Fast and Simple Algorithms for Computing both LCS_k and LCS_{k+}

Filip Pavetić[1], Ivan Katanić[2], Gustav Matula[2], Goran Žužić[3], and Mile Šikić[2]

[1] Google Switzerland GmbH, Zürich, Switzerland
fpavetic@google.com
[2] Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia
{ivan.katanic, gustav.matula, mile.sikic}@fer.hr
[3] Carnegie Mellon University, Pittsburgh, USA
gzuzic@cs.cmu.edu

Abstract. Longest Common Subsequence (LCS) deals with the problem of measuring similarity of two strings. While this problem has been analyzed for decades, the recent interest stems from a practical observation that considering single characters is often too simplistic. Therefore, recent works introduce the variants of LCS based on shared substrings of length exactly or at least k (LCS_k and LCS_{k+} respectively). The main drawback of the state-of-the-art algorithms for computing LCS_k and LCS_{k+} is that they work well only in a limited setting: they either solve the average case well while being suboptimal in the pathological situations or they achieve a good worst-case performance, but fail to exploit the input data properties to speed up the computation. Furthermore, these algorithms are based on non-trivial data structures which is not ideal from a practitioner's point of view. We present a single algorithm to compute both LCS_k and LCS_{k+} which outperforms the state-of-the art algorithms in terms of runtime complexity and requires only basic data structures. In addition, we implement an algorithm to reconstruct the solution which offers significant improvement in terms of memory consumption. Our empirical validation shows that we save several orders of magnitude of memory on human genome data. The C++ implementation of our algorithms is made available at: https://github.com/google/fast-simple-lcsk.

Keywords: longest common subsequence, string similarity, efficient dynamic programming, bioinformatics, memory optimization

1 Introduction

Measuring the similarity of strings is one of the fundamental problems in computer science. It is very useful in many real-world applications such as DNA sequence comparison [5], differential file analysis and plagiarism detection [7]. One of the most popular techniques for efficient measurement of string similarity is the Longest Common Subsequence (LCS) [11,12,3].

LCS and extensions. Recently, there has been some critique of LCS being an oversimplified way to measure string similarity as it does not distinguish well between the sequences which consist mainly of consecutive characters and the ones which do not [4,16]. To overcome this limitation, extensions of LCS to more general variants have been proposed (see Example 1). In particular, Benson et al. [5,4] suggested LCS_k, which computes the similarity by counting the number of non-overlapping substrings of length k contained in both strings. Another extension was given by Pavetić et al. [14]: the LCS_{k++} computes similarity by summing the lengths of non-overlapping substrings of length **at least** k contained in both strings. These variants

Filip Pavetić, Ivan Katanić, Gustav Matula, Goran Žužić, Mile Šikić: Fast and Simple Algorithms for Computing both LCS_k and LCS_{k+}, pp. 50–62.
Proceedings of PSC 2018, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06484-9 © Czech Technical University in Prague, Czech Republic

have been applied in bioinformatics [15]. Later it has been renamed as $LCS_{\geq k}$ by Benson et al. [4] and as (used in this paper) LCS_{k+} by Ueki et al. [16][1].

Example 1 (Values of LCS_k and LCS_{k+} for various string pairs).

- $LCS_3(ABCBA, ABCBA) = 1$ (ABC) or (BCB) or (CBA)
- $LCS_{3+}(ABCBA, ABCBA) = 5$ $(ABCBA)$
- $LCS_2(ABXXXCDE, ABYYYCDE) = 2$ (AB, CD) or (AB, DE)
- $LCS_{2+}(ABXXXCDE, ABYYYCDE) = 5$ (AB, CDE)
- $LCS_1(AAA, AA) = LCS(AAA, AA) = 2$ (A, A)
- $LCS_{1+}(AAA, AA) = LCS(AAA, AA) = 2$ (A, A)

State of the art. For LCS_k and LCS_{k+} to be useful in practice, we need to be able to compute them efficiently. State-of-the-art algorithms are often parametrized by the total number of matching k-length substring pairs between the input strings, denoted by r. The observation that r is often limited in real-world data can be used to speed up the computation. Existing algorithms usually specialize for the situations when r is either low or high. Deorowicz and Grabowski [8] proposed several algorithms for efficient computation of LCS_k. Most notably, their *Sparse* algorithm allows both the computation of LCS_k and its reconstruction in $\mathcal{O}(m + n + r \log l)$ time and $\mathcal{O}(r)$ memory complexity, where l is the length of the optimal solution and m, n are the lengths of the two input strings. In their approach they adapt the Hunt-Szymanski [11] paradigm in a way that requires them to use a persistent red-black tree. Pavetić et al. [14,13] proposed an algorithm based on the Fenwick tree data structure [9] to efficiently compute LCS_{k+} in $\mathcal{O}(m + n + r \log r)$. Ueki et al. [16] proposed an algorithm which achieves a better worst-case complexity than Pavetić and Žužić [13] when $r \sim mn$, but a worse performance in the situations when r is small. The complexity of their algorithm is $\mathcal{O}(mn)$ and does not depend on r.

Improving the current state. The algorithms to compute LCS_k and LCS_{k+} are either of unsatisfactory runtime complexity or they rely on using complex data structures. Implementing these complex algorithms is extremely time consuming for a practitioner who usually has to make an experiment-based decision about which similarity measure is even useful for their cause. Therefore, it is valuable to have a simple and fast algorithm to compute both LCS_k and LCS_{k+}. Our contributions in this paper are:

- We propose an algorithm to compute LCS_k which achieves a favorable runtime complexity when compared to the previously known algorithms. The runtime complexity of the algorithm is $\mathcal{O}(\min(r \log l, r + ml))$, where l is the length of the optimal solution. (Section 3)
- We show that the same algorithm can be easily extended to compute LCS_{k+}. This unifies the solutions for both of these problems. (Section 4)
- We propose a heuristic to reduce memory needed to reconstruct the solution. Experiments on the human genome demonstrate that it reduces the memory usage by several orders of magnitude. (Section 5)
- Our algorithms do not rely on complex data structures such as Fenwick or a persistent red-black tree.

[1] Different authors used different names, however the definitions are the same.

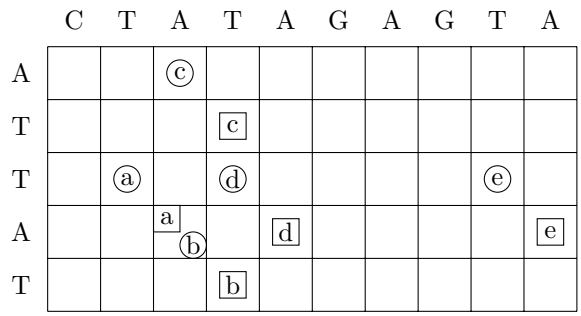

Figure 1. The Figure shows the start and end points of the match pairs produced by the two strings. In this example strings $A = ATTAT$ and $B = CTATAGAGTA$ construct exactly five match pairs for $k = 2$, denoted with a to e. Start points of the pairs are represented by circles and their end points are represented by squares. The following holds: "c precedes e", while the following does not hold: "a precedes b", "c precedes d" (Definition 5). Note that a start point of one match pair can share coordinates with the end point of another: e. g. end point of a and start point of b.

- To speed up future research on the topic, we made the implementation of our algorithms available at `https://github.com/google/fast-simple-lcsk`. As far as we know, this is the first widely accessible implementation of algorithms to compute LCS_k and LCS_{k+}.

2 Preliminaries

This section contains the definitions useful throughout the rest of the paper. Even though we inherit some of the definitions from other sources, we state them here for the sake of completeness. The inputs to all the algorithms are strings A of length m and B of length n, both over alphabet Σ. Without loss of generality we can assume that $m \leq n$. We use $X[i : j)$ to denote the substring of X, starting at (inclusive) index i and ending at (exclusive) index j. Using that notation it holds that $A[0 : m) = A$ and $B[0 : n) = B$.

Definition 2 (The LCS_k problem [4]). *Given two strings A and B of length m and n, respectively, and an integer $k \geq 1$, we say that C is a common subsequence in* exactly *k length substrings of A and B, if there exist $i_1, \ldots, i_t$ and $j_1, \ldots, j_t$ such that $A[i_s : i_s + k) = B[j_s : j_s + k) = C[p_s : p_s + k)$ for $1 \leq s \leq t$, and $i_s + k \leq i_{s+1}, j_s + k \leq j_{s+1}$ and $p_{s+1} = p_s + k$ for $1 \leq s < t, p_1 = 0$ and $|C| = p_t + k$. The longest common subsequence in* exactly *k length substrings (LCS_k) equals to maximum possible t, such that the mentioned conditions are met.*

Definition 3 (The LCS_{k+} problem [14,4,16]). *Given two strings A and B of length m and n, respectively, and an integer $k \geq 1$, we say that C is a common subsequence in* at least *k length substrings of A and B, if there exist $i_1, \ldots, i_t$, $j_1, \ldots, j_t$ and $l_1, \ldots, l_t$ such that $A[i_s : i_s + l_s) = B[j_s : j_s + l_s) = C[p_s : p_s + l_s)$ and $l_s \geq k$ for $1 \leq s \leq t$, and $i_s + l_s \leq i_{s+1}, j_s + l_s \leq j_{s+1}$ and $p_{s+1} = p_s + l_s$ for $1 \leq s < t, p_1 = 0$ and $|C| = p_t + l_t$. The longest common subsequence in* at least *k length substrings (LCS_{k+}) equals to maximum possible sum $\sum_{i=1}^{t} l_i$, such that the mentioned conditions are met.*

We state the recurrence relation to compute $LCS_k(i, j)$=$LCS_k(A[0 : i), B[0, j))$ for two strings A and B, given in [5]:

$$LCS_k(i,j) = \max \begin{cases} LCS_k(i-1,j) & \text{if } i \geq 1 \\ LCS_k(i,j-1) & \text{if } j \geq 1 \\ LCS_k(i-k,j-k)+1 & \text{if } A[i-k:i) = B[j-k:j) \end{cases} \quad (1)$$

The choice to add 1 or k in the last line of Equation 1 is arbitrary since it influences the final result only by a constant factor. We choose to add +1 to be consistent with prior definitions of LCS_k. When expanding the relation to LCS_{k+}, we need to adjust it as shown in [14,16]:

$$LCS_{k+}(i,j) = \max \begin{cases} LCS_{k+}(i-1,j) & i \geq 1 \\ LCS_{k+}(i,j-1) & j \geq 1 \\ \max\limits_{\substack{k \leq k' \leq min(i,j) \\ A[i-k':i) = B[j-k':j)}} LCS_{k+}(i-k',j-k')+k' & \end{cases} \quad (2)$$

Definition 4 (Match pair [5]). *Given the strings A, B and integer $k \geq 1$ we say that at (i,j) there is a **match pair** if $A[i:i+k) = B[j:j+k)$. (i,j) is also called the **start point** or the **start** of the match pair. $(i+k-1, j+k-1)$ is called the **end point** or the **end** of the match pair.*

Definition 5 (Precedence of match pairs). *Let $P=(i_P, j_P)$ and $G=(i_G, j_G)$ be match pairs. Then G **precedes** P if $i_G + k \leq i_P$ and $j_G + k \leq j_P$. In other words, G precedes P if the end of G is on the upper left side of the start of P in the dynamic programming table (see Figure 1).*

2.1 Efficient algorithms for computing LCS

The known efficient algorithms are based on the observation that in order to compute LCS via a classic dynamic programming approach[2], it is not always necessary to fill out the entire matrix. If it happens that many entries repeat in the cases when two strings have only a few pairs of matching characters, it is possible to design a structure which stores the matrix in a compressed form. We sketch the main ideas behind these techniques since we later use them as building blocks.

The algorithm by Hunt and Szymanski [11,6] traverses only the matching character pairs of the two strings in row-major order. The main idea is to maintain an array M such that M_d holds the minimum j such that there is some already processed row i for which $LCS(i,j) = d$. For simplicity we define $M_0 = 0$ and $M_d = \infty$ if no such j exists. In simpler terms, M is a compressed representation of the dynamic programming table, storing only the boundaries of same-value intervals. This is obviously useful when a row contains many repeated values. Note that M is non-decreasing. For every point (i,j) corresponding to matching character pairs in row i, let us find the biggest d such that $M_d < j$. Then there must exist a common subsequence of length d ending with (i',j') where $i' < i$ and $j' < j$. Such a subsequence can be extended by (i,j), so we know that after processing all the points in the current row we will have $M_{d+1} \leq j$. It will suffice to find a d such that $M_d < j \leq M_{d+1}$ (easily done using binary search), and set M_{d+1} to $\min(M_{d+1}, j)$ (since we're extending a subsequence

[2] This is usually done by filling out a matrix based on the relation which we get after we set $k = 1$ in either of the Equation 1 or Equation 2.

of length d ending at column M_d into a subsequence of length $d+1$ ending at column j). We note that the order in which the points of a row are processed matters: they should be ordered descending by column, so that the updates to M with the results of a new row happen effectively at the same time (otherwise queries are influenced by previous updates from the same row, which leads to incorrect results).

Hunt's algorithm was modified by Kuo and Cross [12] by replacing the binary search for each point in a row with a linear scan of M together with all the points in a row. Specifically, as we process the points of the current row, we also maintain an index d into M which we increment until $M_{d+1} \geq j$ for the current point (i, j). The increments of d amortize over the length of M, so the total complexity is $\mathcal{O}(r_i + l)$, where r_i is the number of points in row i and l is the length of the LCS. When $r_i \ll l$ this algorithm is performing worse than Hunt's variant (which would have a runtime complexity of $\mathcal{O}(r_i \log r_i)$). However, it does becomes a significant improvement as r_i approaches l.

3 Algorithm to compute LCS_k

In this section we show how to compute LCS_k efficiently. The main observation is that processing start and end points of the match pairs independently allows us to directly re-use the techniques for the efficient computation of LCS. Additionally, we dynamically adapt the computation between situations where the number of match pairs r is low as well as high, in order to secure a good worst-case performance. We achieve a runtime complexity of $\mathcal{O}(m + n + r + \min(r \log l, r + ml))$ and the memory complexity of $\mathcal{O}(l + m + n)$.

3.1 Decoupling the starts and ends of the match pairs

Another way to formulate the computation of LCS_k is to view it as a problem of finding the longest chain of match pairs. This formulation is an extension of the one previously used in the LCS literature [10]. Given a match pair P, it is possible to compute $LCS_k(P)$ using the following relation:

$$LCS_k(P) = \begin{cases} 1 & \text{if no match pair precedes P} \\ \max_G LCS_k(G) + 1 & \text{over all G preceding P} \end{cases} \tag{3}$$

In other words, we are looking for the longest chain of match pairs such that a pair which occurs earlier in the chain precedes the pairs which come later. The chain ending with a match pair P can be constructed in two ways: (i) P is added to some preceding shorter chain ending with G or (ii) a new chain containing only P is started.

Now that we have reformulated the relation for LCS_k, we show how to compute it efficiently. First we take a step back to the description of Hunt's algorithm for LCS. There we mentioned that the order in which the points within a row are processed matters since reads and updates to the helper array M happen interchangeably. If we allow two traversals of the points, the first one can only read and the second one can only update M. Note that this does not have much effect on the result of the LCS computation, but it only removes the restriction on the order in which we have to process the points within a row. The decoupling of the reads and the updates of

M is an idea which we use to generalize the algorithm to compute LCS_k. Namely, if we decouple the start and end points of all the match pairs and process them in row-major order, at every row we can: 1) do the reads of M for all the start points and then 2) update M with values of all the end points. This entire algorithm is summarized in Algorithm 1.

Algorithm 1 Computing LCS_k by decoupling start and end points

```
1: for 0 ≤ i < m do
2:     for all x = (i, j) ∈ StartPointsForRow(i) do
3:         P ← MatchPair(x)
4:         LCS_{k,start}(P) ← d s.t. M_d < j ≤ M_{d+1}
5:     end for
6:     for all x = (i, j) ∈ EndPointsForRow(i) do
7:         P ← MatchPair(x)
8:         LCS_{k,end}(P) ← LCS_{k,start}(P) + 1
9:         M_{LCS_{k,end}(P)} ← min(M_{LCS_{k,end}(P)}, j)
10:    end for
11: end for
12: return max_P LCS_{k,end}(P)
```

In the Algorithm 1 we use several quantities: 1) $LCS_{k,start}(P)$ stores the value read from M at the start point of match pair P, 2) $LCS_{k,end}(P)$ stores the value of LCS_k at the end point of P and 3) $MatchPair(x)$ retrieves a match pair for which x is a start or an end point. Lines 2-5 of the algorithm can be implemented in two different ways: 1) following Hunt's paradigm and performing binary search over the M array to do the reads and 2) following Kuo's paradigm and doing all the reads in one linear pass over array M. Since we can estimate the number of operations needed for both variants, at each row i we dynamically choose between the two options. Doing this picks up the benefits of both approaches and makes our algorithm work efficiently in both sparse and dense rows. $StartPointsForRow(i)$ and $EndPointsForRow(i)$ can be computed in $\mathcal{O}(n+m+r)$ time in multiple ways: 1) by using a suffix array based approach proposed by Deorowicz and Grabowski [8] or 2) hash the k-mers of B and create a hash table mapping to the indexes (if it happens that Σ^k is small enough to fit a computer word). Querying that table with k-mers from A trivially gives the start/end points for a wanted row. For more details see Appendix A.1.

3.2 Complexity

Generating all the match pairs takes $\mathcal{O}(m+n+r)$ time. Computing the update for row i takes $\mathcal{O}(min(r_i \ log \ l, r_i + l))$ time. Summing over all the rows implies that

$$\sum_{r=0}^{m-1} \min(r_i \ log \ l, r_i + l) \leq \mathcal{O}(\min(r \ log \ l, r + ml)) \tag{4}$$

Theorem 6. *The presented algorithm computes LCS_k with the runtime complexity of $\mathcal{O}(m+n+r+\min(r \ log \ l, r+ml))$. The memory complexity is $\mathcal{O}(l+m+n)$.*

Proof. The runtime complexity directly follows from adding up the complexities for generating the match pairs with the right side of Equation 4. Regarding memory consumption, we need to $\mathcal{O}(l)$ memory for array M, and $\mathcal{O}(m+n)$ memory to generate

the start/end points of the match pairs (same for both described approaches to generate them). This does not takes into account the memory needed for the reconstruction of the solution, which requires $\mathcal{O}(r)$ memory.

4 Algorithm to compute LCS_{k+}

In this section, we show how to modify the algorithm for LCS_k to compute LCS_{k+}. Similar to LCS_k, we look at the LCS_{k+} computation as finding chains of match pairs where consecutive ones precede or continue (see Definition 7) one another. This makes it possible to achieve the runtime complexity of $\mathcal{O}(m+n+r+\min(r(\log l+k), r+ml))$ and the memory complexity of $\mathcal{O}(l+m+n)$.

Definition 7 (Continuation of match pairs). *Let $P = (i_P, j_P)$ and $G = (i_G, j_G)$ be k-match pairs. Then P **continues** G if $i_P = i_G + 1$ and $j_P = j_G + 1$. P is only one down-right position from G, see Figure 1.*

We reproduce the relation for computing LCS_{k+} over match pairs from [14]:

$$dp(P) = \max \begin{cases} k \\ dp(G) + 1 & \text{if P is a continuation of G} \\ \max_G dp(G) + k & \text{over all G preceding P} \end{cases} \tag{5}$$

Note that this relation computes the actual number of characters in the LCS_{k+}, whereas the corresponding relation for LCS_k computes the number of blocks of length k. This difference requires us to reconsider the compressed representation M and make important changes. Instead of the array M, we introduce a new array N, defined as follows. Lets assume that we have finished processing the start and end points in row $i-1$. Let N_d denote the minimum j such that $dp(P) \geq d$ for some match pair P with end point (i', j). Furthermore, we let $N_0 = 0$ and $N_d = \infty$ when no such j exists. Note that by using '$\geq$' instead of '$=$' we make N non-decreasing. Indeed, since $LCS_{k+}(i-1, N_{d+1}) \geq d+1 \geq d$, N_d cannot be greater than N_{d+1}.

Using N to compute the maximum over preceding pairs in Equation 5 turns out to be the same as for LCS_k. However, updating N needs an adjustment. If we look at the end (i, j) of match pair P and suppose that we have calculated the corresponding $dp(P)$, the first temptation is to simply set $N_{dp(P)} \leftarrow \min(N_{dp(P)}, j)$. In order to maintain the non-decreasing property of N, we must ensure that $N_d \leq j$ for all $d \leq dp(P)$. However, it happens that we don't really need to update the whole prefix. If $dp(P)$ is computed from a preceding match pair G as $dp(G) + k$, upon processing the start point $(i-k+1, j-k+1)$ of P, we have $N_{dp(G)} < j-k+1 \leq j$. Since for any d, N_d may only decrease as we move from row to row, this will also hold when we reach the end point of P (in row i). Therefore, $N_{dp(G)} = N_{dp(P)-k}$ will already be smaller than or equal to j. The same holds for all indices less than $dp(G)$, as N is kept non-decreasing so we only need to set $N_{dp(P)-s} \leftarrow \min(N_{dp(P)-s}, j)$ for $s \in 0, .., k-1$. If we encounter $N_{dp(P)-s} = j$ we stop, as further values of N are already smaller than or equal to j. Since the pairs are sorted by column, this bounds the total time spent iterating through N for a single row by $O(l)$. In the case $dp(P) = dp(G)+1$ where P is a continuation of G we have $N_{dp(G)} \leq j-1 < j$, so it is enough to set $N_{dp(P)} \leftarrow \min(N_{dp(P)}, j)$.

i	N_0	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_9	N_{10}	N_{11}	N_{12}	N_{13}	N_{14}
42	0	5	5	5	5	33	43	43	43	43	44	49	49	49	49
45	0	5	5	5	5	31	31	31	31	43	44	49	49	49	49

Figure 2. Example used to highlight the difference in the updates between LCS_k and LCS_{k+}. It shows a typical situation where k entries of N need to be changed. The table shows the state of N after finishing with rows 42 and 45. Note that the content of the strings does not matter, we assume a situation where $k = 4$ and the only match pair in that range is $(42, 28)$ (N does not change between rows 43 and 44). When processing the start point of $P = (42, 28)$ in row 42, we find that $N_4 < 28 \leq N_5$, so we take $d = 4$. This means that using P we can extend a sequence of length 4 into a sequence of length 8. So when we reach row 45, and see the end point $(45, 31)$ of P, we calculate $dp(P) = d + k = 4 + 4 = 8$. Finally, we set N_5 through N_8 to $j = 31$.

Algorithm 2 Algorithm to compute LCS_{k+}

```
1: for 0 ≤ i < m do
2:     for all x = (i, j) ∈ StartPointsForRow(i) do
3:         P ← MatchPair(x)
4:         LCS_{k+,start}(P) ← d s.t. N_d < j ≤ N_{d+1}
5:     end for
6:     for all x = (i, j) ∈ EndPointsForRow(i) do
7:         P ← MatchPair(x)
8:         G ← a match pair s.t. P continues G
9:         LCS_{k+,end}(P) ← max(LCS_{k+,start}(P) + k, LCS_{k+,end}(G) + 1)
10:        for LCS_{k,end}(P) − k < z ≤ LCS_{k,end}(P) do
11:            N_z ← min(N_z, j)
12:        end for
13:    end for
14: end for
15: return max_P LCS_{k,end}(P)
```

We note that lines 2-5 (processing starts of match pairs) are identical for both Algorithm 1 and 2. The differences are in how the ends of the match pairs are processed, where two things happen: (i) continuations of match pairs are handled and (ii) the updates are done as described earlier in the section.

4.1 Complexity

In the sparse case, the time complexities of querying and updating N are $\mathcal{O}(r_i \log l)$ and $\mathcal{O}(kr_i)$ respectively, or $\mathcal{O}(r_i(\log l + k))$ in total. For the dense case we get $\mathcal{O}(r_i + l)$ for both, yielding $\mathcal{O}(\min(r_i(\log l + k), r_i + l))$ as the time complexity of processing a single row. Summing over all the rows, this is bounded by $\mathcal{O}(\min(r(\log l + k), r + ml))$. The runtime complexity of the whole algorithm is presented in Theorem 8.

Theorem 8. *The presented algorithm computes LCS_{k+} with the runtime complexity of $\mathcal{O}(m + n + r + \min(r(\log l + k), r + ml))$. The memory complexity is $\mathcal{O}(l + m + n)$.*

Proof. The analysis of memory complexity is the same as in Theorem 6. For time complexity, we again add up the complexities of generating match pairs and calculating the length of the LCS_{k+}.

Finally, we mention that one can achieve $\mathcal{O}(\log l)$ runtime complexity for querying N by using a more involved data structure (see Appendix A.2). However, we argue that the added complications are not worth it, since we are often in a setting where values

of k range in $\mathcal{O}(\log n)$. A good example of that are the DNA aligners operating on genomes having billions ($\log_2 10^9 \sim 30$) of nucleotides, with typical values of k ranging from 10 to 32 [18]. Additionally, too large values of k result in absence of match pairs, which does not make them useful [14].

5 Notes on the implementation

We implemented the algorithms described in this paper and made the code available at `https://github.com/google/fast-simple-lcsk`. This section briefly describes some details of our implementation which haven't been addressed in the rest of the paper.

The analysis of memory complexity for the reconstruction of the optimal solution in Section 3 shows that maintaining the chains of match pairs causes the biggest part of the memory consumption - if we store the match pairs until the end of the computation to do the reconstruction we need $\mathcal{O}(r)$ memory. For long inputs, this makes it impossible to fit the computation into RAM of a single computer. To reduce the memory requirements we can observe the following: at any moment of the processing, we will have a set of reconstruction paths ending with a match pair contained in the array M - we only need to keep the match pairs on these paths. As soon as there is no reconstruction path going from M to some match pair x, we can delete x. This is implemented as follows: every match pair is reference counted and has a pointer to its predecessor in the reconstruction path. The last match pair in every reconstruction path is pointed to from array M. As soon as M stops pointing to such match pair, its reference count drops to zero and it is deleted. This can further cause that the reference count of its predecessor dropped to zero so that one gets deallocated, etc.[3]

In order to demonstrate savings of the memory consumption on the real world data, we performed simple experiments on different chromosomes from the human genome[4]. We computed LCS_k of several chromosomes with themselves and compared the number of match pairs with the maximum number of match pairs kept in memory during computation for different values of k, in order to get an estimate on their relation in real data. Figure 3 shows that the savings of the memory consumption are significant. In particular, we can see that the described optimization can save 700-1000x of the memory. The savings tend to increase as the number of match pairs increases.

6 Conclusion and future work

We have demonstrated a single and simple algorithm for computing both LCS_k and LCS_{k+} of two strings. The algorithm beats the runtime complexity of the existing approaches and does not require complex data structures. Also, we have demonstrated a heuristic for reducing the memory needed to reconstruct the solution. Experiments on real data demonstrated savings of 700-1000x. Furthermore, we made our implementation widely accessible.

[3] In C++ this can be easily achieved by using std::shared_ptr.
[4] *Homo_sapiens.GRCh38 (Release 88)* obtained from *www.ensembl.org* [17]

k	chromosome	match pairs	max in memory	compression factor
30	1	5 627 330 181	7 802 719	721.20
30	2	5 737 185 065	8 186 269	700.83
30	3	3 575 560 336	6 778 074	527.51
29	1	6 428 219 516	8 090 660	794.52
29	2	6 368 795 382	8 494 144	749.78
29	3	3 971 642 925	7 020 437	565.72
28	1	7 374 448 317	8 399 480	877.96
28	2	7 108 071 229	8 824 093	805.52
28	3	4 440 640 300	7 277 931	610.15
27	1	8 540 954 582	8 732 809	978.03
27	2	8 012 803 096	9 179 547	872.89
27	3	5 014 652 783	7 554 106	663.83
26	1	9 954 502 925	9 092 744	1094.77
26	2	9 100 424 727	9 537 800	954.14
26	3	5 708 852 882	7 849 184	727.31

Figure 3. The comparison of the number of all the match pairs and the number of max match pairs kept in memory at any moment of the computation for the first few chromosomes from the human genome and varying values of k. The ratios of these numbers directly translates to the savings in the memory required for the reconstruction. We note that in our case of computing LCS_k of a chromosome with itself, the number of match pairs is going to be a sum of squares.

As a direction for future research we would like to pose few questions:

- Is it possible to create a link between the memory optimization we described with what is known as dominant points in LCS-related literature (see Appendix A.3)?
- It is not clear that all the match pairs are useful in the search for the optimal sequences. Can we speed up the algorithms by developing rules for discarding some of them, before even starting the computation?

Acknowledgments. The authors would like to thank Maria Brbić and Mario Lučić for the valuable comments on the manuscript.

References

1. A. Apostolico, S. Browne, and C. Guerra: *Fast linear-space computations of longest common subsequences.* Theor. Comput. Sci., 92(1) Jan. 1992, pp. 3–17.
2. A. Apostolico and C. Guerra: *The longest common subsequence problem revisited.* Algorithmica, 2(1) 1987, pp. 315–336.
3. B. S. Baker and R. Giancarlo: *Sparse dynamic programming for longest common subsequence from fragments.* J. Algorithms, 42(2) 2002, pp. 231–254.
4. G. Benson, A. Levy, S. Maimoni, D. Noifeld, and B. R. Shalom: *Lcsk: a refined similarity measure.* Theoretical Computer Science, 638 2016, pp. 11–26.
5. G. Benson, A. Levy, and B. R. Shalom: *Longest common subsequence in k length substrings*, in Proceedings of the 6th International Conference on Similarity Search and Applications - Volume 8199, SISAP 2013, New York, NY, USA, 2013, Springer-Verlag New York, Inc., pp. 257–265.
6. L. Bergroth, H. Hakonen, and T. Raita: *A survey of longest common subsequence algorithms*, in Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00), SPIRE '00, Washington, DC, USA, 2000, IEEE Computer Society, pp. 39–.
7. C. Y. Chen, J. Y. Yeh, and H. R. Ke: *Plagiarism detection using rouge and wordnet.* arXiv preprint arXiv:1003.4065, 2010.

8. S. Deorowicz and S. Grabowski: *Efficient algorithms for the longest common subsequence in k-length substrings.* Inf. Process. Lett., 114(11) Nov. 2014, pp. 634–638.
9. P. M. Fenwick: *A new data structure for cumulative frequency tables.* Software: Practice and Experience, 24(3) 1994, pp. 327–336.
10. C. M. Goeman Heiko: *A new practical linear space algorithm for the longest common subsequence problem.* Kybernetika, 38(1) 2002, pp. 45–66.
11. J. W. Hunt and T. G. Szymanski: *A fast algorithm for computing longest common subsequences.* Commun. ACM, 20(5) May 1977, pp. 350–353.
12. S. Kuo and G. R. Cross: *An improved algorithm to find the length of the longest common subsequence of two strings.* SIGIR Forum, 23(3-4) Apr. 1989, pp. 89–99.
13. F. Pavetić and G. Žužić: *lcskpp.* `https://github.com/fpavetic/lcskpp`, 2014.
14. F. Pavetić, G. Žužić, and M. Šikić: *LCSk++: Practical similarity metric for long strings.* CoRR, abs/1407.2407 2014.
15. I. Sović, M. Šikić, A. Wilm, S. N. Fenlon, S. Chen, and N. Nagarajan: *Fast and sensitive mapping of error-prone nanopore sequencing reads with graphmap.* bioRxiv, 2015.
16. Y. Ueki, Diptarama, M. Kurihara, Y. Matsuoka, K. Narisawa, R. Yoshinaka, H. Bannai, S. Inenaga, and A. Shinohara: *Longest common subsequence in at least k length order-isomorphic substrings*, in SOFSEM 2017: Theory and Practice of Computer Science, 2017, pp. 363–374.
17. A. Yates, W. Akanni, M. R. Amode, D. Barrell, K. Billis, D. Carvalho-Silva, C. Cummins, P. Clapham, S. Fitzgerald, L. Gil, C. G. Girn, L. Gordon, T. Hourlier, S. E. Hunt, S. H. Janacek, N. Johnson, T. Juettemann, S. Keenan, I. Lavidas, F. J. Martin, T. Maurel, W. McLaren, D. N. Murphy, R. Nag, M. Nuhn, A. Parker, M. Patricio, M. Pignatelli, M. Rahtz, H. S. Riat, D. Sheppard, K. Taylor, A. Thormann, A. Vullo, S. P. Wilder, A. Zadissa, E. Birney, J. Harrow, M. Muffato, E. Perry, M. Ruffier, G. Spudich, S. J. Trevanion, F. Cunningham, B. L. Aken, D. R. Zerbino, and P. Flicek: *Ensembl 2016.* Nucleic Acids Research, 44(D1) 2016, p. D710.
18. M. Zaharia, W. J. Bolosky, K. Curtis, A. Fox, D. A. Patterson, S. Shenker, I. Stoica, R. M. Karp, and T. Sittler: *Faster and more accurate sequence alignment with snap.* CoRR, abs/1111.5572 2011.

A Appendix

A.1 Generating the match pairs

Algorithm 1 assumes that it has a list of start and end points already available. Here we address how to obtain it. We offer two approaches which have the same runtime complexity, but different tradeoffs between the simplicity and the assumptions on the input data.

Deorowicz and Grabowski [8] described an algorithm for enumerating all the match pairs by using a suffix array built over the string $B\#A$[5]. The LCP (Longest Common Prefix) table is used to group all the suffixes of that string sharing a prefix of at least k. With careful bookkeeping it is possible to enumerate all the columns j which correspond to match pairs starting in row i. For more details please consult the referenced paper.

We note that in practice we are often in a setting where either the alphabet size Σ is small or useful values of k are small (e. g. DNA has $\Sigma = 4$ and the popular tools for aligning the DNA often set k to a range 10-32 [18]). If it happens that Σ^k is small enough to fit a 64-bit integer, we can easily and cheaply obtain perfect hashing of the k-mers by treating them as k-digit number in base Σ. Having that, we can build a hash table $\mathcal{H}$ mapping from the hashes of all the k-mers of B to the indices of their start position. Enumerating all the match pairs starting in a row i then comes down to hashing $A[i : i + k)$ and looking up all the indices from $\mathcal{H}$. This gives us a simpler algorithm, but still relevant in practice.

Both of the described approaches show how to generate the match pairs in $\mathcal{O}(n+m+r)$ time. We note that Algorithm 1 requires the start and end match points at every row. The start points for row i are obtained directly by generating the match pairs at row i. The end points for row i are obtained by generating the match pairs at row $i-k+1$.

A.2 An alternative to $\mathcal{O}(k)$ updates for LCS_{k+}

The operations we need are querying a single element of the array, and setting all elements in a given prefix $[0..i]$ to the minimum of a given value v and their present value.

One approach is to use a complete binary tree in which the leaves correspond to array elements in order. An internal node then represents an interval of the array. To simplify things, we think of leaves as intervals consisting of a single element. Every node of the tree has an associated value, initially set to ∞. We denote the leaf representing element N_i by $L(i)$. The algorithm is as follows:

- The update for prefix $[0, i]$ with value v is done by setting the values left siblings (if they exist) of nodes along the path from root to $L(i + 1)$ to the minimum of v and their present value (updates are lazily propagated to the leaves).
- The query for N_i is done by finding the minimum of the values along the path from root to the appropriate leaf $L(i)$.

[5] $B\#A$ = string B, concatenated with character '#', concatenated with string A

		a	a	a	a	a	a	a	a
	0	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	0	0	0
a	0	0	②	2	2	2	2	2	2
a	0	0	2	2	2	2	2	2	2
a	0	0	2	2	④	4	4	4	4
a	0	0	2	2	4	4	4	4	4
a	0	0	2	2	4	4	⑥	6	6
a	0	0	2	2	4	4	6	6	6
a	0	0	2	2	4	4	6	6	⑧

		b	b	a	a	d	d	c	c
	0	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	0	0	0
a	0	0	0	0	②	2	2	2	2
b	0	0	0	0	2	2	2	2	2
b	0	0	②	2	2	2	2	2	2
c	0	0	2	2	2	2	2	2	2
c	0	0	2	2	2	2	2	2	④
d	0	0	2	2	2	2	2	2	4
d	0	0	2	2	2	2	④	4	4

Figure 4. The full LCS_k table for two different string pairs, with $k = 2$. The encircled fields are called dominant points. For the purposes of reconstructing the solution, it is sufficient to only keep the match pairs ending at these locations.

Looking at the query and update together, we see that querying for i we will, for every previously applied update on $[0..j]$ such that $i \leq j$, encounter exactly one node that was affected by it, and thus correctly calculate the current value of N_i. Since the depth of the tree is $\mathcal{O}(\log l)$, that is also the time complexity of our operations.

By using this structure, we can reduce the runtime complexity of computing LCS_{k+} to $\mathcal{O}(m + n + r + \min(r \log l, r + ml))$.

A.3 Dominant points

The example strings shown in Figure 4 (left) generate match pairs at almost all the indexes. In this case $r = \Omega(mn)$, which makes the memory required to reconstruct the solution extremely high. Still, looking at the table, we can observe the following: in order to build any of the table entries with value 4, we only need to keep the topmost-leftmost entry with value 2, as opposed to keeping all of them. In fact, the points that we need to keep in order to reconstruct the solution are known in the LCS-related literature as *dominant points*.

Definition 9 (Dominant points [2,1]). *Lets look at a table $T(i, j) = LCS_k(i, j)$ or $T(i, j) = LCS_{k+}(i, j)$. A point (i, j) is then called q-dominant if $T(i, j) = q$ and for any other (i', j') such that $T(i', j') = q$ it holds that either ($i' > i$ and $j' \leq j$) or ($i' \leq i$ and $j' > j$) is true. The dominant points are then the union over q-dominant points over all different q.*