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
sgrabow@kis.p.lodz.pl

Abstract. Calculating the length 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 worst-case oriented results known. We present two algorithms for LCS length calculation with respectively $O(mn \log\log n/\log^2 n)$ and $O(mn/\log^2 n + r)$ time complexity, the latter working for $r = o(mn/(\log n \log\log n))$, 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, LCTS and MerLCS problems.

Keywords: sequence similarity, longest common subsequence, sparse dynamic programming, tabulation

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, $A = a_1 \cdots a_n$ and $B = b_1 \cdots b_m$, 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 $m = \Omega(\log^2 n)$. 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 $O(n \log \sigma')$ 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).

Szymon Grabowski: New Tabulation and Sparse Dynamic Programming Based Techniques for Sequence Similarity Problems, pp. 202–211.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2 © Czech Technical University in Prague, Czech Republic

In this paper we present two techniques for finding the LCS length, one (Section 3) based on tabulation and improving the result of Bille and Farach-Colton [3] by factor $\log\log n$, the other (Section 4) combining tabulation and sparse dynamic programming and being slightly faster if the number of matches is appropriately limited. In Section 5 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.

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

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 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, $1 \leq i \leq n, 1 \leq j \leq m$, 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 $\Omega(mn)$ [19], 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 $\leq$-comparisons are allowed, then the lower bound grows to $\Omega(n \log n)$ [10]. 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 [12] has become a milestone in LCS reseach, 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 $O(n + r\log m)$ or even $O(n + r\log\log m)$ 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 $O(D\log\log(\min(D, mn/D)))$ worst-case time (plus $O(n\sigma)$ preprocessing), where $D \leq r$ 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 [18], is an improvement if the alphabet is very small (in particular, constant), as its time complexity is $O(m\sigma + \min(D\sigma, p(m-q)) + n)$, where $p = LLCS(A,B)$ 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 compexities were $O(mn/\log^2 n)$ and $O(mn(\log\log n)^2/\log^2 n)$, 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 [13] 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 $O(hmn/\log n)$ 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 $O(kn\sqrt{\log(n/k)})$ 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 $O(mn\log\log n/\log^2 n)$ time

In this section we modify the technique of Bille and Farach-Colton (BFC) [3, Sect. 4], improving its worst-case time by factor $\log\log n$, to achieve $O(mn\log\log n/\log^2 n)$ time complexity, 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 x_1 divides y and x_2 divides 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 wasn't found there. This takes $O(\log y)$ time per symbol, thus the substrings of A and B associated with the superblock are remapped in $O(y\log y)$ time. The overall alphabet remapping time for the whole matrix is thus $O((mn\log y)/y)$.

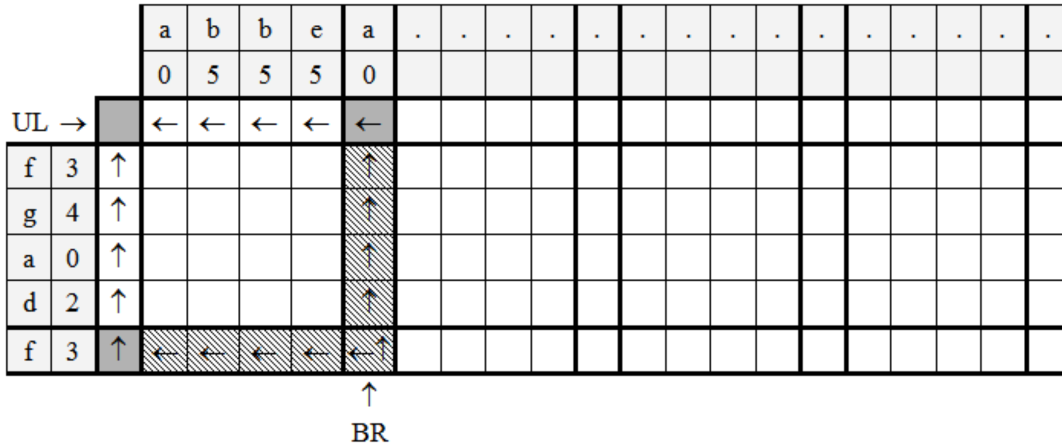

Figure 1. One horizontal stripe of the DP matrix, with 4 blocks of size 5×5 ($x_1 = x_2 = 4$). The corresponding snippets from sequence A and B are `abbea` and `fgadf`, respectively. These snippets are translated to a new alphabet (the procedure for creating the new alphabet is not shown here) of size 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 explictly 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.

This remapping technique allows to represent the symbols from the input components (i) and (ii) on $O(\log \min(y+1, \sigma))$ bits each, rather than $\Theta(\log \sigma)$ 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 $x_1 \log(y + 1) + x_2 \log(y + 1) + x_1 + x_2$ bits, which cannot sum to $\omega(\log n)$ 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 $x_1 = x_2 = \log n/(6 \log\log n)$, we obtain $O(mn(\log\log n)^2/\log^2 n)$ 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 $x_1 \log(y+1) + x_1 + x_2 = x_1 \log(2(y+1)) + x_2$ bits.

The return value associated with each LUT key are 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 $x_1 + x_2 + \log(\min(x_1, x_2) + 1)$ bits in total. Fig. 1 illustrates.

As long as the input and the output of an LUT fits a machine word, i.e., does 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:

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

where $2^{x_1 \log(2(y+1))+x_2}$ is the number of all possible LUT inputs and the $x_1 x_2$ multiplier corresponds to the computation time per one LUT cell. Let us set $y = \log^2 n/2$, $x_1 = \log n/(4 \log\log n)$ and $x_2 = \log n/4$. In total we obtain $O(mn \log\log n/ \log^2 n)$ 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 factor $\log\log n$. 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) between two sequences, A, of length n, and B, of length m, where $n \geq m \geq \log^2 n$, both over an integer alphabet, can be computed in $O(mn \log\log n/ \log^2 n)$ worst-case time. The algorithm needs $o(n)$ words of space, apart for the two sequences themselves.*

4 LCS in $O(mn/ \log^2 n + r)$ 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 boundary 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 $\log(b^2)$ bits, totalling $O(K \log b)$ bits.

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 $b = O(\log n)$ (otherwise the LUT build costs would be dominating) and $b = \omega(\log n/\sqrt{\log\log n})$ (otherwise this algorithm would never be better than the one from Section 3), which implies that $K = \Theta(\log n/ \log\log n)$, with an appropriate constant.

As this algorithm's worst-case time is $\Omega(mn/\log^2 n)$, 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 $\Theta(\log n/\log\log n) \times \Theta(b)$. 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, $f_d = 0.5$). The total time complexity (without preprocessing) is then

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

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 $f_d = O((r/K)/(mn/b^2)) = O(rb^2 \log\log n/(mn \log n))$. From the $f_d = o(1)$ condition we also obtain that $rb^2 = o(mn \log n/\log\log n)$. If $r = o(mn/(\log n \log\log n))$, then we can safely use the maximum possible value of b, i.e., $b = \Theta(\log n)$ and obtain the time of $O(mn/\log^2 n)$.

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) between two sequences, A, of length n, and B, of length m, where $n \geq m \geq \log^2 n$, both over an integer alphabet, can be computed in $O(mn/\log^2 n + r)$ worst-case time, assuming $r = o(mn/(\log n \log\log n))$, where r is the number of matching pairs of symbols between A and B.*

Considering to 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 $O(r \log\log(\min(r, mn/r)))$ in the worst case. Our result is better if $r = \omega(mn/(\log^2 n \log\log\log n))$ and $r = o(mn)$. Overall, it gives the niche of $r = \omega(mn/(\log^2 n \log\log\log n))$ and $r = o(mn \log\log n/\log^2 n)$ in which the algorithm presented in this section is competitive.

The alphabet size is yet another constraint. From the comparison to Sakai's algorithm [18] we conclude that our algorithm needs $\sigma = \omega(\log\log\log n)$ to dominate for the case of $r = \omega(mn/(\log^2 n \log\log\log n))$.

5 Algorithmic applications

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

Lemma 3. *Let Q be a sequence similarity problem returning the length of a desired subsequence, involving two sequences, A of length n and B of length m, both over a*

[1] A slightly more precise bound on D is $\min(r, m^2)$, but it may matter, in complexity terms, only if $m = o(n)$ (cf. also [18, Th. 1]), which is a less interesting case.

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 $O(mn \log\log n / \log^2 n)$ worst-case time. There also exists a solution to Q with $O(mn/\log^2 n + r)$ worst-case time, for $r = o(mn/(\log n \log\log n))$, where r is the number of symbols pairs A_i, B_j such that $A_i = B_j$. The space use in both solutions is $O(n)$ words.

Proof. We straightforwardly apply the ideas presented in the previous two 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 3 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 4. *The edit distance between two sequences, A, of length n, and B, of length m, where $n \geq m \geq \log^2 n$, both over an integer alphabet, can be computed in $O(mn \log\log n/\log^2 n)$ worst-case time. Alternatively, the distance can be found in $O(mn/\log^2 n + r)$ worst-case time, for $r = o(mn/(\log n \log\log n))$, where r is the number of symbols pairs A_i, B_j such that $A_i = B_j$. The space use in both solutions is $O(n)$ words.*

Another feasible problem is the longest common transposition-invariant subsequence (LCTS) [14,5], in which we look for a longest subsequence of the form $(s_1 + t)(s_2 + t) \cdots (s_\ell + t)$ 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 $O(mn \log\log\sigma)$ [16,5] and $O(mn\sigma(\log\log n)^2/\log^2 n)$ if the BFC technique is applied for all transpositions (which is $O(mn)$ if $\sigma = O(\log^2 n/(\log\log n)^2)$). Applying the former result from Lemma 3, for all possible transpositions, gives immediately $O(mn\sigma \log\log n/\log^2 n)$ 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 sum 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 $mn \log\log n/\sigma$ matches. The number of dense transpositions is thus limited to $O(\sigma/\log\log n)$. We handle dense transpositions with the technique from Section 3 and sparse ones with the technique from Section 4. This gives us $O(mn + mn(\sigma/\log\log n)\log\log n/\log^2 n + mn\sigma/\log^2 n) = O(mn(1+\sigma/\log^2 n))$ total time, assuming that $\sigma = \omega(\log n(\log\log n)^2)$, as this condition on σ implies the number of matches in each sparse transposition limited to $o(mn/(\log n \log\log n))$, as required. We note that $\sigma = \omega(\log^2 n/(\log\log n)^2)$ and $\sigma = O(\log^2 n)$ is the niche in which our algorithm is the first one to achieve $O(mn)$ total time.

Theorem 5. *The length of the longest common transposition-invariant subsequence (LCTS) between two sequences, A, of length n, and B, of length m, where $n \geq m \geq$*

$\log^2 n$, *both over an integer alphabet of size* σ, *can be computed in* $O(mn(1+\sigma/\log^2 n))$ *worst-case time, assuming that* $\sigma = \omega(\log n (\log\log n)^2)$.

A natural extension of Lemma 3 is to involve more than two (yet a constant number of) sequences. In particular, problems on three sequences have practical importance.

Lemma 6. *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* $O(mnu/\log^{3/2} n)$ *worst-case time. The space use is* $O(n)$ *words.*

Proof. The solution works on cubes of size $b \times b \times b$, setting $b = \Theta(\sqrt{\log n})$ 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 $O(\log\log n)$ bits each, and three walls, of size $b \times b$ each, in differential representation. The output are the three opposite walls of a cube. The restriction $u = \Omega(n^c)$ implies that the overall time formula *without the LUT build times* is $\Omega(mn^{1+c}/\log^{3/2} n)$, 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. We now set $c'' = c'$ to show the build time for the LUTs is not dominating. □

As an application of Lemma 6 we present the merged longest common subsequence (MerLCS) problem [11], 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 $O(\lceil u/w \rceil mn \log w)$ time. Peng et al. [17] 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 6 to obtain $O(mnu/\log^{3/2} n)$ time for MerLCS (if $u = \Omega(n^c)$ for some $c > 0$), which may be competitive with existing solutions.

Theorem 7. *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* $O(mnu/\log^{3/2} n)$ *worst-case time.*

6 Conclusions

On the example of the longest common subsequence problem we presented two 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 Sebastian Deorowicz and an anonymous reviewer 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, vol. 2 Linear Modeling: Background and Application, Springer, 1997, ch. 8, pp. 361–398.
2. L. Bergroth, H. Hakonen, and T. Raita: *A survey of longest common subsequence algorithms*, in SPIRE, IEEE Computer Society, 2000, pp. 39–48.
3. P. Bille and M. Farach-Colton: *Fast and compact regular expression matching.* Theoret. Comput. Sci., 409(3) 2008, pp. 486–496.
4. M. Crochemore, G. M. Landau, and M. Ziv-Ukelson: *A subquadratic sequence alignment algorithm for unrestricted scoring matrices.* SIAM J. Comput., 32(6) 2003, pp. 1654–1673.
5. S. Deorowicz: *Speeding up transposition-invariant string matching.* Inform. Process. Lett., 100(1) 2006, pp. 14–20.
6. S. Deorowicz and A. Danek: *Bit-parallel algorithms for the merged longest common subsequence problem.* Internat. J. Foundations Comput. Sci., 24(08) 2013, pp. 1281–1298.
7. S. Deorowicz and S. Grabowski: *Subcubic algorithms for the sequence excluded LCS problem*, in Man-Machine Interactions 3, 2014, pp. 503–510.
8. D. Eppstein, Z. Galil, R. Giancarlo, and G. F. Italiano: *Sparse dynamic programming I: Linear cost functions.* J. ACM, 39(3) 1992, pp. 519–545.
9. P. Gawrychowski: *Faster algorithm for computing the edit distance between SLP-compressed strings*, in SPIRE, 2012, pp. 229–236.
10. D. S. Hirschberg: *An information-theoretic lower bound for the longest common subsequence problem.* Inform. Process. Lett., 7(1) 1978, pp. 40–41.
11. K.-S. Huang, C.-B. Yang, K.-T. Tseng, H.-Y. Ann, and Y.-H. Peng: *Efficient algorithm for finding interleaving relationship between sequences.* Inform. Process. Lett., 105(5) 2008, pp. 188–193.
12. J. W. Hunt and T. G. Szymanski: *A fast algorithm for computing longest common subsequences.* Comm. ACM, 20(5) 1977, pp. 350–353.
13. H. Hyyrö: *Bit-parallel LCS-length computation revisited*, in AWOCA, Australia, 2004, University of Sydney, pp. 16–27.
14. V. Mäkinen, G. Navarro, and E. Ukkonen: *Transposition invariant string matching.* Journal of Algorithms, 56(2) 2005, pp. 124–153.
15. W. Masek and M. Paterson: *A faster algorithm computing string edit distances.* J. Comput. Syst. Sci., 20(1) 1980, pp. 18–31.
16. G. Navarro, S. Grabowski, V. Mäkinen, and S. Deorowicz: *Improved time and space complexities for transposition invariant string matching*, Technical Report TR/DCC-2005-4, Department of Computer Science, University of Chile, 2005.

17. Y.-H. Peng, C.-B. Yang, K.-S. Huang, C.-T. Tseng, and C.-Y. Hor: *Efficient sparse dynamic programming for the merged LCS problem with block constraints*. International Journal of Innovative Computing, Information and Control, 6(4) 2010, pp. 1935–1947.
18. Y. Sakai: *A fast on-line algorithm for the longest common subsequence problem with constant alphabet*. IEICE Transactions, 95-A(1) 2012, pp. 354–361.
19. C. K. Wong and A. K. Chandra: *Bounds for the string editing problem.* J. ACM, 23(1) 1976, pp. 13–16.