An average-case efficient two-stage algorithm for enumerating all longest common substrings of minimum length k between genome pairs

1st Mattia Prosperi
dept. of Epidemiology
University of Florida
Gainesville, FL (USA)
m.prosperi@ufl.edu

2nd Simone Marini
dept. of Epidemiology
University of Florida
Gainesville, FL (USA)
simone.marini@ufl.edu

3rd Christina Boucher
dept. of CISE
University of Florida
Gainesville, FL (USA)
cboucher@cise.ufl.edu

2024 IEEE 12th International Conference on Healthcare Informatics (ICHI) | 979-8-3503-8373-7/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICHI61247.2024.00020

***Abstract*—A problem extension of the longest common substring (LCS) between two texts is the enumeration of all LCSs given a minimum length k (ALCS-k), along with their positions in each text. In bioinformatics, an efficient solution to the ALCS-k for very long texts –genomes or metagenomes– can provide useful insights to discover genetic signatures responsible for biological mechanisms. The ALCS-k problem has two additional requirements compared to the LCS problem: one is the minimum length k, and the other is that all common strings longer than k must be reported. We present an efficient, two-stage ALCS-k algorithm exploiting the spectrum of text substrings of length k (k-mers). Our approach yields a worst-case time complexity loglinear in the number of k-mers for the first stage, and an average-case loglinear in the number of common k-mers for the second stage (several orders of magnitudes smaller than the total k-mer spectrum). The space complexity is linear in the first phase (disk-based), and on average linear in the second phase (disk- and memory-based). Tests performed on genomes for different organisms (including viruses, bacteria and animal chromosomes) show that run times are consistent with our theoretical estimates; further, comparisons with MUMmer4 show an asymptotic advantage with divergent genomes.**

***Index Terms*—Computational Biology, Bioinformatics, Algorithms, Biological Sequences**

I. Introduction

The longest common substring (LCS) problem is defined as to find the longest string –which might not exist or be unique– in common between two or more texts. Applications of LCS algorithms include plagiarism detection, text clustering, and several uses in bioinformatics, e.g., finding common genes, or conserved gene signatures among species. Dynamic programming solves the problem with a runtime complexity quadratic in the text length, while the space complexity varies between quadratic and linear depending on optimization. Subquadratic and linear algorithms exist, e.g., rolling hashes and suffix trees [1]. The rolling hash time complexity is quadratic loglinear, but the algorithm is limited to very short strings due to collisions, as hash codes are integer types. The LCS search in the suffix tree runs in linear time and theoretically in linear space, but in-memory implementations have large multipliers, and disk/distributed ones often increase space complexity to quadratic. An extension of the LCS problem is to find all longest common substrings between two texts, given a minimum length k (ALCS-k), along with their positions in each text. While general solutions for more than two texts and for enumerating ALCS exist, to date we are not aware of available approaches or implemented tools solving the ALCS-k problem with well-characterized time and space complexity. However, the constraint of a minimum string length k and the overlap conditions of all strings of length k, i.e., k-mers, open room for devising efficient approaches that reduce the number of the strings to be considered in the search or indexed into a suffix-based structure. As with LCS, an efficient solution to the ALCS-k can provide useful insights for many bioinformatics use cases at large, e.g., for motif finding, or identifying mobile elements, cargo genes, antimicrobial resistance genes in bacteria.

We here propose an ALCS-k solution and a software implementation. Our approach is a two-stage method that first filters out all non-common k-mers between two text (using external sort and merge), and then elongates the remaining k-mers to identify the longest in common (eliminating those that do not appear in both texts after elongation). We will show that the worst-case time complexity of our approach is loglinear in the number of k-mers for the first stage, and then average-case loglinear in the number of common k-mers for the second stage (several order of magnitudes smaller than the total k-mer spectrum). We also we provide a theoretical proof of the expected size for the ALCS-k set, which is an alphabet-dependent fraction of the number of shared k-mers. The space complexity for the first phase is equivalent to the k-mer spectrum (implemented as disk-based), naively linear in the text lengths, while for the second phase is linear in the number of common k-mers (disk- and memory-based).

II. Methods

A. Problem Definition

Let A be an alphabet (finite set) composed by $a = |A|$ symbols, e.g., A can be the set of nucleotide bases $A =$\{A,

Identify applicable funding agency here. If none, delete this.

2575-2634/24/$31.00 ©2024 IEEE
DOI 10.1109/ICHI61247.2024.00020

C, G, T} for genetic sequences. Let there be t, a text of length l_t generated upon A, and s_t a substring of t, i.e., a subsequence of t made of consecutive characters, with an associated length $0 \leq l_s \leq l_t$. Recursively, we can define s_{s_t} as a substring of s_t, with associated length $l_{s_{s_t}}$. We also define the location p of s_t in t where the substring starts, i.e., with the position of the first character; if the substring appears multiple times, it will be located in multiple positions. Given two texts t_1 and t_2, we define $s_{1,2}$ any substring that is a substring of both t_1 and t_2. A string $s_{1,2}$ can appear f_1 and f_2 times in t_1 and t_2, respectively. The number of substrings in common is bounded between 0 and $\min(l_1 - l_{s_{1,2}} + 1, l_2 - l_{s_{1,2}} + 1)$, and the maximum can be reached only when $t_1$is a substring of t_2.

By choosing a minimum length k, we define the set S_t^k, made of all substrings of length k from t, indexed by their positions p_ss, where $|S_t^k| = l_t - k + 1$, i.e., the generic element $s_t^k \in S_t^k$ is the tuple $< s^k, p >$ made by the k-mer s^k and its position p. Upon S_t^k, we further define the set $S_{1_2}^k$ the set of all substrings of a text t_1 that appear at least once also in a text t_2, and $S_{2_1}^k$ as the set of all t_2 substrings that are found in t_1. The two sets S_{1_2} and S_{2_1} contain the same distinct k-mers, but they can have different multiplicity and different positions in the respective texts. We thus define $S_{1,2}^k$ as the union of S_{1_2} and S_{2_1}.

Let $s_i = \{s_{i,1} \dots s_{i,k}\}$ and $s_j = \{s_{j,1} \dots s_{j,k}\}$ be two strings in $S_{1_2}^k$, and p_i, p_j their respective starting positions. If $p_j = p_i + 1$, we define the elongated string $s_e = \{s_{i,1} \dots s_{i,k}, s_{j,k}\}$ when $s_{i,w} = s_{j,w+1}, \forall w = 2 \dots k$. The string elongation can be iteratively applied until $p_j \neq p_i + 1$. For instance, given the text t and the set $S_t^k, k = 18$

position p_i	string s_i
000101573337	AAAGAAAAAATATAAATT
000101573338	AAGAAAAAATATAAATTC
000101573339	AGAAAAAATATAAATTCT
000101573340	GAAAAAATATAAATTCTG
000101580000	TTTGGCCTTAGCTAAAAG

$s_e =$ AAAGAAAAAATATAAATTCTG will be an elongated string of t, with $p_e =$ 000101573337 and $l_{s_e} = 21$.

In the next sections we will show how, by elongating all strings in $S_{1_2}^k$ and all strings in $S_{2_1}^k$, we can create the elongated sets $S_{1_2}^e$ and $S_{2_1}^e$, where the elongated strings from one set do not necessarily coincide from the other, and then reduce them to have the same length, deriving the set $S_{1,2}^e$, i.e., the solution to the ALCS-k problem.

B. Related Work

The classic solution to the LCS is by dynamic programming and has a quadratic complexity of $\mathcal{O}(l_{min} \cdot l_{max})$ [2], where l_{min} is the length of the shortest text, and l_{max} is the length of the longest, among the two. The space complexity is also quadratic, since a $l_{min} \cdot l_{max}$ matrix is used in the algorithm. Optimizations of the lookup matrix can reduce the space complexity up to linear, e.g., $\mathcal{O}(l_{min})$, but the time complexity remains quadratic [3].

Several subquadratic time algorithms exist. The rolling hash algorithm has a quadratic loglinear time complexity and linear space complexity. Similarly to ours, this approach operates on k-mers, by finding first common substrings of length k, and then searching for the maximum k. In detail, the rolling hash can be computed in linear time, sorting the hashes requires $\mathcal{O}(l_{max}\log(l_{max}))$ time, and the binary search requires another $\mathcal{O}(\log(l_{max})$ time. Therefore, the total time complexity of finding the LCS is $\mathcal{O}(l_{max}(\log l_{max})^2)$. Since the prefixes and hashes must be stored, the space complexity is $\mathcal{O}(l_{max})$. As a serious drawback, the rolling hash can produce hash collision as soon as the number of possible strings exceeds the available integers that can be stored in memory. In practice, even with 128-bit integers, the maximum string length handled by rolling hashes over a 4-letter alphabet (genomes) is 64, which makes the approach inapplicable in many real-world use cases. Also, the search on hash matches need to be implemented and stored efficiently, e.g., with hash tables, binary search trees, posing further issues with large texts.

The most efficient approaches use suffix-based data structures and yield linear time/space complexity, i.e., $\mathcal{O}(l_{max})$ with Ukkonen's online algorithm [4], or even sublinear, i.e., $\mathcal{O}(l_{max}\log a/\sqrt{\log l_{max}})$ (where a is alphabet size) time and $\mathcal{O}(l_{max}\log a/\log l_{max})$ space, when word RAM models are used [5]. However, even if the space requirement is linear in the size of the text, the large amount of information required at each node of the suffix tree makes the memory requirements very expensive (20x), even with optimized implementations [6]. Suffix arrays are more efficient (4x) [7], and succinct/compressed structures have been lately introduced improving by several order of magnitudes the storage needs, e.g., FM-index [8], [9], at a slight price of increased querying complexity [10].

For large texts, disk-based and parallel algorithms have been proposed [11], [12]. Notably, most of the disk-based approaches yield quadratic complexity for the construction and the same holds for distributed ones [13], [14], with recent improvements using Cartesian trees (linear work and space, and polylogarithmic time) by Shun and Blelloch [15].

The ALCS-k problem has two additional requirements compared to the LCS problem: one is the minimum length k, and the other is that all common strings longer than k must be reported. The problem is solved efficiently with suffix-based structures [16]. However, highly dissimilar texts can still pose memory and run time issues. The pre-specification of k can be used to reduce search space in such cases. In fact, our approach exploits k-merization, which can be done very efficiently, with a plethora of in-memory, disk-based, distributed, and compressed solutions (loglinear time and linear space complexity for basic serial implementation), thus a very valid alternative to using suffix-based structures. Once the k-mers are extracted, compared and filtered, we work only on the space of common k-mers, finding the ALKS-k set within a lower-dimensional space and time complexity, i.e., a constant fraction of the expected number of common k-mers, well-manageable as an in-memory process.

Since it is related to our idea, we believe it is worth mentioning the Hunt-Szymanski approach [17] for solving the more general problem of the longest common subsequence, which relaxes the criterion of character contiguity from the LCS. Similarly to ours, the algorithm breaks the two texts into smaller chunks –in its implementation for the UNIX/Linux `diff` command these are file lines– and then creates k-bit integer hashes, which are used in the dynamic programming steps. If the chunks do not repeat often, the integer hashes will resemble random strings from an alphabet of size 2^k, and the algorithm will be efficient as expected. The algorithm is subquadratic in the average case, i.e., $\mathcal{O}((r+l_{max})\log(l_{max}))$, where r is the number of character pairs where the two texts match. Since r is at maximum $l_1 l_2$, in the worst case the algorithm has a time complexity of $\mathcal{O}(l_{max}^2 \log(l_{max}))$ and a space of $\mathcal{O}(r + l_{max})$. However, for strings whose characters are drawn uniformly at random from an alphabet of size a, on average $r \sim l_{max}$, and a running time of $\mathcal{O}(l_{max}\log(l_{max}))$ can be expected [17]. As a drawback, the Hunt-Szymanski algorithm needs a good hashing function, which limits the maximum length of a chunk, and a few repeats over the texts –unfortunately uncommon in genomes. The repeats issue has been lately improved by Apostolico [18].

Finally, we reference also the Chvátal-Sankoff constants [19] that estimate the lengths of longest common subsequences of random strings, since there is a relationship between the average length of the LCS and the number of common k-length strings.

C. Proposed Approach

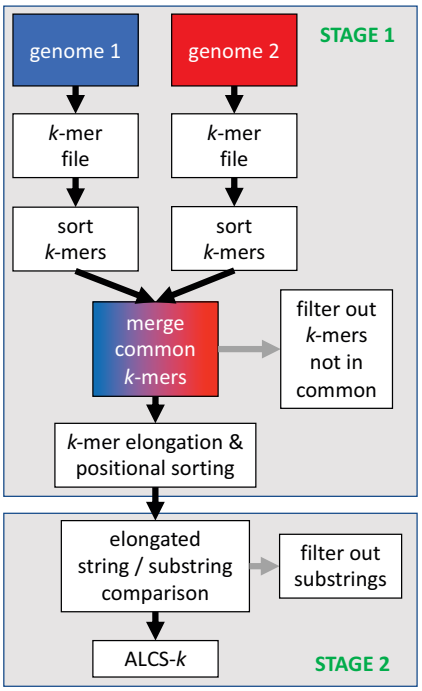
Let t_1 and t_2 be the two texts to be considered for the ALCS-k problem. The algorithm solving the problem will output all strings longer than k in common between the two texts, i.e., the set $S^e_{1,2}$. Let us make an example as follows:

```
text  position  string
t1 000000000010    CTTCCCGGAAAGG
t1 000000000030   AGTTCCCGGAAA
t1 000000000045  GAGTTCCCGGAAA
t1 000000000060 GGAGTTCCCGGAAA
t1 000000000077 GGAGTTCCCGGAAAT
t2 000000000005    CTTCCCGGAAAGG
t2 000000000024 GGAGTTCCCGGAAAT
t2 000000000040 GGAGTTCCCGGAAA
```

From the example, the second string of t_1, `AGTTCCCGGAAA` at position 30, is a substring of the t_1 strings at positions 45, 60, and 77; however, it is included because it has a different starting position (and the same applies in turn for those at positions 45 and 60). Instead, we will not consider any substring of a shared string if its starting and ending positions are within the starting and the ending position of the longer one.

The inclusion of substrings that are found in different positions is a positional extension to the classical ALCS-k problem. Since in order to derive $S^e_{1,2}$, our algorithm also calculates $S^e_{1_2}$ and $S^e_{2_1}$, it is easy to find all the strings of one text that are substrings of one of the longest common substrings, yet are also found in other positions only in one of the two texts. These strings can have an importance in certain genomic domains, e.g., gene duplication, mobile elements, promoters, miRNA, sncRNA.

Fig. 1. Flowchart of the two-stage ALCS-k algorithm.



The two-stage procedure is illustrated in Figure 1. The first stage reads the two input texts t_1 and t_2 and prints all their k-mers along with the starting positions, which takes $\mathcal{O}(l_{t_1} + l_{t_2})$ time. The two files are then sorted independently in $\mathcal{O}(l_{t_1}\log(l_{t_1}) + l_{t_2}\log(l_{t_2}))$ and then merged into a single k-mer set file filtering out k-mers that are not in common in $\mathcal{O}(l_{t_1}+l_{t_2})$, obtaining the $S^k_{1,2}$ set. Thus, the overall time complexity of the first stage $\mathcal{O}(l_{t_1} + l_{t_2} + l_{t_1}\log(l_{t_1}) + l_{t_2}\log(l_{t_2}))$ is bounded by the $\mathcal{O}(l_{max}\log l_{max})$ superset. The space complexity is at best linear, since all k-mers from both texts are stored in a file at some point is $k(l_{t_1}+l_{t_2})$, i.e., $\mathcal{O}(l_{max})$, and quick sort or merge sort take linear space. The second stage starts by sorting positionally the $S^k_{1,2}$ set, which yields again loglinear time complexity and linear space, but is bounded by the time complexity of stage one, since $|S^k_{1,2}| < |S^k_{t_1}| + |S^k_{t_2}|$. After positional sorting, all the common k-mer strings are elongated, constructing and filtering the $S^e_{1_2}$ and $S^e_{2_1}$ sets, and then retaining the ALCS-k set. In the following paragraphs, we will provide the theoretical proof that the number of elongated strings $|S^e_{1_2}|$ and their substrings on average are a fraction of $|S^k_{1,2}|$, and thus the average time/space complexities are bounded by the prior loglinear/linear superset.

a) Estimation of the average number of shared k-mers.: The calculation of occurrence distribution of strings within a text has been largely studied, and several exact and approximated formulae exist, for both Bernoullian and Markovian assumptions [20]. According to the formula by Prosperi *et al.* [21], the probability distribution for a string s of length k, $s = \{a_1, \ldots, a_k\}$, within a text t of length n, over an alphabet of size a, for j occurrences under the Markovian model is

$$\Pr(|s \cap S_t^k| = j) = \Pr(s)^j \sum_{z=1}^{|C_{k,n,j}|} \prod_{y=1}^{j+1} \Pr(s_{0,d_{yz}}), \quad (1)$$

where $\Pr(s) = \Pr(a_1) \cdot \Pr(a_2|a_1) \cdot \cdots \cdot \Pr(a_{k-1}|a_k)$, $\Pr(s_{0,n}) = \Pr(s_{0,n-1}) - \Pr(s) \cdot \Pr(s_{0,n-k})$, $s_{0,n} = s_{0,n-1} \cdot a - \Pr(s) \cdot a^k \cdot s_{0,n-k}$, $d_1 \ldots d_{j+1}$ are the lengths of the $j+1$ segments where the j strings divide the text of length n in exact configurations with $d_1 + \cdots + d_{j+1} = n - kj$, $d_i \in \mathrm{N}$, $C = \{(d_1, d_2, \ldots, d_{j+1}) | d_i\}$, and $|C_{k,n,j}| = \binom{n+j(1-k)}{n-kj}$. In other words, the set C represents all the ways in which $n - kj$ characters can be distributed in $j+1$ positions, where $C_c = \{d_{1c} \ldots d_{(j+1)c}\}$ corresponds to the cth element of the whole set.

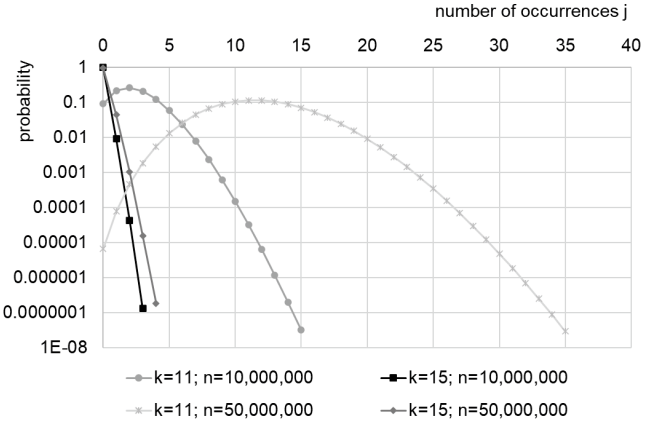
Thus, given Eq. 1 and the probability that a string s of length k appears j times in t_1 (or equivalently in t_2), we can define the average number of occurrences of a k-mer in t_1, and consequently how many times one k-mer is repeated in a text, as

$$\mathrm{E}[|s \cap S_{t_1}^k|] = \sum_{j=0}^{\infty} j \cdot \Pr(|s \cap S_t^k| = j). \quad (2)$$

There are several efficient ways and implementations to calculate Eq. 1 [22]. Since only the average number is needed here, $\mathrm{E}[|s \cap S_{t_1}^k|]$ can also be obtained directly through an approximated formula that is $(l_1 - k + 1)\eta_k$, where η_k is the probability of the string given the the underlying generative model [23]. For a string over a a-letter alphabet with equiprobable characters and Bernoullian model, $\eta_k = a^{-k}$. Note that the approximated formula assumes string independence and non-clumpability (i.e., when strings overlap), but such assumption is robust for large values of k and l_1. Instead, Eq. 1 and Eq. 2 are more generic because they can use the Markovian assumption as well as clumpability through a character switch [22]. As an example, illustrated in Figure 2, using a 4-letter alphabet, $k = 11$, and a 10,000,000 equiprobable bases genome, the mass probability is centered at $j = 2$ and the average number of occurrences following Eq. 2 is 2.38, which matches the simplified equation; if we increase the genome length to 50,000,000 the center of mass becomes 11 and the average number of occurrences is 11.92, and again both equations match.

Let us now consider the two texts t_1 and t_2 of length l_1 and l_2, respectively, and their k-mer sets $S_{t_1}^k$ and $S_{t_2}^k$, with $|S_{t_1}^k| = (l_1 - k + 1)$ and $|S_{t_2}^k| = (l_2 - k + 1)$. We can use Eq. 2 or the approximated formula to estimate the expected number of occurrences of a string $s \in t_1$ within the text t_2, i.e., $\mathrm{E}[|s \cap S_{t_2}^k|]$, as well as the vice versa. Since the total number

Fig. 2. Probability distribution of j occurrences for k-mer strings within texts of length n (alphabet size of 4) using the equiprobable Bernoullian model.



of k-mers in t_1 is $|S_{t_1}^k| = (l_1 - k + 1)$, the expected number of common k-mers with t_2 could be $(l_1 - k + 1) \cdot \mathrm{E}[|s \cap S_{t_2}^k|]$, but this does not consider string multiplicity in t_1. If we consider multiplicity, we need to find the average frequency for distinct k-mers within $S_{t_1}^k$, which we define as $|\hat{S}_{t_1}^k|$. Since all k-mers in t_1 are found at least once, the probability of each of these string to occur zero times, i.e., $\Pr(|s \in S_{t_1}^k| = 0)$, is in fact the probability to occur exactly one time. Thus, the probability of occurring j times for any k-mer $s \in t_1$ is the probability of occurring $j - 1$ times. Hence, we can formalize $|\hat{S}_{t_1}^k|$ as

$$|\hat{S}_{t_1}^k| = \sum_{j=1}^{\infty} \frac{\Pr(|s \in S_{t_1}^k| = (j-1))}{j+1}. \quad (3)$$

For ease of reading and notation, let us rename $\mathrm{E}[|s \cap S_t^k|]$ as μ_t^k. The number of k-mer strings of t_1 found in t_2, i.e., $|S_{2_1}^k|$, will be then $|\hat{S}_{t_1}^k| \cdot \mu_{t_2}^k$, and equivalently $|S_{1_2}^k| = |\hat{S}_{t_2}^k| \cdot \mu_{t_1}^k$.

We now have all components to calculate $|S_{1,2}^k|$ as

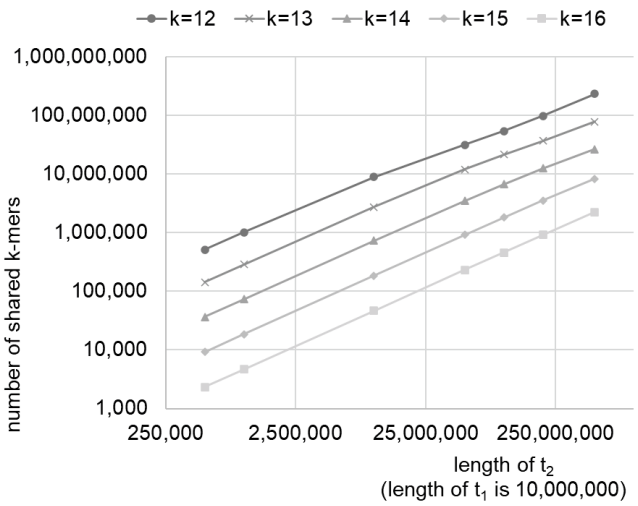
$$|S_{1,2}^k| = |S_{2_1}^k| + |S_{1_2}^k|$$
$$|S_{1,2}^k| = |\hat{S}_{t_1}^k| \cdot \mu_{t_2}^k + |\hat{S}_{t_2}^k| \cdot \mu_{t_1}^k. \quad (4)$$

In Figure 3 we show how $|S_{1,2}^k|$ change by varying text lengths and k-mer lengths. The number of shared k-mers increases when the text length increases, and decreases when k increases; we will show that the complexity of the second phase also decreases inversely to k,

1) String elongation and comparison.: The second phase elongates the k-mers in common between t_1 and t_2, i.e., $S_{1,2}^k$ into $S_{1_2}^e$ and $S_{2_1}^e$, then compares all elongated strings, finding the LCS, and yielding the final $S_{1,2}^e$ as a solution to the ALCS-k problem.

The determination of the LCS for two elongated strings –whose starting positions are known, and thus are already aligned– can be categorized into a base case A, and combination cases B, C, and D. Figure 4 gives a graphical illustration of the basic elongation type A, with the combined B, C, and

Fig. 3. Cardinality of the set of shared k-mers over two texts, increasing k and the length of the second text.



D cases (assuming the LCS is longer than k). The A and C case can be divided into two sub-configurations depending on the elongation direction, but they are basically the same if the elongation direction, or the two sequences, are switched. Case D can result in more than one LCS candidate from the two elongated strings, and thus the resulting set has to be cross-compared to remove substrings (either with a quadratic brute force or using a more efficient procedure such as the suffix-based LCS search).

Let us make some examples for the different categories. Suppose we have the following common k-mer space for $S^{18}_{1,2}$:

text	position	string
t_1	010333999	CAGCATATTTTCTTTTAA
t_2	000004065	CAGCATATTTTCTTTTAA
t_1	010333400	AGCATATTTTCTTTTAAA
t_2	000004066	AGCATATTTTCTTTTAAA
t_1	010333401	GCATATTTTCTTTTAAAA
t_2	000004067	GCATATTTTCTTTTAAAA
t_1	026257228	CATATTTTCTTTTAAAAT
t_2	000004068	CATATTTTCTTTTAAAAT
t_1	026257229	ATATTTTCTTTTAAAATT
t_2	000004069	ATATTTTCTTTTAAAATT
t_1	026257230	TATTTTCTTTTAAAATTA
t_2	000004070	TATTTTCTTTTAAAATTA

After elongation, we obtain the string 010333999 CAGCATATTTTCTTTTAAAA (length of 20) from t_1, and the string 000004065 CAGCATATTTTCTTTTAAAATTA (length of 23) from t_2. The t_2 substring is more elongated to the right, and we retain the one elongated from t_1, thus this corresponds to case A.1. Case A.2 would be instead happening with the following k-mer space:

text	position	string
t_2	000444789	TTTTTAAAAGAAAAGGGG
t_1	000000602	TTTTTAAAAGAAAAGGGG
t_2	000444790	TTTTAAAAGAAAAGGGGG
t_1	000000918	TTTTAAAAGAAAAGGGGG
t_2	000444791	TTTAAAAGAAAAGGGGGG
t_1	000000919	TTTAAAAGAAAAGGGGGG

Here, after elongation, we would get 000444789 TTTTTAAAAGAAAAGGGGGG (length of 20) from t_2, and 000000918 TTTTAAAAGAAAAGGGGGG (length of 19) from t_1. The t_2 substring has extra characters to the left, and we choose the one from t_1.

For case C we can make the following example:

text	position	string
t_1	000015790	TGAAAAAAATTTTTTTTC
t_1	000015791	GAAAAAAATTTTTTTTCC
t_1	000015792	AAAAAAATTTTTTTTCCG
t_1	000015793	AAAAAATTTTTTTTCCGT
t_2	000010000	CTGAAAAAAATTTTTTTT
t_2	000010001	TGAAAAAAATTTTTTTTC
t_2	000010002	GAAAAAAATTTTTTTTCC
t_2	000010003	AAAAAAATTTTTTTTCCG

From t_1 we obtain the right-elongated string TGAAAAAAATTTTTTTTCCGT at position 000015790, while for t_2 we obtain the left-elongated string CTGAAAAAAATTTTTTTTCCG at position 000010000.

Case D is a combination of the previous ones, but each overlap can be treated separately, updating the LCS at each step. Case D is actually biologically plausible and not expected to be a rare case. For instance, a mobile element in a bacterium, such as a plasmid, could carry a resistance gene and transfer it to other bacteria.

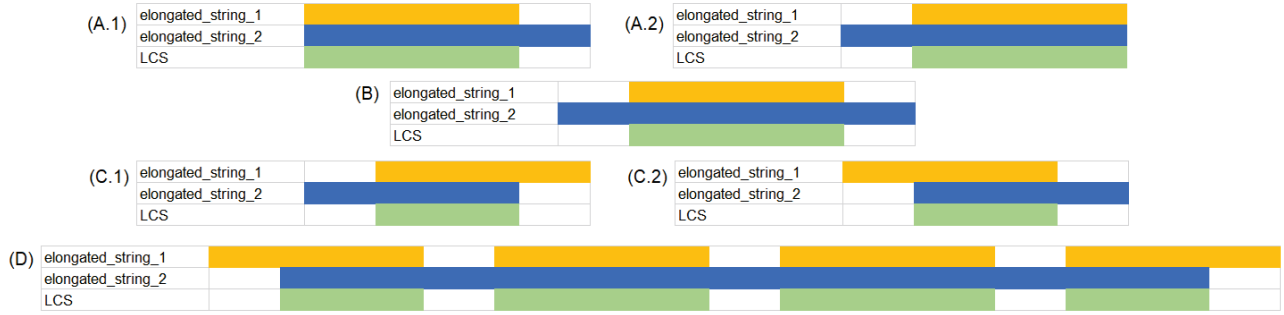
More formally, let be $s^e_{1_2} \in S^e_{1_2}$ and $s^e_{2_1} \in S^e_{2_1}$ two strings of the elongated sets from t_1 and t_2. They can be identical, with length $l_{s^e_{1_2}} = l_{s^e_{2_1}} =: l_{\mathrm{m}}$, or different. If they are different, they will share a substring whose length is between k and l_{m}. In a brute force determination of the LCS, all substrings longer than k for a current elongated unmatched string are calculated, compared with the other elongated string, and the longest is retained. The procedure requires at most $c^k_{l_m} = \sum_{y=k+1}^{l_{\mathrm{m}}}(l_{\mathrm{m}} - y + 1)$ comparisons, i.e., $\frac{(l_{\mathrm{m}}-k)\cdot(l_{\mathrm{m}}-k+1)}{2}$ that is $\mathcal{O}((l_{\mathrm{m}}-k)^2)$. However, the procedure can be stopped as soon as one of the substrings matches both texts. In addition, one can use a more efficient way to perform the search, e.g., via a suffix tree. Yet, for our objective, we show that even with a brute force search, the average case is not quadratic, and reduces to a fraction of the number of common k-mers. Since we are looking for the ALCS-k set, the $c^k_{l_m}$ comparisons must be performed for all elements in $S^e_{1_2}$ and $S^e_{2_1}$.

We can write the total number of comparisons needed to identify the ALCS-k as:

$$c^k_{ALCS} = c^k_{l_m} \cdot (|S^e_{1_2}| + |S^e_{2_1}|). \tag{5}$$

We will demonstrate that actually the quadratic term $c^k_{l_m}$ reduces to a constant multiplied by the number of common k-mers, i.e., $|S^k_{1,2}|$, which also decreases when k increases.

Fig. 4. String elongation and LCS selection, with breakdown of the basic case A and the combined cases B, C, and D.



2) Derivation of time complexity for phase two.: We derive the complexity for the generic alphabet of size a, and then we provide the estimate of the constant for the 4-letter alphabet of genomes. Let $s_1, s_2 \in S^k_{1_2}$ be two strings of t_1 found at least once in t_2. They will be elongated into a string s_3 of length $k+1$ if s_1 starts at p and s_2 starts at $p+1$ position. Let us suppose that there are two other strings equal to s_1 and s_2 in t_2, i.e., $s_4, s_5 \in S^k_{2_1}$, with starting positions at q and $q+1$. The elongated string s_6 will be identical to s_3. The probability of the elongation match to happen is $1/a$ in both texts (using the equiprobable assumption). From Eq. 4, we have estimated the total number of common k-mers $|S^k_{1,2}|$. Let us consider now only one component and its elongated set, e.g., $|S^k_{1_2}|$ and $|S^e_{1_2}|$. Each time, by increasing k of one unit and adding a random character to the right of a k-mer shared by both texts, the probability that a match occurs and that two elongated strings are identical will be always $1/a$. Thus, the number of shared $(k+1)$-mers will be $1/a$ of the number of shared k-mers. In the next step, i.e., the $k+2$ elongated set will contain $1/a$ of the prior one, and at this point the number of shared $(k+1)$-mers will be reduced to $\frac{1}{a}\cdot|S^k_{1_2}| - \frac{1}{a^2}\cdot|S^k_{1_2}| = |S^k_{1_2}|\cdot(\frac{a}{a^2}-\frac{1}{a^2}) = |S^k_{1_2}|\cdot\frac{a-1}{a^2}$. We can continue iteratively, and for $k+n$ elongations, where $(k+n) \leq \max(l_{t_1}, l_{t_2})$, we obtain:

$$
\begin{aligned}
|S^{k+1}_{1_2}| &= (a-1)\cdot|S^k_{1_2}|\cdot\frac{1}{a^2}\\
|S^{k+2}_{1_2}| &= (a-1)\cdot|S^k_{1_2}|\cdot\frac{1}{a^3}\\
&\cdots\\
|S^{k+n}_{1_2}| &= (a-1)\cdot|S^k_{1_2}|\cdot\frac{1}{a^{n+1}}.
\end{aligned}
\tag{6}
$$

For example, if $n=3$, the number of strings that have at least $k+3$ length is $|S^k_{1_2}|\cdot\frac{1}{a^3}$, and thus the total number of strings that include exact $k+1$, $k+2$ lengths, and at least $k+3$, is $|S^k_{1_2}|\cdot((a-1)/a^2+(a-1)/a^3+1/a^3) = (a(a-1)/a^3+(a-1)/a^3+1/a^3 = |S^k_{1_2}|\cdot(a^2-a+a-1+1)/a^3 = |S^k_{1_2}|\cdot 1/a$.

Eq. 6 can be rewritten as:

$$
\begin{aligned}
|S^{k+1}_{1_2}| &= \frac{a-1}{a}\cdot|S^k_{1_2}|\cdot\frac{1}{a}\\
|S^{k+2}_{1_2}| &= \frac{a-1}{a}\cdot|S^k_{1_2}|\cdot\frac{1}{a^2}\\
&\cdots\\
|S^{k+n}_{1_2}| &= \frac{a-1}{a}\cdot|S^k_{1_2}|\cdot\frac{1}{a^n}.
\end{aligned}
\tag{7}
$$

The cardinality of the fully elongated set, i.e., $|S^e_{1_2}|$ is the sum of all sets above. This sum behaves akin to a geometric series of $1/a$ ratio that converge to $1/(1-\frac{1}{a})$. In our case the first (zero) term is missing, and we can write:

$$
\begin{aligned}
|S^e_{1_2}| &= \frac{a-1}{a}\cdot|S^k_{1_2}|\cdot\sum_{k=1}^{n}\frac{1}{a^k}\\
|S^e_{1_2}| &= \frac{a-1}{a}\cdot|S^k_{1_2}|\cdot(\frac{1}{1-\frac{1}{a}}-1)\\
|S^e_{1_2}| &= \frac{a-1}{a}\cdot|S^k_{1_2}|\cdot(\frac{a}{(a-1)}-1)\\
|S^e_{1_2}| &= \frac{1}{a}\cdot|S^k_{1_2}|.
\end{aligned}
\tag{8}
$$

For the 4-letter alphabet, $|S^e_{1_2}| = 0.25\cdot|S^k_{1_2}|$.

Now we can derive c^k_{ALCS} by considering the sum $c^k_{l_m} = \frac{(l_m-k)\cdot(l_m-k+1)}{2}$, and writing

$$
\begin{aligned}
c^k_{l_m}\cdot S^e_{1_2} &= \frac{(k+1-k)\cdot(k+1-k+1)}{2}\cdot\frac{1}{a}\cdot\frac{(a-1)}{a}\cdot|S^k_{1_2}|\\
&+\frac{(k+2-k)\cdot(k+2-k+2)}{2}\cdot\frac{1}{a^2}\cdot\frac{(a-1)}{a}\cdot|S^k_{1_2}|\\
&+\frac{(k+3-k)\cdot(k+3-k+3)}{2}\cdot\frac{1}{a^2}\cdot\frac{(a-1)}{a}\cdot|S^k_{1_2}|\\
&\cdots\\
&= \frac{(a-1)}{a}\cdot|S^k_{1_2}|\cdot\sum_{i=1}^{n}\frac{1}{a^i}\frac{(i+i^2)}{2}.
\end{aligned}
\tag{9}
$$

Eq. 9 can be rewritten as $\frac{(a-1)}{a}\cdot|S^k_{1_2}|\cdot\frac{1}{2}\cdot(\sum_{i=1}^{n} i\frac{1}{a^i} + \sum_{i=1}^{n} i^2\frac{1}{a^i})$. Since $\sum_{i=1}^{n} i\cdot x^i = \frac{x}{(1-x)^2}$, then $\sum_{i=1}^{n} i\frac{1}{a^i} = \frac{a}{(a-1)^2}$. Also, $\sum_{i=1}^{n} i^2\cdot x^i = \frac{x(1+x)}{(1-x)^3}$, thus $\sum_{i=1}^{n} i^2\frac{1}{a^i} = \frac{a(a+1)}{(a-1)^3}$. The final formula for $c^k_{l_m}\cdot S^e_{1_2}$ then becomes

$$c^k_{l_m} \cdot S^e_{1_2} = \frac{(a-1)}{a} \cdot |S^k_{1_2}| \cdot \frac{1}{2} \cdot (\frac{a}{(a-1)^2} + \frac{a(a+1)}{(a-1)^3})$$

$$c^k_{l_m} \cdot S^e_{1_2} = |S^k_{1_2}| \cdot \frac{a}{(a-1)^2}. \quad (10)$$

Let us look in detail now at the time complexity. The positional sorting of $S^k_{1,2}$ is $\mathcal{O}(|S^k_{1,2}|\log(|S^k_{1,2}|))$ since it orders both $S^k_{1_2}$ and $S^k_{2_1}$, i.e., $\mathcal{O}(|S^k_{1_2}|\log(|S^k_{1_2}|) + |S^k_{2_1}|\log(|S^k_{2_1}|)$. In the average case, this step is the superset of all the subsequent ones, because they all include elongated strings, which are a fraction of the set of common k-mers. The string elongation step takes on average $\mathcal{O}(\frac{1}{a}(|S^k_{1_2}| + |S^k_{2_1}|)$, the elongated substring creation, matching, and elimination takes each $\mathcal{O}(\frac{a}{(a-1)^2}(|S^k_{1_2}| + |S^k_{2_1}|)$, while the elongated substring sorting takes $\mathcal{O}(\frac{a}{(a-1)^2}(|S^k_{1_2}| + |S^k_{2_1}|)\log(\frac{a}{(a-1)^2}(|S^k_{1_2}| + |S^k_{2_1}|))$. The space complexity is the original input $S^k_{1,2}$ with the addition of $S^e_{1_2}$ and $S^e_{2_1}$ along with their substrings, thus $\mathcal{O}(\frac{1}{a}(|S^k_{1_2}| + |S^k_{2_1}|)$ and $\mathcal{O}(\frac{a}{(a-1)^2}(|S^k_{1_2}| + |S^k_{2_1}|)$.

3) Implementation.: The stage one and stage two procedures were implemented using C++, and then in Perl by another coder for validation. The first phase reads input genomes in FASTA format, and uses standard i/o libraries and primitive (string, integer) types to write the k-mer spectrum into files, along with their positions. The k-mer files are then ordered alphabetically using an external disk sort, and then merged through a third script, retaining common k-mers and positions. The second phase reads the file of common k-mers, performs positional sorting, and finds the $ALCS$-k set by the elongation, substring ordering/filtering/elimination, and LCS calculation procedures. The C++ and the Perl algorithms differ slightly in the second phase, since the C++ elongates the common k-mers and creates their substrings on the fly, while the Perl version performs elongation first, writes to file, and then performs the substring creation and elimination. The C++ code is available under the MIT license at: https://github.com/DataIntellSystLab/ALCS-k. Binaries for Mac, Windows and Linux/UNIX are provided.

D. Experimental Setup

We selected a variety of organisms from different realms – viruses, bacteria, and animals– with genome lengths spanning tens of thousands to hundreds of millions, and different evolutionary relationships. Table I lists the organisms, their genome sizes and the GenBank accession numbers.

For instance, *H. Sapiens* and *P. troglodytes* (chimpanzee) genomes are ∼98% similar, bacteria can share several genes and mobile elements, some viruses can be found integrated in host genomes, while on the contrary SARS-CoV-2 and HIV-1 have very different gene contents. The variety of the selected genomes should therefore guarantee a comprehensive characterization of the algorithm's efficiency, especially in relation to the time complexity of the second stage.

All genome pairs were analyzed with both the C++ and Perl implementations. As a comparison with state-of-the-art tools, we used MUMmer v.4.0 [16]. Tests were performed on an Intel i7 machine at 2.6 GHz with 16 GB RAM, except when we compared our tool with MUMmer, which can be RAM-intensive for large genomes, where we used the University of Florida's HiPerGator3 'bigmem' allocation with Intel Xeon E7 machines at 2.0 Ghz with 1TB RAM. As our implementation is serial, MUMmer was also run with 1 single thread, printing all non-unique maximal exact matches, i.e., `mummer -maxmatch -threads 1 -qthreads 1 -n -s ref.fasta query.fasta`. Run time (as wall time unless otherwise specified) and memory usage were measured using the UNIX/Linux command usr/bin/time.

Organism	Phylum, family	Abbr.	L	Accession no.
HIV-1	Artverviricota, Retroviridae	hiv	9.1k	NC_001802.1
SARS-CoV-2	Pisuviricota, Coronaviridae	covid	30k	NC_045512.2
Acanthamoeba polyphaga	Nucleocyto-viricota, Mimiviridae	mimi	1.2m	HQ336222.2
Citrobacter freundii	Proteobacteria, Enterobacteri-aceae	citro	5.1m	CP056256.1
Sorangium cellulosum	Proteobacteria, Myxococcales	sora	13m	NC_010162.1
Caenorhabditis elegans (Chr.1)	Nematoda, Rhabditidae	caeno	15m	NC_003279.8
Pan troglodytes (Chr.1)	Chordata, Hominidae	chimp	224.2	NC_036879.1
Homo sapiens (Chr.1)	Chordata, Hominidae	homo	249m	NC_000001.11

TABLE I
ORGANISMS USED IN THE EXPERIMENTAL SETUP. L = GENOME LENGTH

III. RESULTS

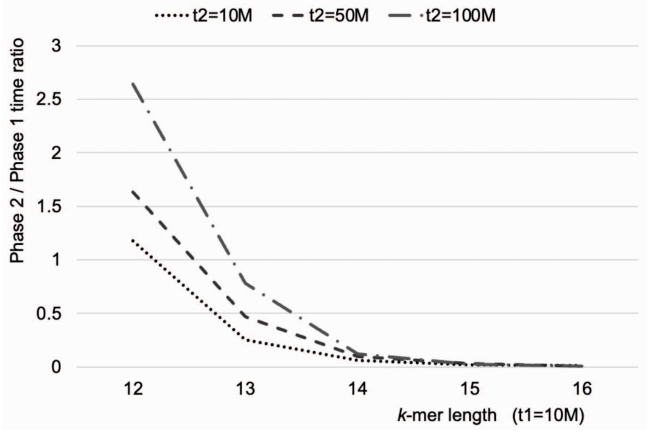
The C++ and Perl implementations yielded identical ALCS-k solutions on all jobs that completed successfully. The program outputs also matched the maximum matches' (non-unique) positions reported by MUMmer. In terms of run times, the C++ code was 1-5x (interquartile range) faster than the Perl, but the Perl implementation outputs also the exact counts of the elongated strings, since the elongation step is decoupled from the LCS selection step. For validation of the theoretical complexity, we thus report the run times only from Perl, while for comparison with MUMmer, we report the C++ ones.

We first empirically verified the theoretical complexity of phase 2 on simulated genomic data (4-letter alphabet), using the Bernoullian model. Table II reports the run times for different text combinations (10 million to 100 million character lengths) and values of k (12 to 16). The linear correlation between the run times and theoretical complexity was $\rho^2 = 0.96$ across all trials. Only for low values of k and longer text lengths (e.g., $k = 12, l_{max} = 100$M), the phase 2 actual complexity and run times deviate from the average case, approaching the worst case limit, since the number of

l_{t_1}, l_{t_2}	k	$\lvert S_{1_2}^k \rvert$	$\lvert S_{2_1}^k \rvert$	theor. compl.	Runtime (s)
10M, 10M	12	4,495,658	4,493,116	143,485,779	114
10M, 10M	13	1,383,709	1,382,706	41,486,622	25
10M, 10M	14	364,645	364,573	10,138,510	6
10M, 10M	15	91,805	91,821	2,345,362	2
10M, 10M	16	21,656	21,660	501,954	1
10M, 50M	12	9,490,871	22,468,950	545,110,250	521
10M, 50M	13	5,248,506	6,911,641	197,194,259	151
10M, 50M	14	1,692,054	1,819,771	53,353,646	31
10M, 50M	15	448,672	457,045	12,753,417	9
10M, 50M	16	107,908	108,343	2,791,063	2
10M, 100M	12	9,973,381	44,939,254	965,595,799	>3,600
10M, 100M	13	7,745,144	13,817,010	360,204,486	394
10M, 100M	14	3,094,411	3,633,306	105,808,265	62
10M, 100M	15	875,220	911,015	26,146,863	14
10M, 100M	16	214,302	216,223	5,799,681	3

TABLE II
EXPERIMENTAL VALIDATION OF PHASE 2 TIME COMPLEXITY USING SIMULATED GENOMES.

Fig. 5. Run time ratio (simulated genomes) between phase 1 and phase 2, varying text (10, 50, and 100 million) and k-mer (12 to 16) lengths.



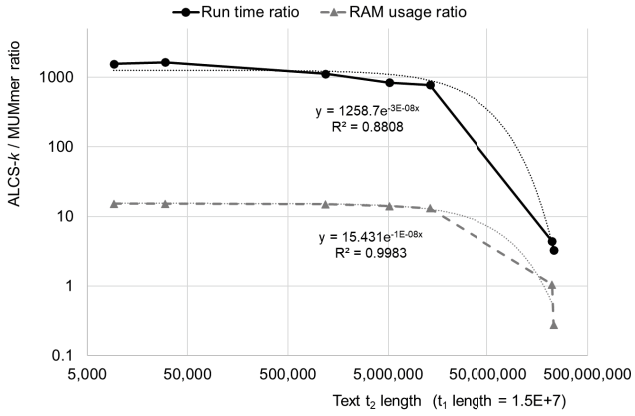
strings in common increases by several orders of magnitude, concomitantly to the expected difference between the LCS length and k, and the number of substrings to be calculated.

Second, we compared the ratio between the complexity of phase 1 and phase 2, using again simulated genomes, verifying that the actual run times would match the theoretical complexity ratio. Figure 5 shows the run time ratios by varying text (10, 50, and 100 million nucleotide bases) and k-mer (12 to 16) lengths. The graph clearly indicates that only for $k = 12$ the second phase approaches the worst case and takes more time than the first phase (i.e., the ratio is lager than 1).

Third, we run the algorithms on the real datasets presented in Table I. Table III provides, for each genome pair, the theoretical and actual number of shared k-mers ($k = 12$), along with real time complexity for phase 2 (based on actual k-mers), and the run times for both phase 1 and 2. The actual number of shared k-mers slightly deviates from the theoretical one ($\rho^2 = 0.61$) because the nucleotide frequencies are not equiprobable and the Bernoullian model does not necessarily hold –we have shown in a prior work that the Markovian model is more accurate [21]. Likewise, the correlation between expected complexity and run times ($\rho^2 = 0.71$) is slightly lower than what observed in simulations. In $5/25 = 20\%$ of the experiments, the phase 2 run time was higher than that of phase 1 (only *Acanthamoeba polyphaga* and *Citrobacter freundii*), but the ratio was never higher than 1.5. We repeated analyses for $k = 20$, and in no cases the run time of phase 2 surpassed that of phase 1, with an average ratio of 0.01.

Fourth, we compared run times of our C++ implementation with MUMmer, using the real genomes. We used *Caenorhabditis elegans* (Chr.1) as the reference (15 million), and all other genomes as queries, from 9 thousand base pairs of HIV to 234.5 million base pairs of the *Homo sapiens* (Chr.1). On the shorter query genomes, MUMmer was over 1000x faster, and used 10x less RAM, but both the run time and RAM ratios rapidly decreased as the genome size increased. With the *Homo sapiens* (Chr.1) genome, our tool had a peak usage of 37 GB of RAM vs. 135 GB used by MUMmer, i.e., $\sim 1/4$ of the RAM, and the run time ratio was reduced to 3x (Figure 6). Since our method is heavily disk-based (all sort and merge procedures are not performed in memory), higher run times are expected, yet it is able to quickly catch up on MUMmer . Of note, the genomes pairs compared are highly divergent, which is the ideal case for our algorithm.

Fig. 6. Run time ratio and RAM usage ration between our algorithm and MUMmer, on a reference genome length of 15 million base pairs, varying query length from 9 thousand to 234.5 million base pairs.



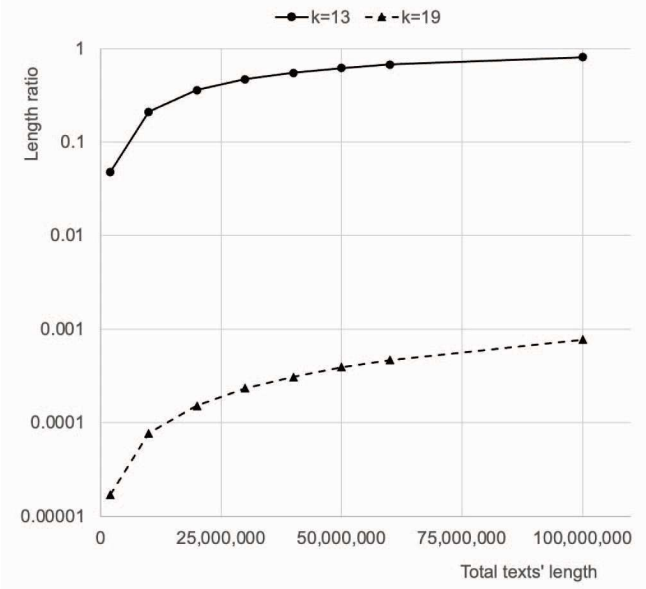
Fifth, we used the first stage and the string elongation of the second stage to mark the original texts' character positions that would be included in the ALCS-k search. For instance, a suffix tree could be constructed only considering these marked portions of the two texts, with reduction in memory footprint and run time. We simulated genomes between 1 million and 50 million base pairs each, and evaluated $k = 13$ and $k = 19$. Figure 7 shows how the procedure reduces the lengths, after filtering and elongation, depending on the value of k and the original lengths. The graphs show that k tremendously affects the reduction, and complements what we showed in Figure 3.

Genome pairs	l_{t_1}	l_{t_2}	$\lvert S^k_{1,2}\rvert$ (theor.)	$\lvert S^k_{1,2}\rvert$ (actual)	Real comp. P2	Runtime (s) P1	Runtime (s) P2
hiv–covid	9E+3	3E+4	33	64	400	1	1
hiv–mimi	9E+3	1E+6	1,271	2,283	20,947	5	1
hiv–citro	9E+3	5E+6	5,205	4,721	46,147	22	0
hiv–sora	9E+3	1E+7	12,089	3,856	37,199	59	0
hiv–caeno	9E+3	2E+7	13,688	16,947	184,666	59	1
hiv–chimp	9E+3	2E+8	130,211	323,855	4,402,102	844	22
hiv–homo	9E+3	2E+8	135,265	348,073	4,753,388	1,094	25
covid–mimi	3E+4	1E+6	4,138	11,251	118,110	5	0
covid–citro	3E+4	5E+6	16,947	4,983	48,948	23	1
covid–sora	3E+4	1E+7	39,362	8,504	87,462	60	0
covid–caeno	3E+4	2E+7	44,566	80,892	987,777	70	1
covid–chimp	3E+4	2E+8	423,875	1,019,691	14,818,135	1,042	35
covid–homo	3E+4	2E+8	440,329	1,062,552	15,479,525	1,089	35
mimi–citro	1E+6	5E+6	657,565	685,275	9,493,123	27	10
mimi–sora	1E+6	1E+7	1,524,562	162,174	2,056,320	65	9
mimi–caeno	1E+6	2E+7	1,725,364	4,559,530	70,752,695	95	146
mimi–chimp	1E+6	2E+8	16,226,540	33,508,960	582,482,745	1,049	1,588
mimi–homo	1E+6	2E+8	16,854,861	34,715,475	604,547,328	1,102	1,710
citro–sora	5E+6	1E+7	6,180,891	6,102,061	95,901,841	84	119
citro–caeno	5E+6	2E+7	6,984,838	6,738,615	106,280,217	93	116
citro–chimp	5E+6	2E+8	63,184,636	59,606,500	1,060,435,451	1,401	1,325
citro–homo	5E+6	2E+8	65,610,051	61,887,060	1,103,147,272	1,468	1,100
sora–caeno	1E+7	2E+7	15,867,593	5,275,650	82,021,390	134	79
sora–chimp	1E+7	2E+8	132,538,334	37,827,767	658,055,947	1,117	967
sora–homo	1E+7	2E+8	137,529,180	45,124,692	785,318,909	1,304	369

TABLE III

EVALUATION OF THE TWO-PHASE ALCS-k APPROACH ON REAL DATASETS. FOR EACH PAIR OF GENOMES, THE THEORETICAL AND ACTUAL NUMBER OF SHARED k-MERS ($k = 12$) IS REPORTED, ALONG WITH RUN TIMES FOR BOTH PHASE 1 AND 2 (P1, P2), AND REAL TIME COMPLEXITY FOR PHASE 2 (BASED ON ACTUAL k-MERS).

Fig. 7. Reduction of genome content and length that would need to be considered for ALCS-k search using the first stage and string elongation procedure..



Finally, to examine a worst-case scenario for our tool, we run it on two highly redundant bacterial databases of mobile genetic elements (MGEs), namely ICEberg (https://bioinfo-mml.sjtu.edu.cn/ICEberg2/index.php) and ACLAME (http://aclame.ulb.ac.be/), containing bacterial integrative and conjugative elements, and prokaryotic MGEs, such as phage genomes, plasmids and transposons. ICEberg contains 552 sequences (28.9 MB), while ACLAME counts 125,190 sequences (119.9 MB). We focused on genes longer than 1,500 bases. As expected, the run time of the second stage surpassed that of the first stage: the algorithm completed in 5 hours with a RAM peak of 173 GB. Instead, MUMmer –which – completed the job in about a minute, with a RAM peak of 18 GB.The final ALCS-1500 set included 84 unique sequences, replicated over 225 positions (74 in ICEberg and 151 in ACLAME).

IV. CONCLUSIONS

We presented a two-stage solution to the ALCS-k problem for two texts based on sort-merge of their k-mer spectra and elongation of common k-mers, providing characterization of computational complexity. Our two-stage approach is of practical utility for long, highly divergent genomes as an alternative to or pre-processing step for suffix-based approaches, given the widespread availability of efficient k-mer tools (both disk-based and in-memory), and a low average complexity footprint of our second stage even by using brute force. In our experiments, we showed that for sufficiently large and highly divergent genomes, our tool uses less RAM than MUMmer

(although MUMmer can be set up to switch on disk-based processing) and has potentially shorter run time.

One limitation of the implementation we provide is that it does not make use of code optimization or efficient data structures. On the one hand, the code was written to validate the theoretical complexity; on the other hand, the software should be usable in real-world applications and use cases. Nonetheless, the first stage involves k-mer parsing, counting and sorting, with additional indexing of positions; several efficient software and library exist that can be used for the purpose. For this reason, we organized our code into modules for which the first stage and the second stage can be run separately. Our novel algorithms implemented in the second stage –given its average complexity that is loglinear in the fraction of the number of shared k-mers– is usable in real-world applications, and for divergent genomes it can be advantageous with respect to tools like MUMmer. In the future, we foresee opportunity to develop ad hoc data structures that can further minimize time/space complexity of the second stage to better handle highly similar genomes, e.g., a suffix-based structure on the subset of elongated k-mers. For instance, MUMmer could be applied to two input genomes where only the elongated strings calculated in phase two are considered, by masking all other positions with unallowable characters.

ACKNOWLEDGMENT

We thank Roberto di Castro, MSc and Luciano Prosperi, MSc for their kind feedback on the formulae and the code.

AVAILABILITY OF DATA AND MATERIALS

Source code, compilation and binaries are available at https://github.com/DataIntellSystLab/ALCS-k. Data used in this work are publicly available–NCBI reference sequences are indicated in Table I.

FUNDING

This work was in part supported by US grants NIH NIAID R01AI141810 (CB, MP, SM), R01AI145552 (MP), and NSF SCH 2013998 (CB, MP).

COMPETING INTERESTS

The authors declare that they have no competing interests.

REFERENCES

[1] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*, 2000, pp. 39–48.

[2] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *J. ACM*, vol. 21, no. 1, p. 168–173, jan 1974. [Online]. Available: https://doi.org/10.1145/321796.321811

[3] R. Bhowmick, M. I. Sadek Bhuiyan, M. Sabir Hossain, M. K. Hossen, and A. Sadee Tanim, "An approach for improving complexity of longest common subsequence problems using queue and divide-and-conquer method," in *2019 1st International Conference on Advances in Science, Engineering and Robotics Technology (ICASERT)*, 2019, pp. 1–5.

[4] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, no. 3, p. 249–260, sep 1995. [Online]. Available: https://doi.org/10.1007/BF01206331

[5] P. Charalampopoulos, T. Kociumaka, S. P. Pissis, and J. Radoszewski, "Faster Algorithms for Longest Common Substring," in *29th Annual European Symposium on Algorithms (ESA 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), P. Mutzel, R. Pagh, and G. Herman, Eds., vol. 204. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 30:1–30:17. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/14611

[6] S. Kurtz, "Reducing the space requirement of suffix trees," *Softw. Pract. Exper.*, vol. 29, no. 13, p. 1149–1171, nov 1999. [Online]. Available: https://doi.org/10.1002/(SICI)1097-024X(199911)29:13\%3C1149::AID-SPE274\%3E3.0.CO;2-O

[7] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," in *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '90. USA: Society for Industrial and Applied Mathematics, 1990, p. 319–327.

[8] P. Ferragina and G. Manzini, "Indexing compressed text," *J. ACM*, vol. 52, no. 4, p. 552–581, jul 2005. [Online]. Available: https://doi.org/10.1145/1082036.1082039

[9] T. Gagie, "Moni can find k-mems," 2022.

[10] L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter, "When indexing equals compression: Experiments with compressing suffix arrays and applications," *ACM Trans. Algorithms*, vol. 2, no. 4, p. 611–639, oct 2006. [Online]. Available: https://doi.org/10.1145/1198513.1198521

[11] M. Comin and M. Farreras, "Efficient parallel construction of suffix trees for genomes larger than main memory," in *Proceedings of the 20th European MPI Users' Group Meeting*, ser. EuroMPI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 211–216. [Online]. Available: https://doi.org/10.1145/2488551.2488579

[12] E. Mansour, A. Allam, S. Skiadopoulos, and P. Kalnis, "Era: Efficient serial and parallel suffix tree construction for very long strings," *Proc. VLDB Endow.*, vol. 5, no. 1, p. 49–60, sep 2011. [Online]. Available: https://doi.org/10.14778/2047485.2047490

[13] B. Phoophakdee and M. J. Zaki, in *Genome-Scale Disk-Based Suffix Tree Indexing*, ser. SIGMOD '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 833–844. [Online]. Available: https://doi.org/10.1145/1247480.1247572

[14] M. Barsky, U. Stege, and A. Thomo, "Suffix trees for inputs larger than main memory," *Inf. Syst.*, vol. 36, no. 3, p. 644–654, may 2011. [Online]. Available: https://doi.org/10.1016/j.is.2010.11.001

[15] J. Shun and G. E. Blelloch, "A simple parallel cartesian tree algorithm and its application to parallel suffix tree construction," *ACM Trans. Parallel Comput.*, vol. 1, no. 1, oct 2014. [Online]. Available: https://doi.org/10.1145/2661653

[16] G. Marçais, A. L. Delcher, A. M. Phillippy, R. Coston, S. L. Salzberg, and A. Zimin, "Mummer4: A fast and versatile genome alignment system," *PLOS Computational Biology*, vol. 14, no. 1, pp. 1–14, 01 2018. [Online]. Available: https://doi.org/10.1371/journal.pcbi.1005944

[17] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Commun. ACM*, vol. 20, no. 5, p. 350–353, may 1977. [Online]. Available: https://doi.org/10.1145/359581.359603

[18] A. Apostolico, "Improving the worst-case performance of the hunt-szymanski strategy for the longest common subsequence of two strings," *Information Processing Letters*, vol. 23, no. 2, pp. 63–69, 1986. [Online]. Available: https://www.sciencedirect.com/science/article/pii/002001908690044X

[19] V. Chvátal and D. Sankoff, "Longest common subsequences of two random sequences," *Journal of Applied Probability*, vol. 12, no. 2, p. 306–315, 1975.

[20] E. Rivals and S. Rahmann, "Combinatorics of periods in strings," *Journal of Combinatorial Theory, Series A*, vol. 104, no. 1, pp. 95–113, 2003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0097316503001237

[21] M. Prosperi, L. Prosperi, R. Gray, and M. Salemi, "On counting the frequency distribution of string motifs in molecular sequences," *International Journal of Biomathematics*, vol. 5, p. 1250055, 2012.

[22] M. Prosperi, S. Marini, and C. Boucher, "Fast and exact quantification of motif occurrences in biological sequences," *BMC Bioinformatics*, vol. 22, no. 1, p. 445, Sep. 2021. [Online]. Available: https://doi.org/10.1186/s12859-021-04355-6

[23] S. Robin, J.-J. Daudin, H. Richard, M.-F. Sagot, and S. Schbath, "Occurrence probability of structured motifs in random sequences," *Journal of Computational Biology*, vol. 9, no. 6, pp. 761–773, 2002, pMID: 12614545. [Online]. Available: https://doi.org/10.1089/10665270260518254