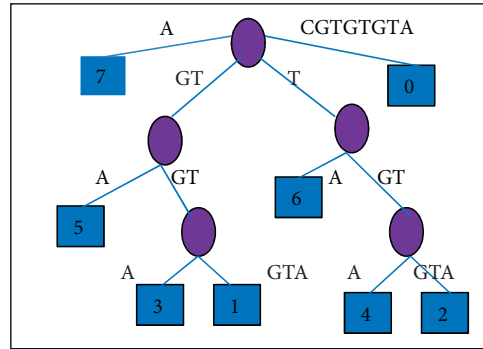
The current location in S would be output every time the final state is reached, and then all instances of P in S could be output.

Having designed the DFA, it takes just $O(l_s)$ time to feed the sequence S and the positions in S into the DFA each time the final state is reached. The total run-time of the pattern matching algorithm would be $O(l_p) + O(l_s)$ if building the DFA takes $O(l_p)$ time.

Let a pattern P be $p_0 p_1 p_2 \dots p_{l_p - 1}$, and P_k represents the prefix $p_0 p_1 \dots p_k$. In each prefix P_k there is one state q_k . Being in the state $q_k \dots$ the state of q_k , the next state has to be decided when the character c is seen after p_k . If c matches p_{k+1} , then the state relevant to $P_{k+1} = p_0 p_1 \dots p_k p_{k+1}$ will be

Index	0	1	2	3	4	5	6	7
Text	C	G	T	G	T	G	T	A

(a)



(b)

Start Index	Suffix
0	CGTGTGTA
1	GTGTGTA
2	TGTGTA
3	GTGTA
4	TGTA
5	GTA
6	TA
7	A

(c)

Start Index	Suffix
7	A
0	CGTGTGTA
5	GTA
3	TGTA
1	GTGTGTA
6	TA
4	TGTA
2	TGTGTA

(d)

FIGURE 1: (a) Sample text and index. (b) Suffix tree. (c) All possible suffixes with start index. (d) Sorted suffixes.

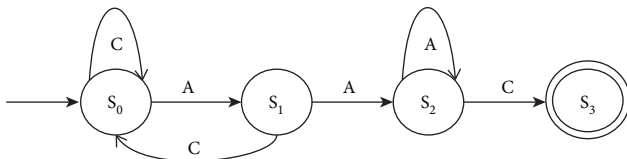


FIGURE 2: DFA for the pattern "AAC."

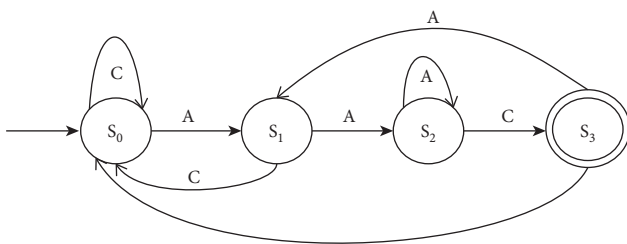


FIGURE 3: DFA for finding all repeated occurrences of the pattern "AAC."

chosen. In any case, c is not the same as p_{k+1} , a wise decision has to be made to choose the next state. It should not miss any possible match in the sequence.

Whenever a match begins in a region that already matches P , the match begins with any prefix of P ; and, there has to be a matching pattern in S up to k . In order to avoid skipping a match in S , the longest such region is best. It is therefore necessary to identify the longest prefix of P that corresponds to P_k . To be more specific, it will require dropping the fewest characters from P_k 's beginning to get something that looks like a prefix of P again.

This observation is recorded in the shift beyond for avoiding redundant comparison (SBARC) table. The table consists of the shift value for the characters of the pattern. This assists in escaping from the redundant comparison in the occurrences of recurring characters. For computing such a skip, we focus on substrings of patterns that are prefix and suffix. A string must be different from the string itself in order to be a proper prefix. As an example, in the string "ACG," the prefixes are "", "A," "AC," and "ACG." However, the proper prefixes are "", "A," and "AC." In particular, we focus on substrings of patterns that are prefixes and suffixes. Whenever the prefix $Pat[0..i]$ of pattern consists of $i = 0$ to $m - 1$, the shift value for the position i of the pattern, $SV_{Pat}[i]$, stores the length of the maximum matching proper prefix, that is, also the suffix of the subpattern $pat[0..i]$.

The algorithm for the computation of the SBARC table is given in Algorithm 1. The calculation is done as follows: the pattern is scanned from left to right; the shift value for the characters of the pattern is computed based on the repetition of any prefix of length 1 to $l_p - 1$ in Pat . If any such prefix is occurring for the next time, the shift value (SV) for the repetition will be related to the position of the same character in the prefix.

For example, considering the pattern "ACGACG," the prefix "ACG" appears twice. Prefix "A," of length 1, appears at position 4 again (at index 3); thus, index 3 has 1 for its shift value. Continuing the prefix "AC" ending at the fifth position (the index 4), the shift value for index 4 is 2; and prefix "ACG" of length 3 is appearing again till the sixth position (at the index 5), index 5 has 3 as the shift value. The shift value is assigned to each index position of the pattern and not to the characters in the pattern.

```

//Input: Pattern (Pat) and length of the pattern ( $l_p$ )
//Output: Shift Value of the pattern
begin
     $l = 0$ 
     $SV_{Pat} [0] = 0$ 
     $i = 1$ 
    while ( $i < l_p$ )
        if ( $Pat [i] == Pat [1]$ )
             $l + = 1$ 
             $SV_{Pat} [i] = l$ 
             $i + = 1$ 
        else if ( $l = 0$ )
             $l = SV_{Pat} [l - 1]$ 
        else
             $SV_{Pat} [i] = 0$ 
             $i + = 1$ 
    endIf
    endWhile
    return  $SV_{Pat} []$ 
end;

```

ALGORITHM 1: compute_SBARC(Pat, l_p).

The shift value assigned for each position of the pattern is otherwise known as the longest proper prefix, that is, also a suffix. The important point to be observed here is that the prefix is the proper substring starting from the beginning of the pattern, but the suffix is considered for each substring ending at all possible positions of the pattern.

During the comparison of the pattern against the sequence, the search starts from the left end of the sequence. The pattern is aligned with the first l_p characters of the sequence. The character-by-character comparison is performed from left to right. When a mismatch is found after matching of few characters, a few characters of the next window are already known. Using this advantage, we do not have to compare those characters that are known to match anyway. Hence, when there is a mismatch, the shift value of the previous index of the mismatched position in the pattern is referred to. In the pattern, if any prefix is repeated as the suffix till the position of mismatch, the comparison of characters in the prefix need not be repeated with the sequence. Accordingly, shift value for each position of the pattern is assigned and the same is referred for continuing the comparison.

Comparison with the characters of the current window starts with $Pat[j]$ with $j = 0$. Matching characters, $Seq[i]$ and $Pat[j]$, is continued and i and j are incremented, while $Pat[j]$ matches $Seq[i]$. When a mismatch occurs, it is known that the characters $Pat[0..j-1]$ match with $Seq[i-j..i-1]$. Additionally, it is true (from the compute_SBARC algorithm) that $SV_{Pat}[j-1]$ is the count of characters of $Pat[0..j-1]$ that are both proper prefixes and suffixes. As a result of these two points, it can be concluded that $SV_{Pat}[j-1]$ characters do not need to be matched with $Seq[i-j..i-1]$, because they will match anyway.

The core idea is not to go back to the previous characters in the sequence during a mismatch, i.e., the index used for the sequence will always be moving forward or idle, but not

backward for sure. Only the index of the pattern will be getting increased or decreased based on the shift value. Hence, the EPM algorithm, presented in Algorithm 2, achieves a linear time complexity for pattern matching.

For the pattern matching process, the comparison of characters starts at index 0 of both Seq and Pat. The comparison continues towards the right linearly till the match of the characters is successful. Once a mismatch occurs between $Pat[j]$ and $Seq[i]$, the character of the pattern at the shift value of $j-1$, i.e., $SV_{Pat}[j-1]$, is aligned to $Seq[i]$ and the comparison continues.

3.3. Demonstration of the Algorithm with Example.

Consider the example given in Figure 4 for the computation of the shift value of each position of the pattern. The shift value is assigned for each index of the pattern and not to the characters of the pattern. A pattern's index number is indicated by Ind_{pat} , while the shift value is represented by SV_{Pat} .

In the pattern "ACTCTAACTGA," at index 5, 6 and 10 prefix A appear, hence, shift value 1 is assigned; prefix AC of length 2 is ending at index 7, hence, shift value 2 is assigned to index 7; prefix ACT of length 3 is ending at index 8, hence, 3 is assigned to 8; the method given in Algorithm 1 computes SBARC table of the pattern.

The example is demonstrated in Figure 5 elaborates the process of pattern matching in a DNA sequence. Ind_{seq} and Ind_{pat} indicate the index positions of the characters in a sequence and in a pattern, respectively. The characters in the first window of 10 characters in the Seq are compared from left to right with the characters of the pattern. The characters from position 0 to 8 are matched successfully. Since $Pat[9]$ is "G" and $Seq[9]$ is "C," a mismatch has occurred. As a result, it is apparent that the pattern is not present at position 0. A naive approach would shift the pattern into the window of

```

//Input: Seq-Sequence, Pat-Pattern
//ls-length of the Seq
//lp-length of Pattern
//Output: Pos[]-Matched Index Positions
begin
    j = 0, i = 0
    compute_SBARC (Pat, lp)
    while i < ls:
        if Pat [j] == Seq [i]:
            increment i and j by 1
        endif
        if j == lp:
            add i - j to Pos
            j = SVPat [j - 1]
        else if (i < ls and Pat [j] != Seq [i]):
            if (j != 0)
                j = SVPat [j - 1]
            else:
                i += 1
            endif
        endif
    endwhile
    return Pos[]
end;

```

ALGORITHM 2: EPM (Seq, Pat, l_s, l_p).

Ind _{pat}	0	1	2	3	4	5	6	7	8	9	10
Pat	A	C	T	C	T	A	A	C	T	G	A
SV _{pat}	0	0	0	0	0	1	1	2	3	0	1

FIGURE 4: SBARC for the pattern.

index positions 1 to 11, and continue the comparison. The EPM algorithm uses the SBARC table to determine the index of the following window in order to optimize the comparison process. A new starting index is not explicitly assigned to the next window. Rather, it takes the index of the pattern character to be compared next to the 9th character from SVPat. Therefore, the character at the position of SVPat [8] in the pattern is aligned with Seq [9] and the matching process continues. Again, the Pat [4] is mismatched with Seq [10]. The next window is chosen by assigning the character at the position of SVPat [3] in the pattern aligned with Seq [10]. Followed by this, all the characters of the pattern are exactly matched from position 10 in the sequence.

In the sequence of length, the window of the pattern is shifted barely three times in order to identify the pattern. If the naive algorithm had been applied, the shifts would have been $l_s - l_p + 1$ times, which would yield $21 - 11 + 1 = 11$. According to our analysis, the number of shifts has reduced drastically, reflecting the fall in the number of character comparisons in total.

3.4. Algorithm for Optimized Pattern Similarity Identification (OPSI). While applying pattern matching in DNA sequences, there is a chance that the mutated pattern will be present in the sequence instead of the exact pattern. Hence,

Ind _{seq}	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Seq	A	C	T	C	T	A	A	C	T	C	A	C	T	C	T	A	A	C	T	G	A
Ind _{pat}	0	1	2	3	4	5	6	7	8	9	10										
Pat	A	C	T	C	T	A	A	C	T	G	A										
Mismatch at Pat[9]. Align the index of SV _{pat} [8] to Seq[9] (As in fig. 2, SV _{pat} [8] = 3)																					
Ind _{seq}	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Seq	A	C	T	C	T	A	A	C	T	C	A	C	T	C	T	A	A	C	T	G	A
Ind _{pat}				0	1	2	3	4	5	6	7	8	9	10							
Pat				A	C	T	C	T	A	A	C	T	G	A							
Mismatch at Pat[4]. Align the index of SV _{pat} [3] to Seq[10] (As in fig. 2, SV _{pat} [3] = 0)																					
Ind _{seq}	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Seq	A	C	T	C	T	A	A	C	T	C	A	C	T	C	T	A	A	C	T	G	A
Ind _{pat}											0	1	2	3	4	5	6	7	8	9	10
Pat											A	C	T	C	T	A	A	C	T	G	A
Match is found at 10 of seq																					

FIGURE 5: Demonstration of EPM.

pattern similarity also needs to be enhanced to efficiently find out possible positions where the pattern is almost in the DNA sequence.

APM can be made by allowing insertion, deletion, and substitution of characters for a given threshold value. The example in Figure 6 demonstrates the possible mutations in the DNA sequence.

Seq	A A C G T T C G A G C T C G G
Mutated _{seq} (Insertion)	A A C G T T C G A T C A G C T C G G
In the mutated sequence, TCA is additionally inserted into the DNA sequence	
Seq	A A C G T T C G A G C T C G G
Mutated _{seq} (Deletion)	A A C C G A G C T C G G
In the mutated sequence, GTT is removed from the DNA sequence	
Seq	A A C G T T C G A G C T C G G
Mutated _{seq} (Substitution)	A A C G T A C G C G C A C G G
In the mutated sequence, 3 base pairs are replaced by other base pairs	

FIGURE 6: Example for mutation types.

OPSI algorithm identifies pattern similarity by allowing the mutation caused by substitution. During the search of a pattern in the DNA sequence, if the pattern is exactly matched with the current window of the sequence, the portion of the sequence matched with the pattern is not considered for approximate matching. Since the approximation is permitted, the time efficiency of the process depends on the number of mismatches considered. Hence, OPSI algorithm has a time complexity in the order of the product of the length of the text and the threshold for mismatches allowed ($O(l_s, \epsilon)$). If the number of mismatches permitted is least considerable, then OPSI algorithm is having the time complexity in linear order. OPSI has shown an improvement as 69% in the execution time. With increasing thresholds for mismatches and sequence length, there will be an increase in time complexity.

When there is a mismatch between the characters of the sequence and the pattern, the pattern similarity is identified by allowing a specified number of mismatches. This threshold for mismatches is represented as ϵ . The flow of OPSI method is illustrated in Figure 7.

The example in Figure 8 explains the pattern similarity permitting a maximum of three mismatches using OPSI. The EPM algorithm is followed the same way when there is a perfect match between the characters of the sequence and the pattern. When a mismatch occurs, the algorithm counts up the mismatch and proceeds as if there was no mismatch. When the count exceeds the specified acceptable number of mismatches, the algorithm alerts to switch to the shift value of the pattern, based on the index, where the first mismatch occurred in the current window of comparison. Hence, if the exact match is found, the algorithm proceeds, as the EPM algorithm works for the current window of characters. And during mismatch, the index of the text is also moved back to the position of the first mismatch.

OPSI algorithm, given in Algorithm 3, gets sequence (Seq), pattern (Pat), length of the sequence (l_s), length of the pattern (l_p), and the permitted number of mismatches (ϵ) as input and returns the list of positions (Pos), wherever the pattern is found in the sequence with mismatches less than or $\epsilon + 1$. The comparison of characters starts as in the EPM algorithm. The matching process starts from the left end and moves to the right, character by character. When there is a mismatch and the count of mismatches till then is less than ϵ , the count of error is updated and the comparison proceeds

towards the right. If the index of the pattern becomes the length of the pattern, a pattern is found with a permitted number of mismatches and hence it is added to the output array, Pos.

If an exact match is found, the pattern index will update based on the last character of the pattern. If the pattern was found with mismatches or the number of mismatches overtakes ϵ , then the index of the sequence is moved to the position, where the first mismatch occurred in the current alignment, and the pattern index is moved to the shift value of the character behind the position of the mismatch in the pattern.

4. Results and Analysis

4.1. Implementation and Experiment Design. OPSI was implemented in Python 3. Intel (R) Core (TM) i5-1035G1 CPU @ 1.00 GHz 1.19 GHz with 8 GB RAM. For the implemented work to be tested, data was taken from *Homo sapiens* chromosome Y, CM000686.2 in the NCBI library. Figure 8 shows a sample input sequence of 2000 base pairs (bp). In double-stranded nucleic acids, a base pair is formed by joining two nucleobases together with hydrogen bonds. Genes and organisms are described in terms of the number of base pairs contained in their genetic sequences as DNA is usually double-stranded. It is for this purpose that for every strand of DNA with a given sequence, there is an opposite strand containing complementary sequences as A complements T, and G complements C. According to Watson and Crick, the pair of researchers, who revealed DNA's structure, this complementarity was designated the Watson–Crick pairing rule [56]. Based on this, the number of nucleotides in each strand equals the total number of base pairs.

To experimentally demonstrate the output of EPM, the test input in Figure 9 was chosen as the sample sequence. The pattern “TCTGGTCTCTTTCTGTCCTCAATGAGACCT” of length 30 bp is given to be searched in the sequence. The pattern gets matched exactly starting from index 719 in the given sequence. The output obtained is given in Figure 8. The same input is then used for verifying the OPSI. The identification of exact pattern matching is done through the OPSI algorithm. If the same matching in EPM is identified by OPSI, it shows that the matching is not missed by the OPSI algorithm.

OPSI algorithm is also executed for searching the same pattern “TCTGGTCTCTTTCTGTCCTCAATGA GACCT” of length 30 bp in the sequence in Figure 10, the tolerable number of mismatches is assigned as 5. The OPSI algorithm is applied to the same inputs as a check. If the same inputs are fed to the OPSI algorithm, it is expected that this exact matching position, as well as approximate matched positions, are to be located. All the approximate matching positions with 0 to 5 mismatches are expected to be identified. The output is shown in Figure 10. As a result, the exact matching position, and the starting indices of the pattern in the sequence, wherever it is approximately matched with a maximum of 5 mismatches, are listed out. The exact matching position 719 as given by the EPM algorithm is listed among the outputs received from OPSI algorithm and the same is highlighted in Figure 11.

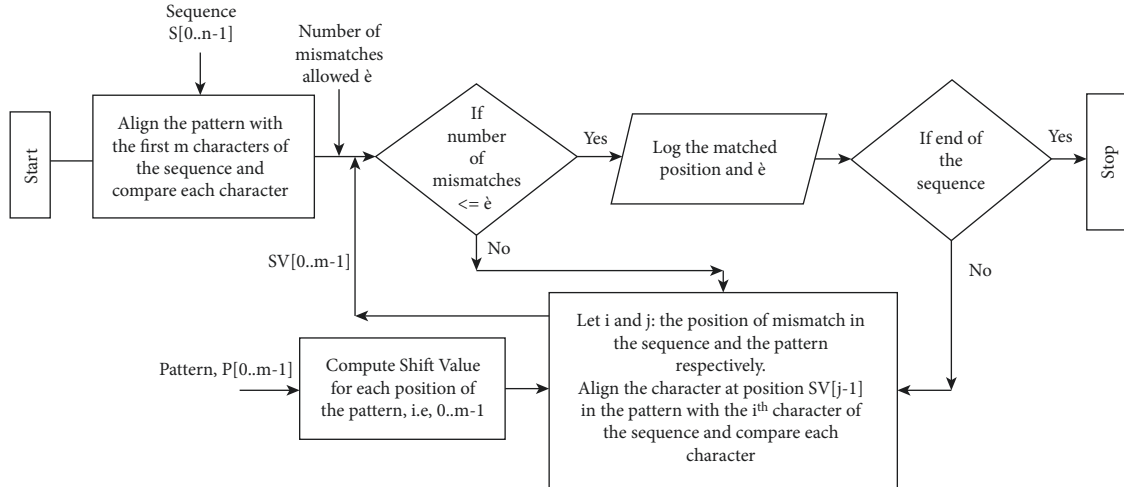


FIGURE 7: Flow diagram of OPSI.

Ind _{Pat}	0	1	2	3	4	5	6
Pat	A	C	G	A	C	G	A
SV _{Pat}	0	0	0	1	2	3	4

Ind _{seq}	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Seq	A	C	G	A	C	G	A	T	G	A	A	C	G
Ind _{pat}	0	1	2	3	4	5	6										
Pat	A	C	G	A	C	G	A										
Match found at 0 with 0 mismatch; Align Pat[SV _{pat} [6] with Seq[7] to find next match																	
Ind _{seq}	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Seq	A	C	G	A	C	G	A	T	G	A	A	C	G
Ind _{pat}				0	1	2	3	4	5	6							
Pat				A	C	G	A	C	G	A							
Match found at 3 with 1 mismatch; Align Pat[SV _{pat} [3] with Seq[7] to find next match																	
Ind _{seq}	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Seq	A	C	G	A	C	G	A	T	G	A	A	C	G
Ind _{pat}							0	1	2	3	4	5	6				
Pat							A	C	G	A	C	G	A				
4 mismatches, Align Pat[0] with Seq[7] to find next match																	

(a)

(b)

FIGURE 8: (a) SBARC table. (b) Finding pattern similarity.

4.2. Analysis of OPSI. In the compute_SBARC() algorithm, the pattern string is traversed at most once through the while loop. The variable i controls the number of iterations of the while loop. The variable l is used for tracking the length of the SV_{Pat} for the previous index. The variables l and SV_{Pat}[0] are initialized to 0. If Pat[l] matches Pat[i], l is incremented by 1 and SV_{Pat}[i] is assigned the incremented value. Whenever Pat[i] does not match to Pat[l] and l is not 0, l is updated to SV_{Pat}[l-1]. This process assures that the pattern is scanned once to its length. Hence the time complexity of the algorithm is $O(l_p)$.

There is only one while loop used in the EPM algorithm. The index variable, “ i ,” that controls the loop is initially set to 0, and it is incremented until “ i ” is equal to the length of the sequence. While this indicates that the while loop is executed l_s times, it does not necessarily mean that it is executed exactly l_s times. This is because “ i ”

is not incremented constantly throughout the while loop. There are few conditions in which “ i ” stays idle. In the event that the i^{th} character of sequence matches the j^{th} character of the pattern, “ i ” is incremented. If not, “ j ” is set to the SV_{Pat} of $j-1$ and the loop continues without an update to the index “ i .” This is possible to a maximum of $2 * l_s$ number of times. This means that the EPM algorithm has $O(l_s)$ time complexity.

Together, the compute_SBARC and EPM algorithm takes $O(l_s + l_p)$ time. Due to the shorter length of the pattern than the length of the sequence, the time complexity is bounded by l_s . This leads to the conclusion that the finding exact matching has the time complexity $O(l_s)$.

OPSI algorithm behaves in a similar way when there are exact matches found in the sequence. When mismatches occur, the behavior of OPSI differs. The input scenario for the OPSI algorithm can be categorized into three cases.

```

//Input: Seq-Sequence, Pat-Pattern,  $l_s$ -length of the Seq
//Input:  $l_p$ -length of Pattern,  $\hat{e}$ -Number of permitted mismatches
//Output: Pos[]-Starting indices of the Pat in Seq with less than  $\hat{e} + 1$  mismatches
begin
     $j = 0, i = 0$ 
    compute_SBARC (Pat,  $l_p$ )
    while ( $i < l_s$ )
        if (pat [ $j$ ] == txt [ $i$ ])
            increment  $i$  and  $j$ 
        else
            increment the count of error
        if count of error  $> \hat{e}$ 
            reinitialize  $i$  and  $j$  based on the shift value calculated
            goto line 5
        endif
        if ( $j = l_p$ )
            add  $i - j$  to Pos
            if count of errors == 0
                assign shift value of  $j - 1$  to  $j$ 
            else
                reinitialize count of errors to 0
                shift  $i$  to the position where the mismatch occurred first in the current window
                shift  $j$  to the shift value of the previous position where the mismatch occurred
            endif
        endif
    endwhile
    return Pos []
end;

```

ALGORITHM 3: OPSI (Seq, Pat, l_s , l_p , \hat{e}).

```

TGTCTCAATGAGACCTAGGGCCAGTGCAGACTCTAAAGTTGCATAGTCTGCTCTCTATCAGTCCTCAG
TGAGACGTAGACCTAATGTAGACTCTAAAGTTTGCAAAGTCTGCTCTCTATCTCTCCTCAGTGAGACCTA
GACCAATGCAGACTCTAAAGTTGCACAGTATGGTCTCTATCTGTCCTCAATGAGACCTAGGCCAATGCAGACTTA
CAGACTCTAAAGTTTACAGTCTGCTCTCTATCTGTCCTCAATGAGACCTAGGCCAATGCAGACTCTA
AACGTTGCACAGTCTGCCCTCTATCTGTCCTCAATGAGACCTAGGCCAATGCAGACTCTAAAGTTTGA
CAGTCTGGTCTCTATCTGTCCTCAATGAGACCTAGGCCAATGCCGACTCTAGAGGTTGCACAGTGTGCT
CTCTATCTGCTCTCAATGAGACCTAGGCCAATGCAGACTCTAAAGTTGCACAGTCTGCTCTCTAACTG
CCCTCAATGAGACCTAGGCCAATGCAGACTCTAAAGTTTGCACAGTCTGTTCTCTATCTGTCCTCAATG
AGACCTAGGCCAAGTGTAGACTCTAAAGCTTGCACAGTCTGCTCTCTATCTGACCTCAATGAGACCTAGG
CCCAATGCAGACTATAAAGTTTACAGTCTGCTCTCTATCTGTCCTCAATGAGACCTAGGCCAATGCA
AACTCTAAAGTTGCACACTCTGGTCTCTTTCTGTCCTCAATGAGACCTAGGCCAATGCAGACTCTAAA
GGTTGCACAGTCTGCTCTCTAACTGTCCTCAATGAGACATAGGCCAATGCAGACTCTAAAGTTGCACA
GTCTGCTCTCTATGTGTCCTCAATGAGACCTAGGCCAAGTGCAGACTCTCAAGGTTGCATAGTATGCTCT
CTATCTGTCCTCAATAAGATCTAGGCCAATGCAGACTCTAAAGGTTGCCGAGCCTGCTCTCTATCTGCC
CTCAATGAGACCTAGGCCAATGCAGACTCTAAAGTTGCACAGTCCGCTCCCTATCTGTCCTCAATGAG
ATCTAGGCCAATAGAGACTCTAAAGTTGCACAGTCTGCTCTCTATCTGTCCTCAATGAGACCGAGGCT
CTATGCAGACTCTAAAGTTACACAGTGTGCTCTCTATCTGTCCTCAGTGAGACCTAGGCCAATGCAGA
CTCTAAAGTTTGCAGTCTGCTCTCTATCTGTCCTCAATGAGACCTAGGGCCAGTGCAGACTCTAAAGG
TTGCATAGTCTGCTCTATCAGTCTCAGTGCAGACTAGACCAATGGAGACTCTAAAGTTTGCAAAGTCT
TGCTCTCTATCTCGCCTCAGTGCAGACTAGACCAATGCAGACTCTAAAGTTTGCACAGTCTGCTCTA
TCTGCCCTCAATGAGACCTAGGCTCAGTGCAGACTTAAAGTTTGCACAGTCTGCTCCGTATCTGTCCTC
AATGAGACCTAGGTCCATTGCAGACTCTAAAGTTGCACAGTCTGCCCTCTATCTGTCCTCAATGAGACC
TAGGCCAATGCAGCCTCTAAAGTTTCAACAGTCTGGTCTCTATCTGTCCTCAATGAGACCTAGGCCAA
TGCCGACTCTAGAGTTGCACAGTGTGCTCTCTATCTGCTCTCAATGAGACCTAGGCCAATGCAGACTC
TAAAGTTTGCACAGTCTGCTCTCTAACTGCCCTCAATGAGACCTAAGCCCAATGCAGACTCTAAAGTTG
TACAGTCTGGTCGCTATCTGTCCTCAATGAGACCCAGGCCAATGCAGACTCTAAAGTTTGCACAGTCTG
CTCTATCTGTCCTCAATGAGACCTAGGACCAGTGCAGACTCTAATGGTTGCCAGTGTGCTCTCTATC
TGTCTCAATGAGACCTAGGCCAATGCAGACTCTAAAGTTTGCACAGTCCGGTCTCTATCTGTCCTCAA
TGAGACCTAGGCCAATGCCGACTCTAAAGTTTGCACAGT

```

FIGURE 9: Sample sequence of length 2000 bp.

```

Run - ApproxKMP.py
Run: KMP x
C:\Users\...anaconda3\python.exe "C:/.../KMP.py"
Found pattern at index 719
Time elapsed: 0.0015995502471923828
Length of the Pattern: 30
Length of the Sequence: 2000

Process finished with exit code 0
    
```

FIGURE 10: EPM search of “TCTGGTCTCTTTCTGTCCTCAATGA GACCT” in the sequence in Figure 7.

```

Run - ApproxKMP.py
Run: ApproxKMP x
C:\Users\...anaconda3\python.exe "C:/.../ApproxKMP.py"
Found pattern at index 48 with 5 mismatches
Found pattern at index 109 with 4 mismatches
Found pattern at index 170 with 3 mismatches
Found pattern at index 231 with 2 mismatches
Found pattern at index 292 with 3 mismatches
Found pattern at index 353 with 1 mismatches
Found pattern at index 414 with 5 mismatches
Found pattern at index 475 with 4 mismatches
Found pattern at index 536 with 2 mismatches
Found pattern at index 597 with 3 mismatches
Found pattern at index 658 with 2 mismatches
Found pattern at index 719 with 0 mismatches
Found pattern at index 780 with 4 mismatches
Found pattern at index 841 with 3 mismatches
Found pattern at index 902 with 5 mismatches
Found pattern at index 963 with 4 mismatches
Found pattern at index 1024 with 5 mismatches
Found pattern at index 1085 with 3 mismatches
Found pattern at index 1146 with 4 mismatches
Found pattern at index 1207 with 2 mismatches
Found pattern at index 1328 with 5 mismatches
Found pattern at index 1389 with 2 mismatches
Found pattern at index 1450 with 4 mismatches
Found pattern at index 1511 with 3 mismatches
Found pattern at index 1572 with 1 mismatches
Found pattern at index 1633 with 5 mismatches
Found pattern at index 1694 with 4 mismatches
Found pattern at index 1755 with 3 mismatches
Found pattern at index 1816 with 3 mismatches
Found pattern at index 1877 with 3 mismatches
Found pattern at index 1938 with 2 mismatches
Time elapsed: 0.008018732070922852
Length of the Pattern: 30
Length of the Sequence: 2000
    
```

FIGURE 11: OPSI search of “TCTGGTCTCTTTCTGTCCTCAATGA GACCT” in the sequence in Figure 7.

Case 1. All exact matches of the pattern in the text: For every window alignment of pattern with the substring of text, if there are always exact matches found in the text, OPSI proceeds as same as the EPM algorithm. Hence, the time complexity will be linear for this case.

Case 2. Lesser number of mismatches than ϵ : If the number of mismatched characters is less than the permitted count of mismatches, then, the next window of search is decided based on the shift value of the position where the first mismatch was found.

Case 3. Number of mismatches exceeding ϵ : When the number of mismatches exceeds the count, then the search will be continued from the shift value of the first mismatched character in the present window of search.

Considering Cases 2 and 3, the position of the mismatch is resumed once the present window is processed. The shift value is used to select the next window for the matching process. Since the shift value assures that the compared pair of characters will not be compared again, the time complexity for the average case will be still linear. The worst case will occur when the number of mismatches in each comparison window exceeds ϵ . In such a case, once the mismatch count

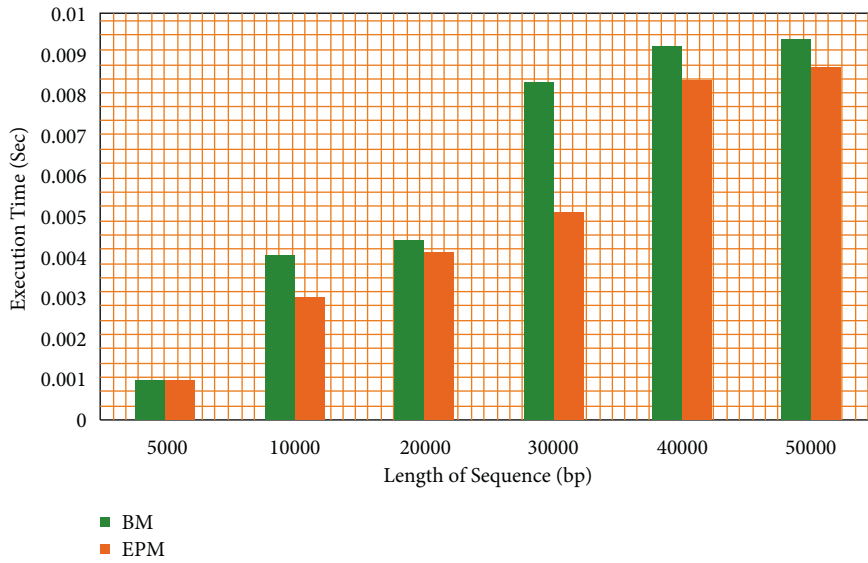


FIGURE 12: Execution time of preprocessing in Boyer–Moore vs compute_SBARC.

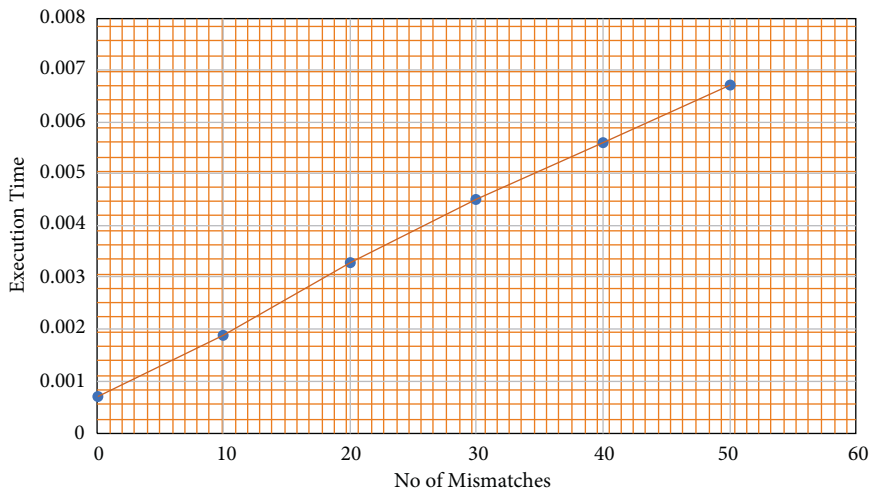


FIGURE 13: Execution time of OPSI vs number of mismatches.

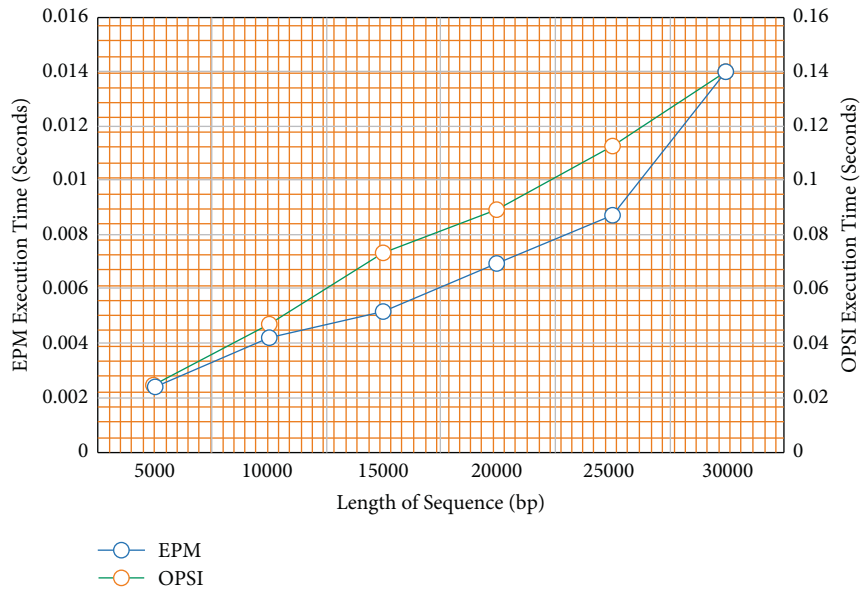
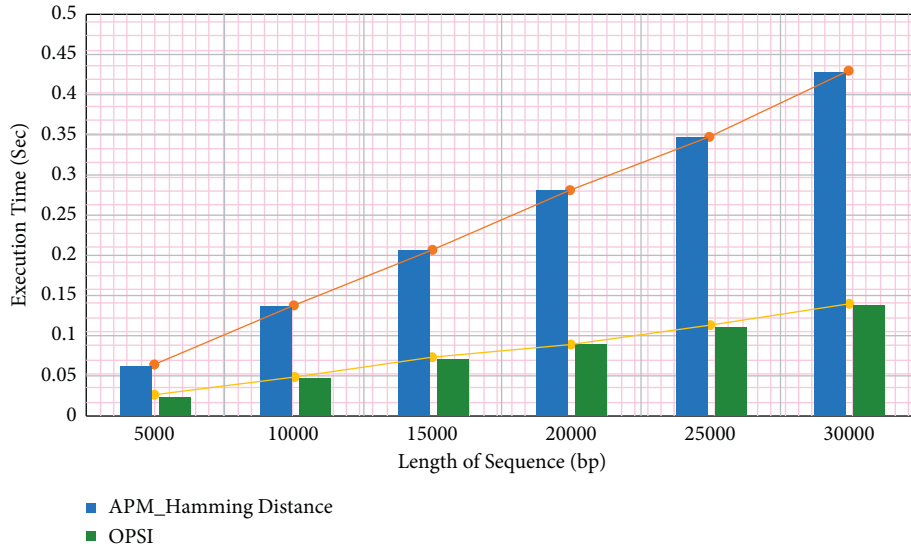
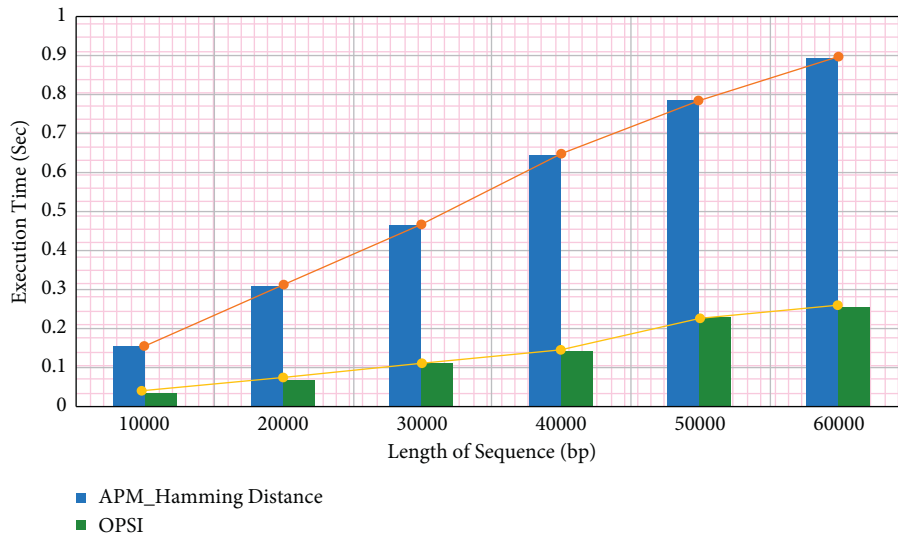


FIGURE 14: EPM vs OPSI.



(a)



(b)

FIGURE 15: Hamming_Distance Based APM vs OPSI. (a) For DNA sequence. (b) For protein sequence.

exceeds ϵ , further characters in the present window will not be compared; instead, the comparison window is immediately shifted. This leads to conclude that the time complexity of OPSI will be $O(l, \epsilon)$ in the worst case.

4.3. Experimental Results. The performance of the OPSI algorithm was experimented with and analyzed from three perspectives such as, (i) the preprocessing of the pattern is compared with the preprocessing of Boyer–Moore for calculating the shift value, (ii) correlate with the EPM to prove the time complexity of OPSI, and (iii) comparison with the APM based on the hamming (edit) distance.

According to the bad character heuristic, the Boyer–Moore algorithm calculates the shift value during its preprocessing. Based on the presence of a prefix as a suffix, the compute_SBARC algorithm calculates the shift value.

Both the algorithms were implemented in Python 3 and tested with patterns of size 5000 to 50000 base pairs. As shown in Figure 12, a graph is generated with data collected during the execution of both methods. Based on the graph, it can be concluded that the difference in execution time is negligible and that both algorithms perform in similar time frames.

OPSI algorithm was applied on a DNA sequence of length 6000 bp and a pattern of length 200 bp. The values for ϵ are varied from 0 to 50 in an interval of 10. The algorithm for OPSI is applied with each of these ϵ values repeatedly for five times and the average execution time is recorded. Figure 13 shows that the execution time increases in the linear order of the growth of the number of mismatches.

OPSI and EPM algorithms are applied on DNA sequences of various lengths and the average execution times are recorded. The value of ϵ is set to 10 and a pattern of

length 200 is chosen for searching. The execution time is portrayed in Figure 14 for both the EPM algorithm and the OPSI algorithm. The execution time is taken as the average time of the multiple executions of the algorithms. It is observed that the execution time of Algorithm 3 is approximately 10 (\approx) times the execution time of the EPM algorithm. Hence, supports the statement that the time complexity of OPSI is $O(l_s * \epsilon)$.

The execution time of the OPSI algorithm is compared with the APM algorithm based on hamming distance [23]. The OPSI algorithm is not limited to DNA sequences, as it can be used with any alphabet. This has been demonstrated by applying it to protein sequences as well. The data for DNA and Protein sequences are taken from <https://pizzachili.dcc.uchile.cl>. A pattern of 200 characters is searched using the algorithms. The average execution time is observed for different lengths of the sequences for both algorithms. The graphs are given in Figures 15(a) and 15(b) report the time taken by OPSI and hamming distance-based APM for searching the pattern in increasing lengths of sequences. According to the difference between the execution times of OPSI and the other algorithm, OPSI algorithm is more efficient.

The improvement in the performance of OPSI is analyzed by comparing the slopes of the lines shown in Figure 15. Considering that the lines are not straight, slopes between every two points are measured. The average of these slopes is calculated and applied for computing the percentage of improvement in the performance of OPSI for APM. It is found to be 68.96% in the case of DNA data set and 70% for Protein sequence. On average, the execution time was improved by 69.48%.

5. Conclusion

Pattern matching is an important and frequent function in any field where a large amount of data is handled. Many algorithms are being proposed by researchers for improving the performance of the search. OPSI finds the positions of the pattern in the text by allowing a specified number of mismatches.

Since the approximation is permitted, the time efficiency of the process depends on the number of mismatches considered. Hence, the OPSI algorithm has a time complexity in the order of the product of the length of the text and the threshold for mismatches allowed ($O(l_s * \epsilon)$). If the number of mismatches permitted is least considerable, then the OPSI algorithm is having the time complexity in linear order.

As compared to the APM based on hamming distance, OPSI has shown an improvement of 69% in the execution time. OPSI possesses scalability because there are no limits on the size of the inputs such as the threshold for mismatches, sequence length, and pattern length. With increasing thresholds for mismatches and sequence length, there will be an increase in time complexity.

In addition, because OPSI is generalizable, it can be applied not just to DNA sequences but also to any string-

matching problems. This algorithm calculates the shift position based on the prefix. In future work, this work can be improved by integrating suffix-based methods such as the BM algorithm.

Data Availability

The datasets used and/or analyzed during the current study are available from the corresponding author on reasonable request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References

- [1] M. Rashid, M. Imran, and A. R. Jafri, "Exploration of hardware architectures for string matching algorithms in network intrusion detection systems," in *Proceedings of the ACM International Conference Proceeding Series*, Bangkok Thailand, July 2020.
- [2] S. Doan, E. W. Yang, S. S. Tilak, P. W. Li, D. S. Zisook, and M. Torii, "Extracting health-related causality from twitter messages using natural language processing," *BMC Medical Informatics and Decision Making*, vol. 19, no. S3, p. 79, 2019.
- [3] D. E. Knuth, J. H. Morris Jr, V. R. Pratt, and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.
- [4] K. V. Voelkerding, S. Dames, and J. D. Durtschi, "Next generation sequencing for clinical diagnostics-principles and application to targeted resequencing for hypertrophic cardiomyopathy: a Paper from the 2009 William Beaumont Hospital Symposium on Molecular Pathology," *Journal of Molecular Diagnostics*, vol. 12, no. 5, pp. 539–551, 2010.
- [5] T. Mantere, S. Kersten, and A. Hoischen, "Long-read sequencing emerging in medical genetics," *Frontiers in Genetics*, vol. 10, pp. 426–514, 2019.
- [6] H. P. J. Buermans and J. T. den Dunnen, "Next generation sequencing technology: advances and applications," *Biochimica et Biophysica Acta - Molecular Basis of Disease*, vol. 1842, no. 10, pp. 1932–1941, 2014.
- [7] S. Behjati and P. S. Tarpey, "What is next generation sequencing?" *Archives of disease in childhood - Education & practice edition*, vol. 98, no. 6, pp. 236–238, 2013.
- [8] X. Huang, J. Wang, S. Aluru, S. P. Yang, and L. Hillier, "Pcap: a whole-genome assembly program," *Genome Research*, vol. 13, no. 9, pp. 2164–2170, 2003.
- [9] M. M. W. De la Bastide, "Assembling genomic DNA sequences with PHRAP," *Bioinformatics*, vol. 11, pp. 1–15, 2007.
- [10] Z. Zhang, W. R. Pearson, and W. Miller, "Aligning a DNA sequence with a protein sequence," *Journal of Computational Biology*, vol. 4, no. 3, pp. 339–349, 1997.
- [11] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller, "A greedy algorithm for aligning DNA sequences," *Journal of Computational Biology*, vol. 7, no. 1–2, pp. 203–214, 2000.
- [12] H. Li, "Aligning Sequence Reads, Clone Sequences and Assembly Contigs with BWA-MEM," pp. 1–3, 2013, <https://arxiv.org/abs/1303.3997>.
- [13] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with Bowtie 2," *Nature Methods*, vol. 9, no. 4, pp. 357–359, 2012.

- [14] K. Reinert, B. Langmead, D. Weese, and D. J. Evers, "Alignment of next-generation sequencing reads," *Annual Review of Genomics and Human Genetics*, vol. 16, no. 1, pp. 133–151, 2015.
- [15] G.-F. Richard, A. Kerrest, and B. Dujon, "Comparative genomics and molecular dynamics of DNA repeats in eukaryotes," *Microbiology and Molecular Biology Reviews*, vol. 72, no. 4, pp. 686–727, 2008.
- [16] Y. Yan, N. Chaturvedi, and R. Appuswamy, "Accel-Align: a fast sequence mapper and aligner based on the seed-embed-extend method," *BMC Bioinformatics*, vol. 22, no. 1, pp. 257–320, 2021.
- [17] H. Hyrö, K. Narisawa, and S. Inenaga, "Dynamic edit distance table under a general weighted cost function," *Journal of Discrete Algorithms*, vol. 34, pp. 2–17, 2015.
- [18] S. Marco-Sola, J. C. Moure, M. Moreto, and A. Espinosa, "Fast gap-affine pairwise alignment using the wavefront algorithm," *Bioinformatics*, vol. 37, no. 4, pp. 456–463, 2021.
- [19] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.
- [20] B. Langmead, "Aligning short sequencing reads with Bowtie," *Current Protocols in Bioinformatics*, 24 pages, Johns Hopkins University, Maryland MD USA, 2010.
- [21] D. Kim, G. Pertea, C. Trapnell, H. Pimentel, R. Kelley, and S. L. Salzberg, "TopHat2: accurate alignment of transcriptomes in the presence of insertions, deletions and gene fusions," *Genome Biology*, vol. 14, no. 4, p. R36, 2013.
- [22] C. Camacho, G. Coulouris, V. Avagyan et al., "BLAST+: architecture and applications," *BMC Bioinformatics*, vol. 10, pp. 421–429, 2009.
- [23] P. Zhang and M. J. Atallah, "On approximate pattern matching with thresholds," *Information Processing Letters*, vol. 123, pp. 21–26, 2017.
- [24] E. Giaquinta, S. Grabowski, and K. Fredriksson, "Approximate pattern matching with k-mismatches in packed text," *Information Processing Letters*, vol. 113, no. 19–21, pp. 693–697, 2013.
- [25] M. Isa Irawan, I. Mukhlash, A. Rizky, and A. Ririsati Dewi, "Application of Needleman-Wunch Algorithm to identify mutation in DNA sequences of Corona virus," *Journal of Physics: Conference Series*, vol. 1218, no. 1, Article ID 012031, 2019.
- [26] L. Salmela and J. Schröder, "Correcting errors in short reads by multiple alignments," *Bioinformatics*, vol. 27, no. 11, pp. 1455–1461, 2011.
- [27] W. C. Kao, A. H. Chan, and Y. S. Song, "ECHO: a reference-free short-read error correction algorithm," *Genome Research*, vol. 21, no. 7, pp. 1181–1192, 2011.
- [28] KMP, "KMP Algorithm for Pattern Recognition," 2022, <https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/>.
- [29] C. R. Rathish and A. Rajaram, "Efficient path reassessment based on node probability in wireless sensor network," *International Journal of Control Theory and Applications*, vol. 34, pp. 817–832, 2016.
- [30] S. Rahamat Basha, C. Sharma, F. Sayeed et al., "Implementation of reliability antecedent forwarding technique using straddling path recovery in manet," *Wireless Communications and Mobile Computing*, vol. 20229 pages, 2022.
- [31] C. R. Rathish and A. Rajaram, "Hierarchical load balanced routing protocol for wireless sensor networks," *International Journal of Applied Engineering Research*, vol. 10, no. 7, pp. 16521–16534, 2015.
- [32] S. Kannan and A. Rajaram, "Enhanced stable path routing approach for improving packet delivery in MANET," *Journal of Computational and Theoretical Nanoscience*, vol. 14, no. 9, pp. 4545–4552, 2017.
- [33] R. P. P. Anand and A. Rajaram, "Effective timer count scheduling with spectator routing using stifle restriction algorithm in manet," *IOP Conference Series: Materials Science and Engineering*, vol. 994, no. 1, Article ID 012031, 2020.
- [34] M. Dinesh, C. Arvind, S. Sreeja Mole et al., "An energy efficient architecture for furnace monitor and control in foundry based on industry 4.0 using IoT," *Scientific Programming*, vol. 2022, Article ID 1128717, 8 pages, 2022.
- [35] K. Mahalakshmi, K. Kousalya, H. Shekhar et al., "Public auditing scheme for integrity verification in distributed cloud storage system," *Scientific Programming*, vol. 2021, Article ID 8533995, 5 pages, 2021.
- [36] J. Divakaran, S. Malipatil, T. Zaid et al., "Technical Study on 5G Using Soft Computing Methods," *Scientific Programming*, vol. 2022, Article ID 1570604, 7 pages, 2022.
- [37] B. Gobinathan, M. A. Mukunthan, S. Surendran et al., "A Novel Method to Solve Real Time Security Issues in Software Industry Using Advanced Cryptographic Techniques," *Scientific Programming*, vol. 2021, Article ID 3611182, 9 pages, 2021.
- [38] S. Shitharth, P. Meshram, P. R. Kshirsagar, H. Manoharan, V. Tirth, and V. P. Sundramurthy, "Impact of big data analysis on nanosensors for applied sciences using neural networks," *Journal of Nanomaterials*, vol. 2021, Article ID 4927607, 9 pages, 2021.
- [39] S. I. Hakak, A. Kamsin, P. Shivakumara, G. A. Gilkar, W. Z. Khan, and M. Imran, "Exact string matching algorithms: survey, issues, and future research directions," *IEEE Access*, vol. 7, pp. 69614–69637, 2019.
- [40] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.
- [41] F. Zhao and Q. Liu, "A string matching algorithm based on efficient hash function," in *Proceedings of 2009 International Conference on Information Engineering and Computer Science ICIECS*, December 2009.
- [42] T. Gururaj and G. M. Siddesh, "Hybrid approach for enhancing performance of genomic data for stream matching," *International Journal of Cognitive Informatics and Natural Intelligence*, vol. 15, no. 4, pp. 1–18, 2021.
- [43] R. Chikhi and G. Rizk, "Space-efficient and exact de Bruijn graph representation based on a Bloom filter," *Algorithms for Molecular Biology*, vol. 8, no. 1, pp. 22–29, 2013.
- [44] M. Najam, R. U. Rasool, H. F. Ahmad, U. Ashraf, and A. W. Malik, "Pattern Matching for DNA Sequencing Data Using Multiple Bloom Filters," *BioMed Research International*, vol. 20199 pages, 2019.
- [45] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.
- [46] A. Daptardar and D. Shapira, "Adapting the Knuth-Morris-Pratt algorithm for pattern matching in Huffman encoded texts," in *Proceedings of the Data Compression Conference Proceedings*, vol. 535, March 2004.
- [47] R. Grossi and G. Italiano, "Suffix trees and their applications in string algorithms," in *Proceedings of the South American Workshop on String Processing*, vol. 20244, pp. 57–76, South America, February 1997.
- [48] A. N. F. Arruggia, T. R. G. Agie, and G. O. N. Avarro, "Relative Suffix Trees," *The Computer Journal*, vol. 61, no. 5, 2018.

- [49] J. C. Na, H. Park, M. Crochemore et al., "Suffix tree of alignment: an efficient index for similar data," *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer, Berlin, Heidelberg, pp. 337–348, 2013.
- [50] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *Journal of Discrete Algorithms*, vol. 2, no. 1, pp. 53–86, 2004.
- [51] A. M. S. Shrestha, M. C. Frith, and P. Horton, "A bioinformatician's guide to the forefront of suffix array construction algorithms," *Briefings in Bioinformatics*, vol. 15, no. 2, pp. 138–154, 2014.
- [52] C. H. Teo and S. V. N. Vishwanathan, "Fast and space efficient string kernels using suffix arrays," in *Proceedings of the 23rd international conference on Machine learning - ICML*, pp. 929–936, Pennsylvania, PA, USA, June 2006.
- [53] G. Özcan and O. S. Ünsal, "Fast bitwise pattern-matching algorithm for DNA sequences on modern hardware," *Turkish Journal of Electrical Engineering and Computer Sciences*, vol. 23, no. 5, pp. 1405–1417, 2015.
- [54] P. Ejendibia and B. Baridam, "String searching with DFA-based algorithm," *International Journal of Applied Information Systems*, vol. 9, no. 8, pp. 1–6, 2015.
- [55] J. Kaur, B. Chauhan, and J. K. Korepal, "Implementation of query processor using automata and natural language processing," *International Journal of Scientific and Research Publications*, vol. 3, no. 5, pp. 1–5, 2013.
- [56] L. Pray, "Discovery of DNA Double Helix: Watson and Crick | Learn Science at Scitable," 2008, <https://www.nature.com/scitable/topicpage/discovery-of-dna-structure-and-function-watson-397/>.