Noname manuscript No.
(will be inserted by the editor)

Apache Spark Implementations for String Patterns in DNA Sequences

Andreas Kanavos · Ioannis Livieris · Phivos Mylonas · Spyros Sioutas · Gerasimos Vonitsanos

the date of receipt and acceptance should be inserted later

Abstract The availability of numerical data grows from one day to another in a remarkable way. New technologies of high-throughput Next Generation Sequencing (NGS) are producing DNA sequences. Next Generation Sequencing describes a DNA sequencing technology which has revolutionised genomic research. In this paper, we perform some experiments using a cloud infrastructure framework, namely Apache Spark, in some sequences derived from the National Center for Biotechnology Information (NCBI). The problems we examine are some of the most popular ones, namely, Longest Common Prefix, Longest Common Substring, and Longest Common Subsequence.

Keywords DNA Sequencing · Longest Common Prefix (LCP) · Longest Common Substring · Longest Common Subsequence (LCS)

1 Introduction

Current scientific advancements in both biological as well as computer sciences have brought new opportunities to intra-disciplinary research topics. On the one hand, the advancement in molecular biological experiments is producing huge amounts of data related to genome and RNA sequences, protein and metabolite abundance, protein-protein interactions, gene expression, and so on. On the other hand, computers and big-data analytics along with cloud software tools are

A. Kanavos and S. Sioutas and G. Vonitsanos
Computer Engineering and Informatics Department
University of Patras, Patras 26504, Greece
E-mail: {kanavos, sioutas, mvonitsanos}@ceid.upatras.gr

I.E. Livieris
Department of Mathematics
University of Patras, Patras 26504, Greece
E-mail: livieris@gmail.com

P. Mylonas
Department of Informatics
Ionian University, Corfu 49100, Greece
E-mail: fmylonas@ionio.gr

being developed and thus, the capability of processing from terabyte data sets to petabytes and beyond has been rapidly increased. As a result, the development of computer science methods and models used to describe these problems in a formal way arised in bioinformatics. The available algorithmic approaches used to solve them are of great interest among researchers.

Regarding the majority of these problems, biological data are forming big, versatile and complex networks. More to the point, in recent years, sequencing has faced major scientific progress and in following has leveraged the development of novel bioinformatic applications. It is not surprising why bioinformatics and life sciences applications, in general, are facing a rapidly increasing demand for data-handling capacity. In many cases, from low-level applications (such as systems biology) to high-level integrated applications (such as systems medicine), the amounts of data to be stored, transferred, and finally processed, meet congestion in many current technologies.

The advances in the fields of bioinformatics and systems biology furtherly require improved computational methods for analyzing data, while the ongoing progress in the field of molecular biology is evident and thus influences the development of computer science methods. Authors in [8] introduce some key problems in bioinformatics, in following discuss the models used to formally describe these problems, and finally analyze the algorithmic approaches used to solve them.

A Deoxyribonucleic Acid (DNA) macromolecule can be coded by a sequence over a four-letter alphabet. The four letters are A, C, G, and T, and code the bases Adenine, Cytosine, Guanine, and Thymine, respectively. More specifically, DNA Sequencing consists in determining the exact order of these bases in a DNA macromolecule. As a result, DNA sequencing technology constitutes a vital role in the advancement of molecular biology. Compared to previous sequencing aspects, Next-Generation Sequencing (NGS) perform swifter, with significantly lower production costs and much higher throughput in the form of short reads, i.e., short sequences coding portions of DNA macromolecules.

The remainder of the paper is organized as follows. Section 2 summarizes the history of NGS and provides an overview of the three corresponding problems. Section 3 discusses Longest Common Prefix along with four different algorithms. Likewise, Sections 4 and 5 present Longest Common Substring and Longest Common Subsequence respectively. In addition, Sections 6 and 7 introduce the details of the implementation of the system (and the respective cloud infrastructure utilized) as well as the experimental results. Finally, in Section 8 our concluding remarks, open problems, and future work are introduced.

2 Related Work

In our days, there is a need for more effective algorithms regarding DNA sequence processing as the acquisition of DNA information is no longer a bottleneck in genetics. This processing includes searching for selected parts of a sequence, analysis of similarities, differences, or even repetitive fragments. Therefore, sequence alignment methods are gaining much attention in terms of the research in biology and medicine. The sequences are, in many cases, expected to be matched despite the existing minute differences, as they may be caused by acquisition errors; hence

the searching procedure should accept a controlled number of mismatches. Heuristic methods, implemented in Basic Local Alignment Search Tool (BLAST)[1], a program commonly used for sequence alignment, can manage searching sequence fragments in large databases.

Researchers in the field of computer science consider the fact that biologically meaningful results could come from considering DNA as a one-dimensional character string, abstracting the reality of DNA as a flexible three-dimensional molecule. Thus, interaction in a dynamic environment with protein and RNA, and repetition of a life-cycle in which even the classic linear chromosome exists for only a fraction of time can take place [17]. Significant contributions to computational biology might be made by extending or adapting algorithms from computer science, even when the original algorithm has no clear utility in biology. This is illustrated by several recent sublinear-time approximate matching methods for database searching that rely on an interplay between exact matching methods from computer science and dynamic programming methods already utilized in molecular biology. Certain string algorithms that were generally deemed to be irrelevant to biology just a few years ago have become adopted by practicing biologists in both large-scale projects and in narrower technical problems.

A number of data structures have been designed with the aim of storing these impressive amounts of data in an efficient way while allowing for immediate indexing and searching. As a result, all occurrences of any given pattern can be found without traversing the whole sequences. In short, indexing is profitable and useful if utilized regularly. It is evident that if sequences are available beforehand and not periodical alteration takes place, researchers and users can enjoy full advantage of the index. The primary goal of these corresponding data structures constitutes the construction of an index that provides efficient answers to queries with reasonable building and maintenance costs. Classical data structures such as Tries [9], Suffix Trees [12, 36, 38], Suffix Arrays [30], Directed Acyclic Word Graphs (DAWG) [7] as well as the compact version CDAWG [11] are arguably considered very popular and useful data structures for string analysis, especially when searching over large sequence collections. Yet, these structures are full-text indexes as they require a large amount of space for a sequence to be represented. A survey of index construction algorithms is properly introduced in [33].

Authors in [28] introduce a collection of string algorithms that lie in the crux of a number of biological problems such as the discovery of potential drug targets, the creation of diagnostic probes, and the universal primers or unbiased consensus sequences. All these problems reduce to the task of identifying a pattern that, with some flaws, occurs in one set of strings (Closest Substring Problem) and does not appear in another (Farthest String Problem).

In addition, all NGS platforms perform sequencing of millions of small fragments of DNA in parallel. The analysis in the area of bioinformatics maps each individual read to the human reference genome with the aim of piecing together the previous fragments. Each of the three billion bases in the human genome is sequenced multiple times, providing thus inner depth for delivering accurate data and an insight into unexpected DNA variation. NGS can also be used for purposes of sequencing entire genomes, including small numbers of individual genes or even all 22.000 coding genes (a whole exome) [5].

[1] `https://blast.ncbi.nlm.nih.gov/Blast.cgi`

3 Longest Common Prefix (LCP)

The LCP array, known as the abbreviation of the phrase "Longest Common Prefix", constitutes a data structure, which is mostly used in combination with the suffix array. More specifically, the array itself contains the length of the longest common prefix of two lexicographically consecutive suffixes [30]. The LCP array is mainly utilized because of its critical information regarding repetitiveness in a given string and can be, therefore, considered as a very advantageous data structure for analysing textual data in several fields such as molecular biology, natural language processing, or musicology. Moreover, sequence variations that may be the result of DNA replication or DNA sequence errors, can also be identified [1, 3, 31]. Other approaches that address the specific problem with the use of LCP-array construction algorithms are considered as well in [14, 16, 26].

Regarding the LCP array, there are numerous text search as well as indexing applications, where the popular ones consist of the construction of the suffix tree, as well as the efficient search of all occurrences of a search pattern in a text.

The algorithms considered in this study for the Longest Common Prefix problem are the following:

- Word by Word Matching
- Character by Character Matching
- Divide and Conquer
- Binary Search

3.1 Word by Word Matching

The Longest Common Prefix problem for the Word by Word Matching Algorithm for n given strings can be considered as

$$LCP(s_1 \dots s_n) = LCP(LCP(LCP(s_1, s_2), s_3), \dots s_n) \tag{1}$$

The Time Complexity of Word by Word Matching Algorithm is $O(n * m)$, where n is the number of strings and m is the length of the largest string. We iterate through all the strings and namely, for each string, we iterate through all of its characters.

3.2 Character by Character Matching

This algorithm differs from the previous one as in the case where there is no common prefix among the given strings, and therefore no need of searching all the strings. Specifically, as this algorithm traverses the characters of each string, once a string that is not common to the other strings is reached, searching stops and it is stated that there is no prefix.

The Time Complexity of Character by Character Matching Algorithm is $O(n * m)$, where n is the number of strings and m is the length of the largest string. We evidently iterate through all the characters of all the strings.

3.3 Divide and Conquer

This algorithm divides a concrete problem into several sub-problems that are similar to the initial problem; in following it recursively solves these sub-problems, and finally combines the solutions derived from the sub-problems in order to solve the initial problem. Because of its recursive function, there is a limitation; each sub-problem must be smaller than the initial problem and there must also be a base case for all corresponding sub-problems [13, 24].

More to the point, these kinds of algorithms constitute of the following three steps:

- Divide the problem into a number of sub-problems that are smaller instances of the same problem.
- Conquer the sub-problems by solving them in a recursive way. If they are small enough, then solve the sub-problems as base cases.
- Combine the solutions of the above sub-problems into the solution corresponding to the initial problem.

This algorithm stems from the associative property of LCP operation. We notice that

$$LCP(S_1 \dots S_n) = LCP(LCP(S_1 \dots S_k) \tag{2}$$

$$LCP(S_{k+1} \dots S_n))LCP(S_1 \dots S_n) = LCP(LCP(S_1 \dots S_k), LCP(S_{k+1} \dots S_n)) \tag{3}$$

where $LCP(S_1 \dots S_n)$ is the longest common prefix in a set of strings $[S_1 \dots S_n]$ with $1 < k < n$.

The Time Complexity of Divide and Conquer Algorithm is $O(n * m)$, where n is the number of strings and m is the length of the largest string. This is since we are iterating through all the characters of all the strings.

3.4 Binary Search

The idea of this algorithm is to apply the well known Binary Search method in order to find the string with maximum value L, which is a common prefix of all the strings [23]. The algorithm searches in the interval $(0 \dots minLen)$, where $minLen$ is of minimum string length and simultaneously has the maximum possible common prefix. At each time period, the search space, which is $(0 \dots minLen)$, is divided in two equal parts; the one of these two is discarded as it doesn't contain the solution.

There are two possible cases:

- The first case assumes that the $S[1 \dots mid]$ is not a common string. This means that for each $j > i$, the string $S[1 \dots j]$ is not a common one and thus, the second half of the search space is discarded.
- The second case assumes that the $S[1 \dots mid]$ is a common string. This means that for each $i < j$, the string $S[1 \dots i]$ is a common one and thus, the first half of the search space is once again discarded (our goal is to find a longer common prefix).

The Time Complexity of Binary Search Algorithm is $O(n * m * logm)$, where n is the number of strings and m is the length of the largest string. This occurs since we use the recurrence relation $T(M) = T(M/2) + O(M * N)$.

4 Longest Common Substring

The Longest Common Substring compares two strings and determines whether they might match by determining the longest length of a sequence of characters (sub-string) that is common to both strings. Specifically, it checks whether that corresponding substring matches exactly or is a part of the given string. The Longest Common Substring is a major problem in the study of strings and it occurs in many different cases in the field of Biology [2, 4, 40]. Specifically, let us consider two strings S and T with length m and n respectively, then the goal is to find the longest strings which are sub-strings of both S and T.

The k-common sub-string problem can be considered as a generalization. Concretely, given the set of strings $S = S_1, \ldots, S_K$, where $|S_i| = n_i$ and $\Sigma n_i = N$, the algorithm finds the longest strings which occur as sub-strings of at least k strings, with $2 \leq k \leq K$.

The algorithms considered in this study for the Longest Common Substring problem are the following:

- Naive Search
- Dynamic Programming
- Suffix Array

4.1 Naive Search

The Naive Search constitutes the simplest method among other pattern searching algorithms. Concretely, it checks whether all the characters of the main string exist in a specific pattern [17]. Furthermore, it is proven to be effective regarding smaller texts and also, it does not require any pre-processing phases. For the identification of a substring, an additional check for the string needs to be performed [27].

The Time Complexity of Naive Search Algorithm is $O(n * m)$, where n is the size of the main string and m is the size of the pattern.

4.2 Dynamic Programming

The Dynamic Programming is considered a powerful enough technique that can be used for solving several different problems in $O(n^2)$ or $O(n^3)$ time, where a naive approach would require an exponential time [17, 22, 37, 40]. One important factor that needs to be taken into account regarding these kinds of problems is the following; if solving a sub-problem is the optimal solution, then this optimal solution for the specific sub-problem must be used [20].

Overall, this method constitutes a general approach for solving problems and resembles "divide-and-conquer" method. The main difference between these two methods is that except that in Dynamic Programming, the sub-problems will

typically overlap. The aim is to somehow split the initial problem into a reasonable number of sub-problems in a way that we can use optimal solutions to the smaller sub-problems; the final output is to provide the initial problem with a near optimal solution. The storing of the concrete solutions can be implemented by using a memory-based data structure, such as an array, a map, etc.

4.3 Suffix Array

The Suffix Array has been introduced by Manber and Myers [30] as a practical and memory-efficient replacement for the suffix tree in string matching applications. The suffix array of a string s having length n is merely an array of these n integers that indicate the lexicographic order of non-empty suffixes of s. Its simplicity and compactness make it an extremely useful tool in modern text processing. Furthermore, the Suffix Array represents in an explicit way all the leaves of the suffix tree, while it omits internal nodes and outgoing edges.

Authors in [32] introduce a linear time and space suffix array construction algorithm, which is novel because of the LMS-substrings used for the problem reduction and the pure induced-sorting used to propagate the order of suffixes as well as that of LMS-substrings.

Definition 1 The Suffix Array of a string S of length n is an array A containing a permutation of the interval $[O, n]$, such that $S_{A[i-1]} <_{lex} S_{A[i]}$ for all $i \in [1, n]$.

5 Longest Common Subsequence (LCS)

The Longest Common Subsequence problem for a given set of sequences constitutes the identification of a common subsequence of all the sequences that has the maximal length [18, 29]. It can be considered as a classic computer science problem as it is the basis of data comparison programs and also has several applications in bioinformatics. Also, LCS addresses various problems in genetics and molecular biology while being used as a measure of similarity between the strings and the biological sequences they represent. In addition, it is widely used by revision control systems, such as SVN and Git in terms of reconciling multiple changes made to a revision-controlled collection of files.

A survey introducing a comprehensive comparison of well-known longest common subsequence algorithms (for two input strings) and in following studying their behaviour in various application environments is presented in [6]. As authors state, the performance of the methods depends heavily on the properties of the problem instance as well as the supporting data structures used in the implementation.

One related work was introduced in [25], where authors presented a randomized algorithm to solve this problem. More to the point, the corresponding algorithm associates with each string X a fingerprint $\phi(X)$, which is shorter enough from the corresponding string. In following, the search procedure compares short fingerprints instead of the initial long strings. Similarly, in [21], new variants of Longest Common Subsequence problem and efficient algorithms for solving them are properly introduced. In particular, authors discuss the notion of gap constraints in corresponding problems.

The algorithms considered in this study for the Longest Common Subsequence (LCS) problem are the following:

- Naive Search
- Dynamic Programming
- Longest Increasing Subsequence (LIS)

5.1 Naive Search

As mentioned in the previous problem, the Naive Search method for this problem considers initially the generation of all the subsequences of the given sequences. In following, the second step constitutes the identification of the longest matching subsequence.

The Time Complexity of the Naive Search Algorithm is exponential. The number of the total possible combinations will be 2^n. Hence, this general recursive solution requires $O(2^n)$.

5.2 Dynamic Programming

The Dynamic Programming method has already been introduced in the Longest Common Substring problem. What is more, each of the sub-problem solutions is indexed based on the values of its input parameters so as to facilitate its lookup. As a result, the time where the same sub-problem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, hence saving computation time. The corresponding technique of storing solutions for the sub-problems, instead of recomputing them, is called memorization.

5.3 Longest Increasing Subsequence (LIS)

The Longest Increasing Subsequence (LIS) problem corresponds to the discovery of the subsequence of a given sequence in which the subsequence's elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible. Notice that this subsequence is not necessary to be continuous or unique. Generally, this method creates a sequence based on the positions of the characters of the one string that match the characters of the other string [10, 15].

Definition 2 A subsequence of sequence $x_1, \ldots, x_n$ is some sequence $x\phi_1, \ldots, x\phi_h$ such that for all k, where $1 \leq k \leq h$, we have $1 \leq \phi_k \leq n$. In addition, for any x_j in the subsequence, all x_i preceding x_j in the subsequence, satisfy that $i < j$. An increasing subsequence constitutes a subsequence such that for any x_j in the subsequence, all x_i preceding x_j in the subsequence satisfy $x_i < x_j$. A largest increasing subsequence is a subsequence of maximum length.

The Time Complexity of the Longest Increasing Subsequence Algorithm is $O(n^2 * logk)$, where n is the length of the strings and k is the maximum length of LCS.

6 Implementation

In this section we will briefly discuss the tools we used to perform the experimental evaluation. The analysis was performed with the use of Apache Spark. We based our experiments on two different DNA sequences and the application we implemented utilised Python language. In the next subsections, the cloud infrastructure, as well as the datasets, are properly introduced.

6.1 Apache Spark

Apache Spark[2] [39] was founded in 2009 at the University of California, Berkley. Although it shares the same principles as Hadoop, its philosophy differs. It uses the abstraction of "Resilient Distributed Dataset" (RDD's), which represent a fault-tolerant correlation of elements, distributed across many compute nodes that can be manipulated in parallel. Using them, a wide range of tasks, including SQL, streaming, machine learning and graph processing, in a unified manner, can be captured. Its main advantage over MapReduce paradigm is that we don't have to flush the intermediate data to the disk, just to read them at the reduce stage, since it can perform iterative computations in memory, which can have a positive impact on the performance [35].

The creators of Apache Spark have also founded Databricks that supplies researchers with a web based platform in which they can store and analyse their data with Spark. It offers researchers a mini cluster with 6 Gb of RAM for their analysis and also cloud storage. As programming language, Python (PySpark) was chosen.

6.2 Datasets

As previously mentioned, the experiments were conducted with DNA sequences derived from the database of the National Center for Biotechnology Information (NCBI). More specifically, these sequences are part of the genomes entitled Escherichia coli K-12 [34] as well as Streptococcus pneumoniae R6 [19].

7 Results

The results of our work are presented in the following Tables 1 to 6 for different number of input strings as well as different number of strings characters. The execution time (in milliseconds and seconds) is used as the evaluation metric of the different algorithms.

Regarding Table 1, we present four different experiments for the problem of the Longest Common Prefix. In the first case, the number of characters for the three input strings is 450, 300 as well as 125 respectively and the output consists of an LCP with 76 characters. In following, in the second case, the number of characters for input strings is 2605, 2455 as well as 2060 respectively and the output consists

[2] http://spark.apache.org/

of an LCP with 250 characters. For the last two cases, the number of characters for input strings is 9948, 8884, 8504 and 21350, 18790, 16845 whereas the output consists of an LCP with 1054 and 2530 characters respectively.

Table 1 Four different scenarios for Longest Common Prefix Implementations

Scenario	Number of Input Strings	Longest Common Prefix Output
1	450, 300, 125	76
2	2605, 2455, 2060	250
3	9948, 8884, 8504	1054
4	21350, 18790, 16845	2530

Binary Search, as expected, achieved the best performance in Table 2, while the other three algorithms, although they almost have the same complexity, in fact are quite different. Binary Search takes the lowest time as it examines each character starting from the first one until the one that is in the position equal to the length of the smallest string. Divide and Conquer is the next best performance as it divides the problem into smaller sub-problems and calculates the final solution through solutions in each sub-problem. Word by Word Matching is the slowest, as it has to look at all the strings and the prefix, which occur each time between the pairs of the strings and may be larger than the final prefix, so there is the possibility of additionally unnecessary calculations. On the other hand, Character by Character Matching, instead of going through the strings one by one, looks at the characters separately. So, once a character in any string is not the same with the other, the query stops and the output consists of the occurred prefix; moreover, no additional unnecessary calculations need to be done.

Table 2 Time for different scenarios of Longest Common Prefix Implementations

Longest Common Prefix Algorithm	1	2	3	4
Word by Word Matching	2,02	2,88	3,58	5,75
Character by Character Matching	1,55	2,68	3,39	5,39
Divide and Conquer	1,52	1,73	1,97	2,61
Binary Search	0,95	1,09	1,28	1,44

Furthermore, results in Table 3 introduce the problem of the Longest Common Substring. We also present four different experiments, where in the first case, the number of characters for the two input strings is 300 as well as 280 respectively and the output consists of a substring with 7 characters. In the second case, the number of characters for input strings is 4575 as well as 4270 respectively and the output consists of a substring with 13 characters. For the last two cases, the number of characters for input strings is 7500, 7000 and 14925, 14070 whereas the output consists of a substring with 13 and 23 characters respectively.

The results in Table 4 show that the Naive Search algorithm is the slowest one as it requires, after the identification of all the possible strings of the first input, to

Table 3 Four different scenarios for Longest Common Substring Implementations

Scenario	Number of Input Strings	Longest Common Substring Output
1	300, 280	7
2	4575, 4270	13
3	7500, 7000	13
4	14925, 14070	23

additionally check whether each one of them is substring of the initial; the largest one will be the desired Longest Common Substring. The Dynamic Programming method performs better because in this case, a table that contains the lengths of the maximum common suffixes of the two strings, will be created. In following, as already mentioned, every corresponding output will be stored in the table in order to be used in calculations that precede. On the other hand, the Suffix Array approach, even if quite simple, is faster than the previous two algorithms. It is considered a simple data structure that contains all the information needed for finding the Longest Common Substring; a table is created with all the possible string suffixes resulting from the combination of input strings and after being sorted in lexicographic order using an LCP algorithm, we find all the LCPs between each value with the exactly next one. The largest of these LCPs constitutes the desired Longest Common Substring.

Table 4 Time for different scenarios of Longest Common Substring Implementations

Longest Common Substring Algorithm	1	2	3	4
Naive Search	3,82	$14,16 * 10^3$	$58,14 * 10^3$	$482,53 * 10^3$
Dynamic Programming	0,01	3,32	8,02	33,22
Suffix Array	0,003	0,22	0,42	1,59

Finally, Table 5 presents the problem of the Longest Common Subsequence, where in the first case, the number of characters for both two input strings is 20 and the output consists of an LCS with 9 characters. In the second case, the number of characters for the two input strings is 83 as well as 81 respectively and the output consists of an LCS with 46 characters. For the last two cases, the number of characters for input strings is 158, 155 and 300, 280 whereas the output consists of an LCSub with 90 and 179 characters respectively.

Table 5 Four different scenarios for Longest Common Subsequence Implementations

Scenario	Number of Input Strings	Longest Common Subsequence Output
1	20, 20	9
2	83, 81	46
3	158, 155	90
4	300, 280	179

As in the Longest Common Substring Implementations, we observe in Table 6 that the Naive method for calculating the Longest Common Subsequence is the slowest. Specifically, its complexity is exponential and in the worst case reaches $O(2^n)$. This undoubtedly proves that it is unsuitable for long sequences like the ones used for computations in the field of bioinformatics. On the other hand, the Dynamic Programming is clearly a faster method because it uses a table for storing the temporary results in order to be used in subsequent calculations. The LIS approach achieves the medium performance as initially, it creates a sequence based on the positions of the string characters and then, a LIS algorithm that will produce the desired LCS, is applied.

Table 6 Time for different scenarios of Longest Common Subsequence Implementations

Longest Common Subsequence Algorithm	1	2	3	4
Naive Search	102,48	-	-	-
Dynamic Programming	0,001	0,009	0,012	0,037
Longest Increasing Subsequence (LIS)	0,006	0,62	21,41	320,53

8 Conclusions

The aim of this work was to study DNA sequences regarding three well known problems, namely, the Longest Common Prefix, the Longest Common Substring and the Longest Common Subsequence. The application was developed in Apache Spark environment with Python programming language. The use of Spark has accelerated the processing of large-scale biological sequences, while it has also contributed to the versatility of the use of Python.

It would be interesting to analyze the Longest Common Extension (LCE) problem that appears to be a sub-problem in several fundamental problems with strings such as the k-Difference Global Alignment for the construction of alignment tools in bioinformatics. Moreover, another potential future work is to incorporate in our experiments the use of Suffix Tree in the Longest Common Substring problem. In addition, the ongoing research is aimed at investigating the performance of algorithms using various text compression algorithms. To be more specific, these algorithms will take full advantage of genomic sequence data.

References

1. Alamro H, Ayad LAK, Charalampopoulos P, Iliopoulos CS, Pissis SP (2018) Longest common prefixes with k-mismatches and applications. In: 44th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM), pp 636–649
2. Arnold M, Ohlebusch E (2011) Linear time algorithms for generalizations of the longest common substring problem. Algorithmica 60(4):806–818

3. Ayad LAK, Barton C, Charalampopoulos P, Iliopoulos CS, Pissis SP (2018) Longest common prefixes with k-errors and applications. In: 25th International Symposium on String Processing and Information Retrieval (SPIRE), pp 27–41
4. Babenko MA, Starikovskaya TA (2008) Computing longest common substrings via suffix arrays. In: Computer Science - Theory and Applications, Third International Computer Science Symposium in Russia (CSR), pp 64–75
5. Behjati S, Tarpey PS (2013) What is next generation sequencing? Archives of Disease in Childhood-Education and Practice 98(6):236–238
6. Bergroth L, Hakonen H, Raita T (2000) A survey of longest common subsequence algorithms. In: Seventh International Symposium on String Processing and Information Retrieval (SPIRE), pp 39–48
7. Blumer A, Blumer J, Haussler D, Ehrenfeucht A, Chen MT, Seiferas J (1985) The smallest automation recognizing the subwords of a text. Theoretical Computer Science 40:31–55
8. Bockenhauer HJ, Bongartz D (2007) Algorithmic Aspects of Bioinformatics. Springer
9. Crochemore M, Lecroq T (2009) Trie. In: Encyclopedia of Database Systems, pp 3179–3182
10. Crochemore M, Porat E (2010) Fast computation of a longest increasing subsequence and application. Information and Computation 208(9):1054–1059
11. Crochemore M, Vérin R (1997) On compact directed acyclic word graphs. In: Structures in Logic and Computer Science, A Selection of Essays in Honor of Andrzej Ehrenfeucht, pp 192–211
12. Farach M (1997) Optimal suffix tree construction with large alphabets. In: 38th Annual Symposium on Foundations of Computer Science (FOCS), pp 137–143
13. Farach M, Ferragina P, Muthukrishnan S (1998) Overcoming the memory bottleneck in suffix tree construction. In: 39th Annual Symposium on Foundations of Computer Science (FOCS), pp 174–185
14. Fischer J (2011) Inducing the lcp-array. In: 12th International Symposium on Algorithms and Data Structures (WADS), pp 374–385
15. Garcia T, Myoupo JF, Seme D (2001) A work-optimal cgm algorithm for the longest increasing subsequence problem. In: International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), vol 2, pp 563–569
16. Gog S, Ohlebusch E (2011) Fast and lightweight lcp-array construction algorithms. In: 13th Workshop on Algorithm Engineering and Experiments (ALENEX), pp 25–34
17. Gusfield D (1997) Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press
18. Hirschberg DS (1977) Algorithms for the longest common subsequence problem. Journal of the ACM 24(4):664–675
19. Hoskins J, Alborn WE, Arnold J, Blaszczak LC, Burgett S, DeHoff BS, Estrem ST, Fritz L, Fu DJ, et al (2001) Genome of the bacterium streptococcus pneumoniae strain r6. Journal of Bacteriology 183(19):5709–5717
20. Hsu WJ, Du MW (1984) New algorithms for the LCS problem. Journal of Computer and System Sciences 29(2):133–152

21. Iliopoulos CS, Rahman MS (2008) Algorithms for computing variants of the longest common subsequence problem. Theoretical Computer Science 395(2-3):255–267
22. Iliopoulos CS, Rahman MS (2008) New efficient algorithms for the LCS and constrained LCS problems. Information Processing Letters 106(1):13–18
23. Irving RW, Love L (2003) The suffix binary search tree and suffix avl tree. Journal of Discrete Algorithms 1(5-6):387–408
24. Kärkkäinen J, Sanders P (2003) Simple linear work suffix array construction. In: 30th International Colloquium on Automata, Languages and Programming (ICALP), pp 943–955
25. Karp RM, Rabin MO (1987) Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development 31(2):249–260
26. Kasai T, Lee G, Arimura H, Arikawa S, Park K (2001) Linear-time longest-common-prefix computation in suffix arrays and its applications. In: 12th Annual Symposium on Combinatorial Pattern Matching (CPM), pp 181–192
27. Knuth DE, Jr JHM, Pratt VR (1977) Fast pattern matching in strings. SIAM Journal on Computing 6(2):323–350
28. Lanctôt JK, Li M, Ma B, Wang S, Zhang L (2003) Distinguishing string selection problems. Information and Computation 185(1):41–55
29. Lowrance R, Wagner RA (1975) An extension of the string-to-string correction problem. Journal of the ACM 22(2):177–183
30. Manber U, Myers EW (1993) Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing 22(5):935–948
31. Manzini G (2015) Longest common prefix with mismatches. In: 22nd International Symposium on String Processing and Information Retrieval (SPIRE), pp 299–310
32. Nong G, Zhang S, Chan WH (2009) Linear suffix array construction by almost pure induced-sorting. In: Data Compression Conference (DCC), pp 193–202
33. Nsira NB, Lecroq T, Elloumi M (2017) Algorithms for indexing highly similar DNA sequences. In: Algorithms for Next-Generation Sequencing Data, Techniques, Approaches, and Applications, pp 3–39
34. Rudd KE (2000) Ecogene: A genome sequence database for escherichia coli K-12. Nucleic Acids Research 28(1):60–64
35. Shi J, Qiu Y, Minhas UF, Jiao L, Wang C, Reinwald B, Özcan F (2015) Clash of the titans: Mapreduce vs. spark for large scale data analytics. PVLDB 8(13):2110–2121
36. Ukkonen E (1995) On-line construction of suffix trees. Algorithmica 14(3):249–260
37. Wagner RA, Fischer MJ (1974) The string-to-string correction problem. Journal of the ACM 21(1):168–173
38. Weiner P (1973) Linear pattern matching algorithms. In: 14th Annual Symposium on Switching and Automata Theory (SWAT), pp 1–11
39. Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, Meng X, Rosen J, Venkataraman S, Franklin MJ, Ghodsi A, Gonzalez J, Shenker S, Stoica I (2016) Apache spark: a unified engine for big data processing. Communications of the ACM 59(11):56–65
40. chao Zhang Y, Che M, Ma J (2007) Analysis of the longest common substring algorithm. Computer Simulation 12:025