Contents lists available at ScienceDirect

Discrete Applied Mathematics

journal homepage: www.elsevier.com/locate/dam

New tabulation and sparse dynamic programming based techniques for sequence similarity problems

Szymon Grabowski

Lodz University of Technology, Institute of Applied Computer Science, Al. Politechniki 11, 90–924 Łódź, Poland

ARTICLE INFO

Article history:
Received 5 October 2014
Received in revised form 19 July 2015
Accepted 27 October 2015
Available online 2 December 2015

Keywords:
Sequence similarity
Longest common subsequence
Sparse dynamic programming
Tabulation

ABSTRACT

Calculating the length ℓ of a longest common subsequence (LCS) of two strings, A of length n and B of length m, is a classic research topic, with many known worst-case oriented results. We present three algorithms for LCS length calculation with respectively $O(mn \lg\lg n/\lg^2 n)$, $O(mn/\lg^2 n + r)$ and $O(n + r)$ time complexity, where the second one works for $r = o(mn/(\lg n \lg\lg n))$, and the third one for $r = \Theta(mn/\lg^k n)$, for a real constant $1 \le k \le 3$, and $\ell = O(n/(\lg^{k-1} n(\lg\lg n)^2))$, where r is the number of matches in the dynamic programming matrix. We also describe conditions for a given problem sufficient to apply our techniques, with several concrete examples presented, namely the edit distance, the longest common transposition-invariant subsequence (LCTS) and the merged longest common subsequence (MerLCS) problems.

1. Introduction

Measuring the similarity of sequences is an old research topic and many actual measures are known in the string matching literature. One classic example concerns the computation of a longest common subsequence (LCS) in which a subsequence that is common to all sequences and has the maximal possible length is looked for. A simple dynamic programming (DP) solution works in $O(mn)$ time for two sequences of length n and m, respectively, but faster algorithms are known. The LCS problem has many applications in diverse areas, like version control systems, comparison of DNA strings, structural alignment of RNA sequences. Other related problems comprise calculating the edit (Levenshtein) distance between two sequences, the longest common transposition-invariant subsequence, or LCS with constraints in which the longest common subsequence of two sequences must contain, or exclude, some other sequence.

Let us focus first on the LCS problem, for two sequences A and B. It is defined as follows. Given two sequences, $A = a_1 \ldots a_n = A[1 \ldots n]$ and $B = b_1 \ldots b_m = B[1 \ldots m]$, over an alphabet Σ of size σ, find a longest subsequence $\langle a_{i_1}, a_{i_2}, \ldots, a_{i_\ell} \rangle$ of A such that $a_{i_1} = b_{j_1}, a_{i_2} = b_{j_2}, \ldots, a_{i_\ell} = b_{j_\ell}$, where $1 \le i_1 < i_2 < \cdots < i_\ell \le n$ and $1 \le j_1 < j_2 < \cdots < j_\ell \le m$. The found sequence may not be unique. W.l.o.g. we assume $n \ge m$. To avoid uninteresting complications, we also assume that $m = \Omega(\lg^2 n)$. Additionally, we assume that $\sigma = O(m)$. The case of a general alphabet, however, can be handled with standard means, i.e., we can initially map the sequences A and B onto an alphabet of size $\sigma' = O(m)$, in $O(n \lg \sigma')$ time, using a balanced binary search tree. We do not comprise this tentative preprocessing step in further complexity considerations.

Often, a simplified version of the LCS problem is considered, when one is interested in telling only the length of a longest common subsequence (LLCS).

In this paper we present three techniques for finding the LCS length, one (Section 3) based on tabulation and improving the result of Bille and Farach-Colton [3] by a factor of $\lg\lg n$, second (Section 4) combining tabulation and sparse dynamic

E-mail address: sgrabow@kis.p.lodz.pl.

http://dx.doi.org/10.1016/j.dam.2015.10.040
0166-218X/© 2015 Elsevier B.V. All rights reserved.

Table 1
Summary of main results for finding ℓ, the length of a longest common subsequence (LLCS) for two sequences of length m and n, respectively, where $m \le n$, over an integer alphabet of size σ. The number of pairs of symbols shared by the two input sequences is denoted with r, $0 \le r \le mn$. The number of dominant matches is denoted with D, $D \le r$. All the algorithms require (at most) linear extra space in addition to the space for storing the input sequences. "Stand. DP" and "Stand. 4R" are abbreviations for the standard dynamic programming and the standard Four Russians (tabulation) algorithm.

Algorithm	Time complexity	Conditions	Ref.
Stand. DP	$O(mn)$	–	folklore
Stand. 4R	$O(n + mn/\lg^2 n)$	$\sigma = O(1)$ and $m \ge \lg n$	[15]
Sparse DP	$O(n\sigma + D \lg\lg(\min(D, mn/D)))$	–	[8]
BFC	$O(n + mn(\lg\lg n)^2/\lg^2 n)$	$m \ge \lg n \lg\lg n$	[3]
Our-1	$O(mn \lg\lg n/\lg^2 n)$	$m \ge \lg^2 n$	Section 3
Our-2	$O(mn/\lg^2 n + r)$	$m \ge \lg^2 n$ and $r = o(mn/(\lg n \lg\lg n))$	Section 4
Our-3	$O(n + r)$	$m \ge \lg^2 n$ and $r = \Theta(mn/\lg^k n)$, for real $k \in [1, 3]$, and $\ell = O(n/(\lg^{k-1} n(\lg\lg n)^2))$	Section 5

programming and being slightly faster if the number of matches is appropriately limited, and the last (Section 5) improving the previous result if also the LCS length is conveniently limited. In Section 6 we show the conditions necessary to apply these algorithmic techniques. Some other, LCS-related, problems fulfill these conditions, so we immediately obtain new results for these problems as well. The summary of our results for the LLCS-finding problem is presented in Table 1. The other results listed there are briefly discussed in Sections 2 and 3.

Throughout the paper, we assume the word-RAM model of computation with machine word size $w \ge \lg n$. All used logarithms are base 2.

A preliminary version of this article appeared in Proc. PSC 2014 [10].

2. Related work

A standard solution to the LCS problem is based on dynamic programming, and it is to fill a matrix M of size $(n+1) \times (m+1)$, where $n + 1$ is the number of columns, $m + 1$ the number of rows, and each cell value depends on a pair of compared symbols from A and B (that is, only if they match or not), and its (at most) three already computed neighbor cells. Each computed $M[i, j]$ cell, $1 \le i \le n$, $1 \le j \le m$, stores the value of $LLCS(A[1 \dots i], B[1 \dots j])$. A well-known property describes adjacent cells: $M(i, j) - M(i - 1, j) \in \{0, 1\}$ and $M(i, j) - M(i, j - 1) \in \{0, 1\}$ for all valid i, j.

Despite almost 40 years of research, surprisingly little can be said about the worst-case time complexity of LCS. It is known that in the very restrictive model of unconstrained alphabet and comparisons with equal/unequal answers only, the lower bound is $\Omega(mn)$ [21], which is reached by a trivial DP algorithm. If the input alphabet is of constant size, the known lower bound is simply $\Omega(n)$, but if total order between alphabet symbols exists and $\le$-comparisons are allowed, then the lower bound grows to $\Omega(n \lg n)$ [11]. In other words, the gap between the proven lower bounds and the best worst-case algorithm is huge.

A simple idea proposed in 1977 by Hunt and Szymanski [13] has become a milestone in LCS research, and the departure point for theoretically better algorithms (e.g., [8]). The Hunt–Szymanski (HS) algorithm is essentially based on dynamic programming, but it visits only the matching cells of the matrix, typically a small fraction of the entire set of cells. This kind of selective scan over the DP matrix is called *sparse dynamic programming* (SDP). We note that the number of all matches in M, denoted with the symbol r, can be found in $O(n)$ time, and after this (negligible) preprocessing we can decide if the HS approach is promising to given data. More precisely, the HS algorithm works in $O(n + r \lg m)$ or even $O(n + r \lg\lg m)$ time. Note that in the worst case, i.e., for $r = \Theta(mn)$, this complexity is however superquadratic.

The Hunt–Szymanski concept was an inspiration for a number of subsequent algorithms for LCS calculation, and the best of them, the algorithm of Eppstein et al. [8], achieves $O(D \lg\lg(\min(D, mn/D)))$ worst-case time (plus $O(n\sigma)$ preprocessing), where $D \le r$ is the number of so-called dominant matches in M (a match (i, j) is called dominant iff $M[i, j] = M[i-1, j]+1 = M[i, j - 1] + 1$). Note that this complexity is $O(mn)$ for any value of D. A more recent algorithm, by Sakai [19], is an improvement if the alphabet is very small (in particular, constant), as its time complexity is $O(m\sigma + \min(D\sigma, \ell(m-q)) + n)$, where $\ell = LLCS(A, B)$ and $q = LLCS(A[1 \dots m], B)$.

A different approach is to divide the dynamic matrix into small blocks, such that the number of essentially different blocks is small enough to be precomputed before the main processing phase. In this way, the block may be processed in constant time each, making use of a built lookup table (LUT). This "Four Russians" technique was first used to the LCS problem by Masek and Paterson [15], for a constant alphabet, and refined by Bille and Farach-Colton [3] to work with an arbitrary alphabet. The obtained time complexities were $O(mn/\lg^2 n)$ and $O(mn(\lg\lg n)^2/\lg^2 n)$, respectively, with linear space.

A related, but different approach, is to use bit-parallelism to compute several cells of the dynamic programming matrix at a time. There are a few such variants (see [14] and references therein), all of them working in $O(\lceil m/w \rceil n)$ worst-case time, after $O(\sigma \lceil m/w \rceil + m)$-time and $O(\sigma m)$-space preprocessing, where w is the machine word size.

Yet another line of research considers the input sequences in compressed form. There exist such LCS algorithms for RLE-, LZ- and grammar-compressed inputs. We briefly mention two results. Crochemore et al. [4] exploited the LZ78-factorization of the input sequences over a constant alphabet, to achieve $O(hmn/\lg n)$ time, where $h \leq 1$ is the entropy of the inputs. Gawrychowski [9] considered the case of two strings described by SLPs (straight line programs) of total size k, to show a solution computing their edit distance in $O(kn\sqrt{\lg(n/k)})$ time, where n is the sum of their (non-compressed) length.

Some other LCS-related results can be found in the surveys [1,2].

3. LCS in $O(mn\lg\lg n/\lg^2 n)$ time

In this section we modify the technique of Bille and Farach-Colton (BFC) [3, Sect. 4], improving its worst-case time complexity to $O(mn\lg\lg n/\lg^2 n)$, i.e., by a factor of $\lg\lg n$, with linear space. First we present the original BFC idea, and then signal how our algorithm diverts from it. In the presentation, some unimportant details of the BFC solution are changed, to make the description more compatible with our variant.

The dynamic programming matrix $M[0\ldots n, 0\ldots m]$ is divided into rectangular blocks with shared borders, of size $(x_1+1)\times(x_2+1)$, and the matrix is processed in horizontal stripes of x_2 rows. By "shared borders" we mean that e.g. the bottom row of some block being part of its output is also part of the input of the block below. Values inside each block depend on:

(i) x_1 corresponding symbols from sequence A,
(ii) x_2 corresponding symbols from sequence B,
(iii) the top row of the block, which can be encoded differentially in x_1 bits,
(iv) the leftmost column of the block, which can be encoded differentially in x_2 bits.

The output of each block will be found via a lookup table built in a preprocessing stage. The key idea of the BFC technique is alphabet remapping in superblocks of size $y\times y$. W.l.o.g. we assume that both x_1 and x_2 divide y. Consider one superblock of the matrix, corresponding to the two substrings: $A[i'y+1\ldots(i'+1)y]$ and $B[j'y+1\ldots(j'+1)y]$, for some i' and j'. For the substring $B[j'y+1\ldots(j'+1)y]$ its symbols are sorted and $q\leq y$ unique symbols are found. Then, the y symbols are remapped to $\Sigma_{B_{j'}}=\{0\ldots q-1\}$, using a balanced BST. Next, for each symbol from the snippet $A[i'y+1\ldots(i'+1)y]$ we find its encoding in $\Sigma_{B_{j'}}$, or assign q to it if it was not found there. This takes $O(\lg y)$ time per symbol, thus the substrings of A and B associated with the superblock are remapped in $O(y\lg y)$ time. The overall alphabet remapping time for the whole matrix is thus $O((mn\lg y)/y)$.

This remapping technique allows to represent the symbols from the input components (i) and (ii) on $O(\lg\min(y+1,\sigma))$ bits each, rather than $\Theta(\lg\sigma)$ bits. It works because not the actual symbols from A and B are important for LCS computations, but only equality relations between them. To simplify notation, let us assume a large enough alphabet so that $\min(y+1,\sigma)=y+1$.

In this way, the input per block, comprising the components (i)–(iv) listed above, takes $x_1\lg(y+1)+x_2\lg(y+1)+x_1+x_2$ bits, which cannot sum to $\omega(\lg n)$ bits, otherwise the preprocessing time and space for building the LUT handling all possible blocks would be superpolynomial in n. Setting $y=x_1^2$ and $x_1=x_2=\lg n/(6\lg\lg n)$, we obtain $O(mn(\lg\lg n)^2/\lg^2 n)$ overall time, with sublinear LUT space.

Now, we present our idea. Again, the matrix is processed in horizontal stripes of x_2 rows and the alphabet remapping in superblocks of size $y\times y$ is used. The difference concerns the lookup table; instead of one, we build many of them. More precisely, for each (remapped) substring of length x_2 from sequence B we build a lookup table for fast handling of the blocks in one horizontal stripe. Once a stripe is processed, its LUT is discarded to save space. This requires to compute the answers for all possible inputs in components (i), (iii) and (iv) (the component (ii) is fixed for a given stripe). The input thus takes $x_1\lg(y+1)+x_1+x_2=x_1\lg(2(y+1))+x_2$ bits.

The return value associated with each LUT key is the bottom and the right border of a block, in differential form (the lowest cell in the right border and the rightmost cell in the bottom border are the same cell, which is represented twice; once as a difference (0 or 1) to its left neighbor in the bottom border and once as a difference (0 or 1) to its upper neighbor in the right border) and the difference between the values of the bottom right and the top left corner (to know the explicit value of M in the bottom right corner), requiring $x_1+x_2+\lg(\min(x_1,x_2)+1)$ bits in total. Fig. 1 illustrates.

As long as the input and the output of an LUT fit a machine word, i.e., do not exceed w bits, we will process one block in constant time. Still, as in the original BFC algorithm, the LUT building costs also impose a limitation. More precisely, we are going to minimize the total time of remapping the alphabet in all the superblocks, building all $O(m/x_2)$ LUTs and finally processing all the blocks, which is described by the formula:

$$O(m\lg y+(mn\lg y)/y+(m/x_2)2^{x_1\lg(2(y+1))+x_2}x_1x_2+mn/(x_1x_2)),$$

where $2^{x_1\lg(2(y+1))+x_2}$ is the number of all possible LUT inputs and the x_1x_2 multiplier corresponds to the computation time per one LUT cell. The four additive terms of the formula correspond, in order, to: creating the alphabets for all superblocks, performing all the alphabet remappings, building all LUTs and processing all the blocks. Let us set $y=\lg^2 n/2$, $x_1=\lg n/(4\lg\lg n)$ and $x_2=\lg n/4$. In total we obtain $O(mn\lg\lg n/\lg^2 n)$ time with $o(n)$ extra space (for the lookup tables, used one at a time, and alphabet remapping), which improves the Bille and Farach-Colton result by a factor of $\lg\lg n$.

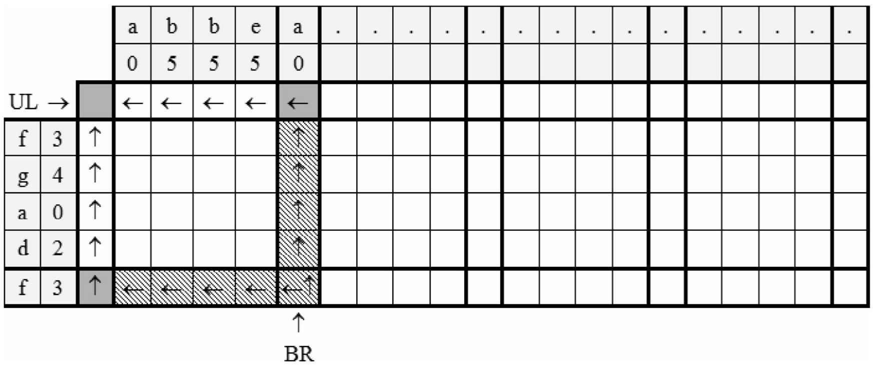

Fig. 1. One horizontal stripe of the DP matrix, with 4 blocks of size 5×5 ($x_1 = x_2 = 4$). The corresponding snippets from sequences A and B are `abbea` and `fgadf`, respectively. These snippets are translated to a new alphabet (the procedure for creating the new alphabet is not shown here) of size 6, where the characters from A are mapped onto the alphabet $\{0, 1, \ldots, 4\}$ and value 5 is used for the characters from B not used in the encoding of the symbols from A belonging to the current superblock (the superblock is not shown here). The LCS values are stored explicitly in the dark shaded cells. The white and dark shaded cells with arrows are part of the input, and their LCS values are encoded differentially, with regard to their left or upper neighbor. The diagonally shaded cells are the output cells, also encoded differentially. The bottom right corner (BR) is stored in three forms: as the difference to its left neighbor (0 or 1), as the difference to its upper neighbor (0 or 1) and the value of UL (upper-left corner) plus the difference between BR and UL. The difference between BR and UL is part of the LUT output for the current block.

The improvement is achieved thanks to using multiple lookup tables (one per horizontal stripe). Formally, we obtain the following theorem.

Theorem 1. *The length of the longest common subsequence (LCS) of two sequences, A, of length n, and B, of length m, where* $n \geq m \geq \lg^2 n$, *both over an integer alphabet, can be computed in* $O(mn \lg\lg n/\lg^2 n)$ *worst-case time. The algorithm needs* $o(n)$ *words of space, apart for the two sequences themselves.*

4. LCS in $o(mn/\lg^2 n + r)$ time (for some r)

In this algorithm we also work in blocks, of size $(b+1) \times (b+1)$, but divide them into two groups: sparse blocks are those which contain at most K matches and dense blocks are those which contain more than K matches. Obviously, we do not count possible matches on the input boundaries of a block.

We observe that knowing the left and top boundaries of a block plus the location of all the matches in the block is enough to compute the remaining (right and bottom) boundaries. This is a nice property as it eliminates the need to (explicitly) access the corresponding substrings of A and B.

The sparse block input will be encoded as:

(i) the top row of the block, represented differentially in b bits,
(ii) the leftmost column of the block, represented differentially in b bits,
(iii) the match locations inside the block, each in $\lg(b^2)$ bits, with $O(K \lg b)$ bits in total.

Each sparse block will be computed in constant time, thanks to an LUT. Dense blocks, on the other hand, will be partitioned into smaller blocks, which in turn will be handled with our algorithm from Section 3. Clearly, we have $b = O(\lg n)$ (otherwise the LUT build costs would be dominating) and $b = \omega(\lg n/\sqrt{\lg\lg n})$ (otherwise this algorithm would never be better than the one from Section 3), which implies that $K = \Theta(\lg n/\lg\lg n)$, with an appropriate constant.

As this algorithm's worst-case time is $\Omega(mn/\lg^2 n)$, it is easy to notice that the preprocessing costs for building required LUTs and alphabet mapping will not dominate. Each dense block is divided into smaller blocks of size $\Theta(\lg n/\lg\lg n) \times \Theta(b)$. Let the fraction of dense blocks in the matrix be denoted as f_d (for example, if half of the $(b+1) \times (b+1)$ blocks in the matrix are dense, then $f_d = 0.5$). The total time complexity (without preprocessing) is then

$$O((1-f_d)mn/b^2 + f_d(mn \lg\lg n/(b \lg n))).$$

The fraction f_d must be $o(1)$, otherwise this algorithm is not better in complexity than the previous one. This also means that $1 - f_d$ may be replaced with 1 in further complexity considerations.

Recall that r is the number of matches in the dynamic programming matrix. We have $f_d = O((r/K)/(mn/b^2)) = O(rb^2 \lg\lg n/(mn \lg n))$. From the $f_d = o(1)$ condition we also obtain that $rb^2 = o(mn \lg n/\lg\lg n)$. If $r = o(mn/(\lg n \lg\lg n))$, then we can safely use the maximum possible value of b, i.e., $b = \Theta(\lg n)$ and obtain the time of $O(mn/\lg^2 n)$. Note that the conditions $r = o(mn/(\lg n \lg\lg n))$ and $f_d = o(1)$ are equivalent, remembering that $K = \Theta(\lg n/\lg\lg n)$.

Unfortunately, in the preprocessing we have to find and encode all matches in all sparse blocks, which requires $O(n + r)$ time. Overall, this leads to the following theorem.

Theorem 2. *The length of the longest common subsequence (LCS) of two sequences, A, of length n, and B, of length m, where $n \geq m \geq \lg^2 n$, both over an integer alphabet, can be computed in $O(mn/\lg^2 n + r)$ worst-case time, assuming $r = o(mn/(\lg n \lg\lg n))$, where r is the number of symbol pairs A_i, B_j such that $A_i = B_j$. The algorithm needs $o(n)$ words of space, apart for the two sequences themselves.*

Considering the presented restriction on r, the achieved complexity is better than the result from the previous section.

On the other hand, it is essential to compare the obtained time complexity with the one from Eppstein et al. algorithm [8]. All we can say about the number of dominant matches D is the $D \leq r$ inequality,[1] so we replace D with r in their complexity formula to obtain $O(r \lg\lg(\min(r, mn/r)))$ in the worst case. Our result is better if $r = \omega(mn/(\lg^2 n \lg\lg\lg n))$ and $r = o(mn)$. Overall, it gives the niche of $r = \omega(mn/(\lg^2 n \lg\lg\lg n))$ and $r = o(mn \lg\lg n/\lg^2 n)$ in which the algorithm presented in this section is competitive.

The alphabet size is yet another constraint. From the comparison to Sakai's algorithm [19] we conclude that our algorithm needs $\sigma = \omega(\lg\lg\lg n)$ to dominate for the case of $r = \omega(mn/(\lg^2 n \lg\lg\lg n))$.

5. LCS in $O(n + r)$ time (sometimes)

The improvement of the first of the two previously presented algorithms (Section 3) over the standard dynamic programming is bounded by a factor related to the $O(\lg\lg n)$ bits needed for a symbol encoding in the block. The second algorithm (Section 4) removes this obstacle if the total number of matches, r, is small enough, since the match positions (in sparse blocks, which dominate) may be used instead of the sequence snippets. The first term in its time complexity of $O(mn/\lg^2 n + r)$ is however related to the differential representation of the top and leftmost boundaries of the block. In this section we present another sparse dynamic programming technique, which, under favorable conditions, removes even the latter constraint.

In the DP matrix for the LCS problem the number of cells in a row (column) which have their value greater than their left (upper) neighbor is up to ℓ, the LCS length. Correspondingly, the number of set bits in a differentially encoded row or column is upper-bounded by ℓ. If ℓ is small, this may allow for a more compact representation of the top row or the leftmost column of a rectangular block, the computation unit in the tabulation-based algorithms.

Assume for a moment that we know the value of ℓ beforehand. We follow the general approach of the algorithm from the previous section, dividing the blocks into sparse and dense ones, but using a different definition. A block of size $(x_1 + 1) \times (x_2 + 1)$ will now be called sparse if it contains not more than K_1 matches and its top row in the differential representation contains not more than K_2 set bits. Otherwise, a block is considered dense. We set $x_2 = \lg n/4$. At the moment we know that $x_1 = \omega(\lg n/\lg\lg n)$, otherwise this algorithm cannot obtain lower time complexity than the one from Section 3. Dense blocks will be partitioned into smaller blocks, of size $(\lg n/(4\lg\lg n)+1) \times (\lg n/4+1)$, and processed in constant time each with the algorithm from Section 3. Now we have to find such maximal x_1 which allows to process sparse blocks in constant time.

A dense block is split into $h = \Theta(x_1 \lg\lg n/\lg n)$ smaller blocks. In other words, processing a dense block is h times more costly than processing a sparse one. We should thus choose such x_1 for which the fraction of dense blocks f_d is $O(1/h)$. The total time complexity (without preprocessing) of $O((1 - f_d)mn/(x_1 \lg n) + f_d(mn \lg\lg n/\lg^2 n))$ becomes then simply $O(mn/(x_1 \lg n))$, and we assume that $h = \omega(1)$ and thus $x_1 = \omega(\lg n/\lg\lg n)$.

The maximal x_1 we can choose depends on two output parameters, r and ℓ. Note that the average number of set bits in a block's top row is upper-bounded by $O(x_1\ell/n)$. Let us assume that $r = mn/\operatorname{polylog}(n)$, or more precisely, $r = \Theta(mn/\lg^k n)$, where $k \geq 1$ is some real constant.

Setting $K_1 = O(rh/(mn/(x_1 \lg n))) = O(x_1^2 \lg\lg n/\lg^k n)$ satisfies the requirement $f_d = O(1/h) = O(\lg n/(x_1 \lg\lg n))$, as far as the match component of the block's input is involved. The other crucial component is however the top row of the block. Setting $K_2 = O((x_1\ell/n)h) = O(x_1^2\ell \lg\lg n/(n \lg n))$ is enough to have $f_d = O(1/h)$ with regard to this constraint.

We however also require that (i) $K_1 \lg(x_1 \lg n) = O(\lg n)$ (with an appropriately small constant), since the (up to) K_1 encoded matches are part of the input of a sparse block, and (ii) $K_2 \lg(x_1/K_2) = O(\lg n)$, again with an appropriately small constant, since the gaps between successive set bits in the top row may be encoded with, e.g., Elias code [20, Sect. 2.4], with less than $2\lg_2 g$ bits spent per gap of size g; the cost is maximized for K_2 equal-sized gaps, from the convexity of the log function.

Putting together the conditions on K_1 and choosing K_1 as large as possible (since it also maximizes x_1), we obtain $x_1^2(\lg\lg n/\lg^k n)\lg(x_1 \lg n) = \Theta(\lg n)$, hence $x_1^2 \lg(x_1 \lg n) = \Theta(\lg^{k+1} n/\lg\lg n)$, and finally $x_1 = \Theta(\lg^{(k+1)/2} n/\sqrt{k})$.

We now use the obtained value of x_1 in the constraints on K_2. Namely, we have $K_2 = O(\ell \lg^k n \lg\lg n/(nk))$ and $K_2 \lg(\lg^{(k+1)/2} n/(K_2\sqrt{k})) = \Theta(\lg n)$. From the latter we have $K_2 = \Theta(\lg n/(k \lg\lg n))$. The former condition entails ℓ, and with the latter condition taken into account we must have $\ell = O(n/(\lg^{k-1} n(\lg\lg n)^2))$.

There is one more thing to notice. The output of each block is its bottom and right boundaries, and while the block's rightmost column of size $\Theta(\lg n)$ poses no problem, one may wonder if this is alike for the bottom row, which may contain

[1] A slightly more precise upper bound on D is $\min(r, m^2)$, but it may matter, in complexity terms, only if $m = o(n)$ (cf. also [19, Th. 1]), which is a less interesting case.

more set bits than the top row of the block. Fortunately, the increase in the number of set bits is obviously not greater than the number of matches in this block, i.e., does not exceed K_2, and the (Elias) encoding of a set bit in the bottom row is not more expensive than the encoding of a match within a block. We thus conclude that handling the possibly greater number of set bits in the bottom row will not compromise the time complexity.

Let us now settle the final time complexity. Using $x_1 = \Theta(\lg^{(k+1)/2} n/\sqrt{k})$ (with a small enough constant) we obtain $O(n + mn\sqrt{k}/\lg^{(k+3)/2} n)$ time, if $r = \Theta(mn/\lg^k n)$, for any real constant $k \geq 1$, and $\ell = O(n/(\lg^{k-1} n(\lg\lg n)^2))$. To this, we need to add the preprocessing costs: $o(n)$ time for the LUT construction and $O(n + r)$ time for encoding the matches. The achieved time complexity dominates over the Eppstein et al. one [8] for $k \leq 3$.

Now we recall the initial assumption that we know the value of ℓ beforehand. We solve this issue with simple means: find first the value of r (in $O(n)$ time) and if $r = \Theta(mn/\lg^k n)$, we assume that $\ell = \Theta(n/(\lg^{k-1} n(\lg\lg n)^2))$ and set the values of x_1, K_1 and K_2 correspondingly. If, after $\Theta(r)$ (with a large enough constant) units of computation, we have not arrived to the solution yet, which means that the fraction of dense blocks is too large, we stop this procedure and switch to another algorithm, e.g. the one from Section 4 (with no loss in complexity terms in this case).

To sum up, we just obtained the following result.

Theorem 3. *The length ℓ of the longest common subsequence (LCS) of two sequences, A, of length n, and B, of length m, where $n \geq m \geq \lg^2 n$, both over an integer alphabet, can be computed in $O(n + r)$ worst-case time, assuming $r = \Theta(mn/\lg^k n)$, for a real constant $1 \leq k \leq 3$, and $\ell = O(n/(\lg^{k-1} n(\lg\lg n)^2))$, where r is the number of symbol pairs A_i, B_j such that $A_i = B_j$. The algorithm needs $o(n)$ words of space, apart for the two sequences themselves.*

We note that the time complexity is better than the one from Theorem 2 for $k > 2$. As a result, the algorithm's niche is for $2 < k \leq 3$.

6. Algorithmic applications

The techniques presented in the three previous sections may be applied to any sequence similarity problem fulfilling certain properties. The conditions are specified in the following lemma.

Lemma 4. *Let Q be a sequence similarity problem returning the length ℓ of a desired subsequence, involving two sequences, A of length n and B of length m, both over a common integer alphabet Σ of size $\sigma = O(m)$. We assume that $1 \leq m \leq n$. Let Q admit a dynamic programming solution in which $M(i,j) - M(i-1,j) \in \{-1, 0, 1\}$, $M(i,j) - M(i,j-1) \in \{-1, 0, 1\}$ for all valid i and j, and $M(i,j)$ depends only on the values of its (at most) three neighbors $M(i-1,j)$, $M(i,j-1)$, $M(i-1,j-1)$, and whether $A_i = B_j$.*

There exists a solution to problem Q with $O(mn\lg\lg n/\lg^2 n)$ worst-case time. There also exists a solution to Q with $O(mn/\lg^2 n + r)$ worst-case time, for $r = o(mn/(\lg n\lg\lg n))$, where r is the number of symbol pairs A_i, B_j such that $A_i = B_j$. Finally, there exists a solution to Q with $O(n + r)$ worst-case time, for $r = \Theta(mn/\lg^k n)$, for a real constant $1 \leq k \leq 3$, and $\ell = O(n/(\lg^{k-1} n(\lg\lg n)^2))$. The space use in all the solutions is $O(n)$ words.

Proof. We straightforwardly apply the ideas presented in the previous three sections. The only modification is to allow a broader range of differences ($\{-1, 0, 1\}$) between adjacent cells in the dynamic programming matrix. This only affects a constant factor in parameter setting. □

Lemma 4 immediately serves to calculate the edit (Levenshtein) distance between two sequences (in fact, the BFC technique was presented in [3] in terms of the edit distance). We therefore obtain the following theorem.

Theorem 5. *The edit distance between two sequences, A, of length n, and B, of length m, where $n \geq m \geq \lg^2 n$, both over an integer alphabet, can be computed in $O(mn\lg\lg n/\lg^2 n)$ worst-case time. Alternatively, the distance can be found in $O(mn/\lg^2 n + r)$ worst-case time, for $r = o(mn/(\lg n\lg\lg n))$, where r is the number of symbol pairs A_i, B_j such that $A_i = B_j$. The space use in both solutions is $O(n)$ words.*

In fact, our result can also be extended to the weighted edit distance problem, if the set of weights is predefined and bounded (of constant size), as the penalties in time and/or preprocessing space are only by a constant factor.

Another feasible problem is the longest common transposition-invariant subsequence (LCTS) [16,5], in which we look for a longest subsequence of the form $(s_1 + t)(s_2 + t)\ldots(s_\ell + t)$ such that all s_i belong to A (in increasing order), all corresponding values s_i+t belong to B (in increasing order), and $t \in \{-\sigma+1\ldots\sigma-1\}$ is some integer, called a transposition. This problem is motivated by music information retrieval. The best known results for LCTS are $O(mn\lg\lg\sigma)$ [17,5] and $O(mn\sigma(\lg\lg n)^2/\lg^2 n)$ if the BFC technique is applied for all transpositions (which is $O(mn)$ if $\sigma = O(\lg^2 n/(\lg\lg n)^2)$). Applying the former result from Lemma 4, for all possible transpositions, gives immediately $O(mn\sigma\lg\lg n/\lg^2 n)$ time complexity (if $\sigma = O(n^{1-\varepsilon})$, for any $\varepsilon > 0$, otherwise the LUT build costs would dominate). Applying the latter result requires more care. First we notice that the number of matches over all the transpositions sums up to mn, so $\Theta(mn)$ is the total preprocessing cost. Let us divide the transpositions into dense ones and sparse ones, where the dense ones are those that have at least $mn\lg\lg n/\sigma$ matches. The number of dense transpositions is thus limited to $O(\sigma/\lg\lg n)$. We handle dense transpositions with the technique from Section 3 and sparse ones with the technique from Section 4. This gives us $O(mn + mn(\sigma/\lg\lg n)\lg\lg n/\lg^2 n + mn\sigma/\lg^2 n) = O(mn(1+\sigma/\lg^2 n))$ total time, assuming that $\sigma = \omega(\lg n(\lg\lg n)^2)$, as this condition on σ implies the number of matches in each sparse transposition limited to $o(mn/(\lg n\lg\lg n))$, as required. We note that $\sigma = \omega(\lg^2 n/(\lg\lg n)^2)$ and $\sigma = O(\lg^2 n)$ is the niche in which our algorithm is the first one to achieve $O(mn)$ total time.

Theorem 6. *The length of the longest common transposition-invariant subsequence (LCTS) of two sequences, A, of length n, and B, of length m, where $n \ge m \ge \lg^2 n$, both over an integer alphabet of size σ, can be computed in $O(mn(1+\sigma/\lg^2 n))$ worst-case time, assuming that $\sigma = \omega(\lg n(\lg\lg n)^2)$.*

A natural extension of Lemma 4 is to involve more than two (yet a constant number of) sequences. In particular, problems on three sequences have practical importance.

Lemma 7. *Let Q be a sequence similarity problem returning the length of a desired subsequence, involving three sequences, A of length n, B of length m and P of length u, all over a common integer alphabet Σ of size $\sigma = O(m)$. We assume that $1 \le m \le n$ and $u = \Omega(n^c)$, for some constant $c > 0$. Let Q admit a dynamic programming solution in which $M(i,j,k) - M(i-1,j,k) \in \{-1,0,1\}$, $M(i,j,k) - M(i,j-1,k) \in \{-1,0,1\}$ and $M(i,j,k) - M(i,j,k-1) \in \{-1,0,1\}$, for all valid i, j and k, and $M(i,j,k)$ depends only on the values of its (at most) seven neighbors: $M(i-1,j,k)$, $M(i,j-1,k)$, $M(i-1,j-1,k)$, $M(i,j,k-1)$, $M(i-1,j,k-1)$, $M(i,j-1,k-1)$ and $M(i-1,j-1,k-1)$, and whether $A_i = B_j$, $A_i = P_k$ and $B_j = P_k$.*

There exists a solution to Q with $O(mnu/\lg^{3/2} n)$ worst-case time. The space use is $O(n)$ words.

Proof. The solution works on cubes of size $b \times b \times b$, setting $b = \Theta(\sqrt{\lg n})$ with an appropriate constant. Instead of horizontal stripes, 3D "columns" of size $b \times b \times u$ are now used. The LUT input consists of b symbols from sequence P, encoded with respect to a supercube in $O(\lg\lg n)$ bits each, and three walls, of size $b \times b$ each, in differential representation. The outputs are the three opposite walls of a cube. The restriction $u = \Omega(n^c)$ implies that the overall time formula *without the LUT building times* is $\Omega(mn^{1+c}/\lg^{3/2} n)$, which is $\Omega(mn^{1+c'})$, for some constant c', $c \ge c' > 0$, e.g., for $c' = c/2$. The build time for all LUTs can be made $O(mn^{1+c''})$, for any constant $c'' > 0$, if the constant associated with b is chosen appropriately. It is now enough to set $c'' = c'$, to notice that the build time for the LUTs is not dominating. $\square$

As an application of Lemma 7 we present the merged longest common subsequence (MerLCS) problem [12], which involves three sequences, A, B and P, and its returned value is a longest sequence T that is a subsequence of P and can be split into two subsequences T' and T'' such that T' is a subsequence of A and T'' is a subsequence of B. Deorowicz and Danek [6] showed that in the DP formula for this problem $M(i,j,k)$ is equal to or larger by 1 than any of the neighbors: $M(i-1,j,k)$, $M(i,j-1,k)$ and $M(i,j,k-1)$. They also gave an algorithm working in $O(\lceil u/w\rceil mn\lg w)$ time. Peng et al. [18] gave an algorithm with $O(\ell mn)$ time complexity, where $\ell \le n$ is the length of the result. Motivations for the MerLCS problem, from bioinformatics and signal processing, can be found e.g. in [6].

Based on the cited DP formula property [6] we can apply Lemma 7 to obtain $O(mnu/\lg^{3/2} n)$ time for MerLCS (if $u = \Omega(n^c)$ for some $c > 0$), which may be competitive with existing solutions.

Theorem 8. *The length of the merged longest common subsequence (MerLCS) involving three sequences, A, B and P, of length respectively n, m and u, where $m \le n$ and $u = \Omega(n^c)$, for some constant $c > 0$, all over an integer alphabet of size σ, can be computed in $O(mnu/\lg^{3/2} n)$ worst-case time.*

7. Conclusions

On the example of the longest common subsequence problem we presented three algorithmic techniques, making use of tabulation and sparse dynamic programming paradigms, which allow to obtain competitive time complexities. Then we generalize the ideas by specifying conditions on DP dependencies whose fulfilments lead to immediate applications of these techniques. The actual problems considered here as applications comprise the edit distance, LCTS and MerLCS.

As a future work, we are going to relax the DP dependencies, which may for example improve the SEQ-EC-LCS result from [7]. Another research option is to try to improve the tabulation based result on compressible sequences.

Acknowledgments

The author wishes to thank Marcin Raniszewski, Sebastian Deorowicz and the anonymous reviewers for helpful comments on a preliminary version of the manuscript.

References

[1] A. Apostolico, String editing and longest common subsequences, in: Handbook of Formal Languages, in: Linear Modeling: Background and Application, vol. 2, Springer, 1997, pp. 361–398. Ch. 8.
[2] L. Bergroth, H. Hakonen, T. Raita, A survey of longest common subsequence algorithms, in: SPIRE, IEEE Computer Society, 2000, pp. 39–48.

[3] P. Bille, M. Farach-Colton, Fast and compact regular expression matching, Theoret. Comput. Sci. 409 (3) (2008) 486–496.
[4] M. Crochemore, G.M. Landau, M. Ziv-Ukelson, A subquadratic sequence alignment algorithm for unrestricted scoring matrices, SIAM J. Comput. 32 (6) (2003) 1654–1673.
[5] S. Deorowicz, Speeding up transposition-invariant string matching, Inform. Process. Lett. 100 (1) (2006) 14–20.
[6] S. Deorowicz, A. Danek, Bit-parallel algorithms for the merged longest common subsequence problem, Internat. J. Found. Comput. Sci. 24 (08) (2013) 1281–1298.
[7] S. Deorowicz, S. Grabowski, Subcubic algorithms for the sequence excluded LCS problem, in: Man-Machine Interactions, vol. 3, 2014, pp. 503–510.
[8] D. Eppstein, Z. Galil, R. Giancarlo, G.F. Italiano, Sparse dynamic programming I: Linear cost functions, J. ACM 39 (3) (1992) 519–545.
[9] P. Gawrychowski, Faster algorithm for computing the edit distance between slp-compressed strings, in: SPIRE, 2012, pp. 229–236.
[10] S. Grabowski, New tabulation and sparse dynamic programming based techniques for sequence similarity problems, in: J. Holub, J. Žďárek (Eds.), Proceedings of the Prague Stringology Conference, PSC, 2014, Czech Technical University in Prague, Czech Republic, 2014, pp. 202–211.
[11] D.S. Hirschberg, An information-theoretic lower bound for the longest common subsequence problem, Inform. Process. Lett. 7 (1) (1978) 40–41.
[12] K.-S. Huang, C.-B. Yang, K.-T. Tseng, H.-Y. Ann, Y.-H. Peng, Efficient algorithm for finding interleaving relationship between sequences, Inform. Process. Lett. 105 (5) (2008) 188–193.
[13] J.W. Hunt, T.G. Szymanski, A fast algorithm for computing longest common subsequences, Commun. ACM 20 (5) (1977) 350–353.
[14] H. Hyyrö, Bit-parallel LCS-length computation revisited, in: AWOCA, University of Sydney, Australia, 2004, pp. 16–27.
[15] W. Masek, M. Paterson, A faster algorithm computing string edit distances, J. Comput. System Sci. 20 (1) (1980) 18–31.
[16] V. Mäkinen, G. Navarro, E. Ukkonen, Transposition invariant string matching, J. Algorithms 56 (2) (2005) 124–153.
[17] G. Navarro, S. Grabowski, V. Mäkinen, S. Deorowicz, Improved time and space complexities for transposition invariant string matching, in: Technical Report TR/DCC-2005-4, Department of Computer Science, University of Chile, 2005, URL ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/mnloglogs.ps.gz.
[18] Y.-H. Peng, C.-B. Yang, K.-S. Huang, C.-T. Tseng, C.-Y. Hor, Efficient sparse dynamic programming for the merged lcs problem with block constraints, Inf. Control 6 (4) (2010) 1935–1947.
[19] Y. Sakai, A fast on-line algorithm for the longest common subsequence problem with constant alphabet, IEICE Trans. 95-A (1) (2012) 354–361.
[20] D. Salomon, Variable-Length Codes for Data Compression, vol. 140, Springer, 2007.
[21] C.K. Wong, A.K. Chandra, Bounds for the string editing problem, J. ACM 23 (1) (1976) 13–16.