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

Towards Analyzing Next Generation Sequencing Algorithms

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 the other in an extraordinary way. This is the case for DNA sequences produced by new technologies of high-throughput Next Generation Sequencing (NGS). In this paper, we perform some experiments using Apache Spark in some sequences derived from National Center for Biotechnology Information (NCBI). The problems we deal with are some of the most popular, namely, Longest Common Prefix (LCP), Longest Common Substring (LCS) and Longest Common Subsequence (LCSub).

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

1 Introduction

Current scientific advancements in both computer and biological sciences are bringing new opportunities to intra-disciplinary research topics. On one hand, computers and big-data analytics cloud software tools are being developed, rapidly increasing the capability of processing from terabyte data sets to petabytes and beyond. On the other hand, the advancement in molecular biological experiments is producing huge amounts of data related to genome and RNA sequences, protein and metabolite abundance, proteinprotein interactions, gene expression, and so on. In most cases, biological data are forming big, versatile, complex networks.

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

I.E. Livieris
Department of Computer & Informatics Engineering (DISK Lab)
Technological Educational Institute of Western Greece 26334, Greece
E-mail: livieris@teiwest.gr

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

Sequencing has seen major scientific progress in recent years and has leveraged the development of novel bioinformatic applications. Consequently, 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 store, transfer, and process meet congestion in many current technologies.

A deoxyribonucleic acid (DNA) macromolecule can be coded by a sequence over a four-letter alphabet. These letters are A, C, G, and T, and they code respectively the bases Adenine, Cytosine, Guanine and Thymine. DNA sequencing consists then in determining the exact order of these bases in a DNA macromolecule. As a matter of fact, DNA sequencing technology is playing a key role in the advancement of molecular biology. Compared to previous sequencing machines, Next-Generation Sequencing (NGS) machines function much faster, with significantly lower production costs and much higher throughput in the form of short reads, i.e., short sequences coding portions of DNA macromolecules.

Several data structures have been designed that enable storing these impressive amounts of data efficiently, while allowing to search quickly among them. Thus, all occurrences of any given pattern can be found without traversing the whole sequences. In short, indexing is profitable and useful if it is used regularly. So, one takes advantage of the index if sequences are available beforehand, very long and do not change periodically. In general, the aim constitutes the construction of an index that provides efficient answers to queries with reasonable building and maintenance costs. Basic, also called classical data structures such as tries [2], suffix trees [4, 8, 9] and suffix arrays [5], Directed Acyclic Word Graphs (DAWG) [1] and the compact version CDAWG [3] are arguably very interesting data structures for string analysis, especially when searching over large sequence collections. Yet, these structures are full-text indexes: they require a large amount of space to represent a sequence. A survey of index construction algorithms is properly introduced in [6].

The remainder of the paper is organized as follows. Section 2 discusses Longest Common Prefix (LCP) along with four different algorithms. Same wise, Sections 3 and 4 present Longest Common Substring (LCS) and Longest Common Subsequence (LCSub) respectively. In addition, Sections 5 and 6 introduce the details of the implementation of the system (and the respective cloud infrastructure utilized) as well as the experimental results. Finally, in Section 7 our concluding remarks, open problems and future work are introduced.

2 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.

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 (LCP) problem are the following:

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

2.1 Word by Word Matching

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

$$LCP(s_1 \dots s_n) = LCP(LCP(LCP(s_1, s_2), s_3), \dots s_n) \quad (1)$$

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. This is since we are iterating through all the strings and namely, for each string we are iterating through all of its characters.

2.2 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, there is 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, then searching stops and state 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. This is since we are iterating through all the characters of all the strings.

2.3 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, each sub-problem must be smaller than the initial problem and also there must be a base case for all corresponding sub-problems.

More to the point, this kind of algorithms constitutes 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.

In addition, the idea of this algorithm comes 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 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.

2.4 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 common prefix of all the strings. 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 discarded (because we try to find 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 is since we are using the recurrence relation $T(M) = T(M/2) + O(M * N)$.

3 Longest Common Substring (LCS)

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 values; whether that substring represents the whole or a part of the given string.

Given two strings S and T with length m and n respectively, 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 $S = S_1, \dots, S_K$, 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 (LCS) problem are the following:

- Naive Search
- Dynamic Programming
- Suffix Array

3.1 Naive Search

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. Furthermore, it is proven to be effective regarding smaller texts and also, it does not need any pre-processing phases. For the identification of a substring, and additional check for the string needs to be performed.

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 pattern.

3.2 Dynamic Programming

Dynamic Programming can be considered as 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. One important that needs to be taken into account regarding this kind of problems is that if the optimal solution involves solving a sub-problem, then the optimal solution for the specific sub-problem must be used.

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 goal 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 to 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.

3.3 Suffix Array

The Suffix Array has been introduced by Manber and Myers [5] 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, Suffix Array represents in an explicit way all the leaves of the suffix trie, while it omits internal nodes and outgoing edges.

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 $i \in [1, n]$.

4 Longest Common Subsequence (LCSub)

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. 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. 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.

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

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

4.1 Naive Search

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

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

4.2 Dynamic Programming

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 memoization.

4.3 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 contiguous 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.

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

5 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 was using Python language. In the next subsections, the cloud infrastructure as well as the datasets are properly introduced.

5.1 Apache Spark

Apache Spark[1] [10] 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 [7].

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.

5.2 Datasets

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

6 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

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

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 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 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 LCP Implementations

Scenario	Number of Input Strings	LCP 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 they 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 one 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 is has to look at all the strings and the prefix, which occurs each time between the pairs of the strings, 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, it looks at the characters one by one. So, once a character in any string is not the same with the other, then the query stops and the output consists of the occurred prefix; moreover, no additionally unnecessary calculations need to be done.

Table 2 Time for different scenarios of LCS Implementations

LCP 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 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 an LCS 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 an LCS 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 an LCS with 13 and 23 characters respectively.

Table 3 Four different scenarios for LCS Implementations

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

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 additionally check whether each one of them is substring of the initial; the largest one will be the desired LCS. Dynamic Programming method performs better as a table that will contain 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 following calculations. On the other hand, the Suffix Array approach, even 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 LCS; 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 consitutes the desired LCS.

Table 4 Time for different scenarios of LCS Implementations

LCS 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 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 LCSub 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.

As in LCS Implementations, we observe in Table 6 that the Naive method for calculating LCSub is the slowest. Specifically, its complexity is exponential and in worst case is $O(2^n)$. That proves us that it is unsuitable for long sequences like the ones used for computations in the field of bioinformatics. On the other hand, Dynamic Programming is clearly the faster method because it uses a table for storing the temporary results in order to be used in subsequent calculations. The

Table 5 Four different scenarios for LCSub Implementations

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

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 LCSub, is applied.

Table 6 Time for different scenarios of LCSub Implementations

LCSub 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

7 Conclusion

The aim of this work was to study DNA sequences regarding three well known problems, namely, Longest Common Prefix (LCP), Longest Common Substring (LCS) and Longest Common Subsequence (LCSub). The application was developed in Apache Spark environment with the 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.

References

1. 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
2. Crochemore M, Lecroq T (2009) Trie. In: Encyclopedia of Database Systems, pp 3179–3182
3. 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
4. Farach M (1997) Optimal suffix tree construction with large alphabets. In: 38th Annual Symposium on Foundations of Computer Science (FOCS), pp 137–143
5. Manber U, Myers EW (1993) Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing 22(5):935–948

6. 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
7. 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
8. Ukkonen E (1995) On-line construction of suffix trees. Algorithmica 14(3):249–260
9. Weiner P (1973) Linear pattern matching algorithms. In: 14th Annual Symposium on Switching and Automata Theory (SWAT), pp 1–11
10. 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