$O(mn \log \sigma)$ Time Transposition Invariant LCS Computation

Szymon Grabowski[1] and Gonzalo Navarro[2⋆]

[1] Computer Engineering Department, Technical University of Łódź, Poland.
[2] Center for Web Research, Department of Computer Science, University of Chile.

Abstract. Given strings A and B of lengths m and n over a finite alphabet $\Sigma \subset \mathbb{Z}$ of size $O(\sigma)$, the length of the *longest common transposition invariant subsequence* is $LCTS(A,B) = \max_{t \in \mathbb{Z}}\{LCS(A+t, B)\}$, where $A + t = (a_1 + t)(a_2 + t)\ldots(a_m + t)$ and $LCS(A + t, B)$ is the length of the longest common subsequence between $A + t$ and B. $LCTS(A,B)$ can be computed naively in $O(mn\,\sigma)$ time. We present a simple and easy to implement algorithm obtaining $O(mn \log \sigma)$ time. We also show that transposition invariant Levenshtein distance can be computed in $O(mn\sqrt{\sigma})$ time.

1 Introduction

Transposition invariant string matching is the problem of matching two strings when all the characters of either of them can be "shifted" by some amount t. By "shifting" we mean that the strings are sequences of numbers and we add or subtract t from each character of one of them.

Interest in transposition invariant string matching problems has recently arisen in the field of music information retrieval (MIR) [2, 8, 9]. In music analysis and retrieval, one often wants to compare two music pieces to test how similar they are. A reasonable way of modeling music is to consider the pitches and durations of the notes. The durations are however often omitted, since it is usually possible to recognize the melody from a sequence of pitches. In general, edit distance measures can be used for matching two pitch sequences. One of the most widely accepted similarity measures for matching music is the *longest common subsequence* (LCS) among the pitch sequences. This is the longest string that can be obtained by removing characters from each of the two sequences. A second measure (actually a dissimilarity measure) is Levenshtein distance, which permits substituting characters by others apart from removing them.

A particular feature of music retrieval is *transposition invariance*: The same melody is perceived even if the pitch sequence is shifted from one key to another. This is equivalent to adding a constant to all the pitch values of one sequence. This paper focuses on computing LCS under transposition invariance, that is, finding the longest common subsequence that can be obtained after the best possible shifting of one sequence.

⋆ Supported by Millenium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile.

2 Problem Statement and Our Contribution

Let $\Sigma \subset \mathbb{Z}$ be a finite numerical alphabet. For simplicity we consider $\Sigma = \{0, \ldots, \sigma\}$ in this paper, although any subset of $\mathbb{Z}$ can be handled with little extra overhead. Let $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$ be two *strings* over Σ^*, that is, *characters* a_i, b_j of the two strings belong to Σ for all $1 \leq i \leq m, 1 \leq j \leq n$. String A' is a *subsequence* of A, denoted by $A' \sqsubseteq A$, if $A' = a_{i_1} a_{i_2} \ldots a_{i_{|A'|}}$ for some indexes $1 \leq i_1 < i_2 < \cdots < i_{|A'|} \leq m$. The length of the *longest common subsequence (LCS)* of A and B is $LCS(A, B) = \max\{|S|,\ S \sqsubseteq A, S \sqsubseteq B\}$.

A *transposed copy* of a string A, denoted by $A + t$ for some $t \in \mathbb{Z}$, is $A + t = (a_1 + t)(a_2 + t) \cdots (a_m + t)$. Our goal is to compute the length of the *longest common transposition invariant subsequence* of A and B:

$$LCTS(A, B) \quad = \quad \max_{t \in \mathbb{Z}} LCS(A + t, B) \quad = \quad \max_{t \in [-\sigma, \sigma]} LCS(A + t, B),$$

where the latter equality owes to the fact that transpositions t outside the range $[-\sigma, \sigma]$ will not match any character of A to B, and thus $LCS(A + t, B) = 0$ for those t. The computation of $LCS(A, B)$ can be carried out in $O(mn)$ time using the well-known recurrence (e.g. [5])

$$M(i, 0) \quad = \quad M(0, j) \quad = \quad 0, \tag{1}$$

$$M(i, j) \quad = \quad \textbf{if } a_i = b_j \textbf{ then } 1 + M(i - 1, j - 1) \quad \textbf{else } \max(M(i - 1, j), M(i, j - 1)), \tag{2}$$

so that $LCS(A, B) = M(m, n)$.

The computation of $LCTS(A, B)$ can be done naively in $O(mn\,\sigma)$ time [9] by considering all transpositions $t \in [-\sigma, \sigma]$. A more sophisticated algorithm, based on the idea that only some characters of A and B match for each transposition t, resorts to sparse dynamic programming to obtain $O(mn \log\log \min(m, n))$ time [12]. Most recently [7], an algorithm that backtracks over the set of possible transpositions obtains a best case of $O((mn + \log\log\sigma)\log\sigma)$ and a worst case of $O((mn + \log\sigma)\sigma)$.

In this paper we present an algorithm that computes $LCTS(A, B)$ in worst case time $O(mn \log\sigma)$. The main idea is to reduce the problem to the naive computation over small submatrices of M, so that few transpositions are relevant inside each submatrix. The algorithm is practical and simple to implement. In terms of complexity, it compares favorably against the naive algorithm [9], against the best case of the backtracking algorithm [7] and, for moderate $\sigma < \log\min(m, n)$, against the sparse dynamic programming algorithm [12].

At the end, we show how the techniques we developed for LCTS can be used to compute transposition invariant Levenshtein distance in $O(mn\sqrt{\sigma})$ time. We defer the formal definitions for this case to Section 6.

3 A Basic $O(mn\sqrt{\sigma})$ Time Algorithm for LCTS

We compute a dynamic programming matrix $M_t(0 \ldots m, 0 \ldots n)$ for each transposition $t \in -\sigma \ldots \sigma$, which corresponds to $LCS(A + t, B)$. The main idea is

not to compute all the cells of all the matrices. We divide each matrix M_t into $O(mn/k^2)$ blocks of $k \times k$ cells. Blocks will be labeled (i, j), for $0 \leq i < \lceil m/k \rceil$ and $0 \leq j < \lceil n/k \rceil$, corresponding to $M(ki+1 \ldots ki+k, kj+1 \ldots kj+k)$. The bottom and rightmost blocks may not be full but we ignore that for simplicity.

Inside each block, there are at most k^2 *relevant* transpositions t such that $LCS(A_{ki+1 \ldots ki+k} + t, B_{kj+1 \ldots kj+k}) > 0$. That is, only those transpositions $t = b_{kj+s} - a_{ki+r}$, where $0 < r, s \leq k$, yield at least one match inside block (i, j). All the other (at least) $2\sigma + 1 - k^2$ *irrelevant* transpositions yield a null longest common subsequence.

According to Eq. (2), matrices M_t can be computed in any order, as long as cell (r, s) is computed after cells $(r-1, s)$ and $(r, s-1)$. We choose to compute all the M_t matrices simultaneously, row by row of blocks, each row from left to right. All the different transpositions for each block (i, j) are computed simultaneously.

Let us focus on the computation of a single block (i, j). The idea is to treat the (at most) k^2 relevant transpositions differently from the other irrelevant (at least) $2\sigma + 1 - k^2$ ones. First, we compute array $S(-\sigma \ldots \sigma)$ so that $S(t) = 1$ iff transposition t is relevant for the current block (i, j). Array S is trivially computed in $O(\sigma + k^2)$ time. Now, for each relevant transposition t, we fully compute $M_t(ki+1 \ldots ki+k, kj+1 \ldots kj+k)$ in $O(k^2)$ time.

A problem to compute the first row and column values $M_t(ki+1, kj+s)$ and $M_t(ki+r, kj+1)$, for $0 < r, s \leq k$, for a relevant transposition t, is that we need to know the bottom and rightmost values of neighboring blocks, $M_t(k(i-1)+k, kj+s)$ and $M_t(ki+r, k(j-1)+k)$, so as to apply Eq. (2). Yet, t might not be relevant on those neighboring blocks and hence those values might not be known. Therefore, previously to computing M_t in block (i, j), we must fill the bottom row of block $(i-1, j)$ and the rightmost column of block $(i, j-1)$.

Let us focus on computing $M_t(k(i-1)+k, kj+s)$, as the case of $M_t(ki+r, k(j-1)+k)$ is symmetric. Let $(i-p, j)$ be the last block where transposition t was relevant. That is, we have computed value $M_t(k(i-p)+k, kj+s)$, and we know that transposition t is not relevant for blocks $(i-p+1, j), \ldots, (i-1, j)$. Thus $a_{k(i-p+1)+r'} + t \neq b_{kj+s}$, for $0 < r' \leq k(p-1)$. The following property tells us how to compute $M_t(k(i-1)+k, kj+s)$ directly.

Property 1. If $a_r \neq b_s$ for all $i < r \leq i'$ and $j < s \leq j'$, then Eq. (2) implies

$$M(i', j') \quad = \quad \max(M(i, j'), M(i', j)).$$

Proof. It is easy to see by induction on Eq. (2) that

$$M(i', j') \quad = \quad \max\left(\max_{i<r\leq i'} M(r, j), \max_{j<s\leq j'} M(i, s)\right) \quad = \quad \max(M(i', j), M(i, j')),$$

where the second step obeys to the obvious monotonocity property $LCS(Aa, B) \geq LCS(A, B)$, and therefore $M_t(r+1, s) \geq M_t(r, s)$ for any r, s. See also [1]. □

Therefore, we have

$$M_t(k(i-1)+k, kj+s) \; = \; \max(M_t(k(i-1)+k, kj+s-1), M_t(k(i-p)+k, kj+s)), \tag{3}$$

which solves our problem but brings in other two subproblems: (i) for the case $s = 1$ we must compute corner values $M_t(ki + k, kj + k)$ for all i, j and all transpositions, not only for the relevant ones; (ii) we must determine $i - p$ in constant time.

To solve subproblem (i), we must compute corner values for irrelevant transpositions too. These are easily computed given previous corners since, by the same Property 1,

$$M_t(ki + k, kj + k) \quad = \quad \max(M_t(k(i-1) + k, kj + k), M_t(ki + k, k(j-1) + k)).$$

To solve subproblem (ii) in constant time we will maintain, for every transposition t, vectors

$$\begin{aligned} R_t(s) &= M_t(\max\{r',\ M_t(r', s) \text{ has been computed}\},\ s), \\ C_t(r) &= M_t(r,\ \max\{s',\ M_t(r, s') \text{ has been computed}\}). \end{aligned}$$

Note that row and column zero of matrices M_t are always zero by Eq. (1), so we initialize $R_t(s) = C_t(r) = 0$ for all $1 \le r \le m$, $1 \le s \le n$. Later, every time a new matrix value $M_t(r, s)$ is written, we update $R_t(s) = C_t(r) = M_t(r, s)$. The next time we try to compute $M_t(r', s)$ or $M_t(r, s')$, for $r' > r$ or $s' > s$, the latest value $M_t(r, s)$ we require will be in $R_t(s)$ and $C_t(r)$ [1].

Hence, Eq. (3) is implemented using $R_t(kj + s) = M_t(k(i - p) + k, kj + s)$, and $M_t(ki + r, k(j - 1) + k)$ is filled similarly:

$$\begin{aligned} M_t(k(i-1) + k, kj + s) &= \max(M_t(k(i-1) + k, kj + s - 1), R_t(kj + s)), \\ M_t(ki + r, k(j-1) + k) &= \max(M_t(ki + r - 1, k(j-1) + k), C_t(ki + r)). \end{aligned}$$

Adding all the computation costs per block, we have $O(\sigma + k^2)$ to compute S, $O(\sigma)$ to compute the corners $M_t(ki + k, kj + k)$, and $O(k^4)$ to compute the block of size $O(k^2)$ for all the $O(k^2)$ relevant transpositions. This is

$$O\left(mn/k^2\left(\sigma + k^2 + \sigma + k^4\right)\right) \quad = \quad O\left(mn\left(\sigma/k^2 + k^2\right)\right),$$

which is optimized for $k = \sigma^{1/4}$ to yield overall complexity $O(mn\sqrt{\sigma})$. To this we must add the cost to initialize R and C, so overall we have $O(mn\sqrt{\sigma} + (m+n)\sigma)$. Note that we assumed $\min(m, n) \ge \sigma^{1/4}$ to optimize the block size. This will be true when we use this algorithm as a building block for Section 4.

Let us consider space. For each new row i of blocks computed we only need array S for the current block, all $R_t(s)$ values for $0 < s \le n$, values $C_t(r)$ for the rows $ki + 1 \le r \le ki + k$ in the current block, and corner values $M_t(k(i - 1) + k, kj + k)$ for $0 < j \le \lceil n/k \rceil$ in the previous row. Thus overall we need $O(\min(m, n)\,\sigma)$ space (since A and B can be exchanged if $n > m$).

[1] We could have saved some work by defining R and C block-wise rather than cell-wise. However, we need such finer-grained version for Section 4.

4 Recursing for Improved Complexity

The basic method of Section 3 demonstrates that all the $LCS(A+t,B)$ values, where the strings are of length m and n and there are $O(\sigma)$ relevant transpositions to consider, can be computed in $T_1(m,n,\sigma) = O(mn\sqrt{\sigma})$ time if we exclude initialization of R and C. We use that basic algorithm to compute the results for the k^2 relevant transpositions inside a block, instead of computing them naively. Hence the cost for relevant transpositions would drop from $T_0(k,k,k^2) = O(k^4)$ to $T_1(k,k,k^2) = O(k^3)$.

It is necessary, however, to consider that in the previous section we assumed a contiguous alphabet of the form $-\sigma \ldots \sigma$, which will not hold when we use the algorithm on a subset of the alphabet of size k^2. We redefine $S(-\sigma \ldots \sigma)$ as a mapping that indexes into an array $U(1 \ldots k^2)$ holding the relevant transpositions of the block, so $U(S(t)) = t$, while $S(t) = 0$ for the irrelevant transpositions. Arrays S and U are computed in $O(\sigma + k^2)$ time and passed to the basic procedure for a $k \times k$ block. The basic procedure takes U as the universe of valid transpositions, and uses S to find the place in U of a given transposition $t = b_s - a_r$. To further recurse with $k' \times k'$ blocks, the invoked procedure can reuse the same S array and a new universe U'. Array S is reinitialized in $O(|U|)$ time by setting $S(U(u)) = 0$ for $1 \le u \le |U|$, and then the mapping onto U' is computed in $O(k'^2 + |U'|)$ time. After the invocation, S is restored in $O(|U|)$ time by setting $S(U(u)) = u$ for $1 \le u \le |U|$. Therefore, we have $T_1(k,k,k^2) = O(k^3)$ obliviously to the universe Σ.

The invoked procedures will share global R and C vectors retaining the last value assigned to the global M_t matrix. Hence a recursive computation for a block reads the last values written by a previous recursive computation of another block. This works correctly because the top border cells (r,s) are filled considering only that transposition t is not relevant in column s since the last time $R_t(s)$ was assigned. The same holds for the leftmost border and $C_t(r)$.

By using $T_1(k,k,k^2)$ instead of the naive $O(k^4)$ algorithm, the complexity becomes

$$O\left(mn/k^2\left(\sigma + k^2 + k^3\right)\right) \quad = \quad O\left(mn\left(\sigma/k^2 + k\right)\right),$$

which is optimized for $k = \sigma^{1/3}$ to yield overall complexity $T_2(m,n,\sigma) = O(mn\,\sigma^{1/3})$. By using this $T_2(k,k,k^2) = O(k^{2+2/3})$ time procedure instead of T_1 we obtain a lower complexity, and so on.

To generalize the analysis to h levels of recursion we must consider the constant factors, which grow with the depth of the recursion. In this case, derivation is necessary to find the optimum k.

With h levels of recursion we obtain complexity $O\left((h+1)mn\,\sigma^{\frac{1}{h+1}}\right)$, thus $T_h(k,k,k^2) = O\left((h+1)k^{2+\frac{2}{h+1}}\right)$. This is clearly true for $h = 0$, while for general h we have by induction the complexity

$$O\left(mn/k^2\left(\sigma + k^2 + T_{h-1}(k,k,k^2)\right)\right) \quad = \quad O\left(mn\left(\sigma/k^2 + h\,k^{\frac{2}{h}}\right)\right),$$

which, by deriving with respect to k, is optimized for $k = \sigma^{\frac{h}{2h+2}}$ to yield overall complexity $O((h+1)mn\,\sigma^{\frac{1}{h+1}})$ as promised.

This complexity is optimized after $h = \ln(\sigma) - 1$ levels of recursion, to yield $O(mn \log \sigma)$ complexity. Yet, we still have the additional cost $O((m+n)\sigma)$ to initialize vectors R and C. In the next section we get rid of this additional complexity. Note, however, that this is important only for large alphabets, $\sigma / \log \sigma > \min(m, n)$.

A second possible problem resulting from small $\min(m, n)$ is that the choice of optimum k assumes $\min(m, n) \geq \sqrt{\sigma}$. This is a result of having used square blocks for simplicity. However, we could perfectly use blocks of $k_1 \times k_2$, and the optimization for level h would simply state $k_1 k_2 = \sigma^{\frac{h}{h+1}}$. Say that $m \leq n$. Then we could use $k_1 = k_2 = \sigma^{\frac{h}{2h+2}}$ except when $m < \sigma^{\frac{h}{2h+2}}$, in which case we could use $k_1 = m$ and $k_2 = \sigma^{\frac{h}{2h+2}}/m$. (Since k_2 has to be an integer, we could use $k_1 = \alpha m$ and $k_2 = \sigma^{\frac{h}{2h+2}}/(\alpha m)$, for any $0 < \alpha \leq 1$ to get closer to the desired $k_1 k_2$.)

It is possible, however, that now $k_2 = \sigma^{\frac{h}{2h+2}}/m > n$, which means that $mn < \sqrt{\sigma}$. In this case it is better to collect all the mn transpositions $b_s - a_r$ into a balanced tree and work only with these. This effectively reduces σ to mn and the problem cannot occur anymore. The cost to collect the transpositions is $O(mn \log(mn)) = O(mn \log \sigma)$, so the complexity does not change (actually the complexity improves to $O(mn \log \max(m, n)))$. In order to map transpositions $b_s - a_r$ into the compact set we are using, we must use vectors S and U as described in the beginning of this section for the top-level procedure too. We only have to avoid initializing S fully, but just for the existing transpositions $b_s - a_r$.

5 Reducing Initialization Time

In this section we show how the $O((m+n)\sigma)$ cost to initialize vectors R_t and C_t can be reduced to $O(mn)$. The idea is to initialize those vectors only for the transpositions that appear in the matrix. That is, for each cell (r, s), $1 \leq r \leq m$ and $1 \leq s \leq n$, we only initialize $R_t(s) = C_t(r) = 0$ for transposition $t = b_s - a_r$. This clearly makes initialization time $O(mn)$. However, we must ensure that $R_t(s)$ will be accessed only when t is relevant for some cell (r', s), $1 \leq r' \leq m$, and similarly $C_t(r)$ for (r, s'), $1 \leq s' \leq n$.

For this sake, we will replace the basic $O(k^4)$ algorithm that computes the relevant transpositions inside a block by a more sophisticated $O(k^3)$ one. The improved complexity is not important, as in Section 4 we have shown that recursion permits obtaining $O(mn \log \sigma)$ independently of the quality of the basic procedure. What is important of the new procedure is that it will only access $M_t(r, s)$ when t is relevant somewhere in row r, and somewhere in column s.

Consider the computation for the relevant transpositions in a single block. When we scan the block for the relevant transpositions, we compute not only the original array $S(-\sigma \ldots \sigma)$, but also an array $H(-\sigma \ldots \sigma)$, where $H(t)$ is the

list of cells (r, s) in the block corresponding to transposition t, that is, cell (r, s) appears in list $H(b_{kj+s} - a_{ki+r})$. All this is accomplished in $O(\sigma + k^2)$ time.

Then, the relevant transpositions t are processed one by one. All the rows r and columns s in list $H(t)$ are marked, and subsequently traversed to make two increasing lists $r_1, \ldots, r_f$ and $s_1, \ldots, s_c$ of rows and columns (without duplicates). Finally, the procedure of Eq. (2) is applied by referring only to those marked rows and columns. All the others yield no matches between $A + t$ and B. Thus the following recurrence holds for the cells $(ki + r_p, kj + s_q)$ where $t = b_{kj+s_q} - a_{ki+r_p}$:

$$\begin{aligned} M_t(ki + r_p, kj + s_q) &= 1 + M_t(ki + r_p - 1, kj + s_q - 1) \\ &= 1 + \max(M_t(ki + r_{p-1}, kj + s_q), M_t(ki + r_p, kj + s_{q-1})), \end{aligned}$$

where the first equality owes to Eq. (2), and the second to Property 1. On the other hand, when $t \neq b_{kj+s_q} - a_{ki+r_p}$, we have by Property 1:

$$M_t(ki + r_p, kj + s_q) = \max(M_t(ki + r_{p-1}, kj + s_q), M_t(ki + r_p, kj + s_{q-1})).$$

Hence each transposition can be computed in $O(r_f s_c)$ time. The sum of all the lengths of the (up to) k^2 entries in H is exactly k^2. In the worst case all the rows and columns are different in each list, so the r_f values and the s_c values might add up k^2, but no individual r_f or s_c term can exceed k. The worst case occurs when as few terms as possible are as large as possible, that is, k values r_f and s_c have value (almost) k and the others have value (almost) zero. In this case the overall cost for all the relevant transpositions is $O(k^3)$.

Note that, this time, even for the relevant transpositions we do not fill the whole bottom row and rightmost column of the blocks. This is not necessary, as R_t and C_t keep row-wise and column-wise information of the last time transposition t was relevant.

6 Transposition Invariant Levenshtein Distance

The Levenshtein distance [10] $ed(A, B)$ between strings A and B is the minimum number of character insertions, deletions, and substitutions, necessary to make them equal. Levenshtein distance can be computed in $O(mn)$ time [13] time with the classical recurrence:

$$\begin{aligned} D(0,0) &= 0 \\ D(i,j) &= \textbf{if } a_i = b_j \textbf{ then } D(i-1, j-1) \\ &\quad \textbf{else } 1 + \min(D(i-1,j), D(i,j-1), D(i-1,j-1)), \end{aligned}$$

so that $ed(A, B) = D(m, n)$.

Transposition invariant Levenshtein distance aims at finding the minimum distance $ed(A + t, B)$ over all t. Apart from the obvious $O(mn\,\sigma)$ time solution, there exists an $O(mn \log\log \max(m, n))$ time algorithm based on sparse dynamic programming [12]. The backtracking algorithm [7] can also be extended to deal

with this distance. In this section we show how the ideas we developed for LCTS can be extended to obtain $O(mn\sqrt{\sigma})$ time for Levenshtein distances.

First, we can apply the basic algorithm of Section 3. However, as no simple rule like Property 1 holds to handle regions without matches, we must fully compute the bottom row and rightmost column for all the irrelevant transpositions as well. This ensures that the neighboring blocks always have computed the transpositions one needs for the current block. The $O(\sigma k)$ border cells for irrelevant transpositions can be filled in $O(\sigma k)$ time using the algorithm for *different letter boxes* from [11, Section 3]. Thus the overall cost is

$$O\left(mn/k^2\left(\sigma + k^2 + k^4 + \sigma k\right)\right) \quad = \quad O\left(mn\left(\sigma/k + k^2\right)\right),$$

which is optimized for $k = \sigma^{1/3}$ to yield $O(mn\,\sigma^{2/3})$ complexity.

Recursion can be applied over this basic scheme, just as in Section 4. We call $T_0(k,k,k^2) = O(k^4)$ the complexity of the naive computation of relevant transpositions, $T_1(k,k,k^2) = O(k^{2+4/3})$ that of computing the transpositions using the $O(mn\,\sigma^{2/3})$ time algorithm, and so on. This time the analysis is more complex and we choose k optimizing only the complexity, not the constant. This is sufficient since, as we see next, even so the constant does not grow. With h levels of recursion we obtain $O\left(2^{\frac{2^{h+2}-h-4}{2^{h+1}-1}}\, mn\,\sigma^{\frac{2^h}{2^{h+1}-1}}\right)$ complexity, thus $T_h(k,k,k^2) = O\left(2^{\frac{2^{h+2}-h-4}{2^{h+1}-1}}\, k^{2+\frac{2^{h+1}}{2^{h+1}-1}}\right)$. This is clearly true for $h = 0$, while for general h we have by induction the complexity

$$O\left(\frac{mn}{k^2}\left(\sigma + k^2 + T_{h-1}(k,k,k^2) + \sigma k\right)\right) \quad = \quad O\left(mn\left(2^{\frac{2^{h+1}-h-3}{2^h-1}}\, k^{\frac{2^h}{2^h-1}} + \frac{\sigma}{k}\right)\right),$$

which is optimized (in complexity) for $k = \sigma^{\frac{2^h-1}{2^{h+1}-1}}/2^{\frac{2^{h+1}-h-3}{2^{h+1}-1}}$ to yield the promised result. Since the constant (as a function of h) is upper bounded by 4, the complexity is actually $O\left(mn\,\sigma^{\frac{2^h}{2^{h+1}-1}}\right)$.

Thus we can reach complexity $O\left(mn\,\sigma^{1/2+\varepsilon}\right)$ for any $\varepsilon > 0$, by using $h = \log_2\left(1 + \frac{1}{2\varepsilon}\right) - 1$ levels of recursion. In particular, by choosing $\varepsilon = 1/\log\sigma$ we obtain the promised $O(mn\sqrt{\sigma})$ complexity with $O(\log\log\sigma)$ levels of recursion.

7 Conclusions

We have presented an $O(mn\log\sigma)$ time algorithm to compute the longest common subsequence with transposition invariance. The algorithm is simple and easy to implement. Its complexity is better than all existing alternatives, except the one based on dynamic programming [12] for large $\sigma > \log\min(m,n)$. We have also shown that the transposition invariant version of Levenshtein distance can be computed in time $O(mn\sqrt{\sigma})$.

The algorithms are easily extended to the search problem, where one seeks for all the substrings of B that are similar enough to A. In Levenshtein distance

a threshold $k < m$ is chosen and one reports all the endpoints of substrings of B that are at (transposition invariant) distance at most k from A. This is easily handled by setting $D(0, j) = 0$ for all j and produces no effect in our algorithms. We have also to consider that each computed cell $D(i, j)$ induces cost $D(m, j') = D(i, j) + (m - i)$ at columns $j \leq j' \leq j + (m - i)$, and $D(i, j + (m-i)+c) = D(i, j)+(m-i)+c$ for $c > 0$. Hence, for each cell $D(i, j)$ we compute, we may mark an interval $j \leq j' \leq j + d$ of occurrence positions in B, that is, where $D(m, j') \leq k$. This marking, and remotion of duplicated occurrences, can easily be handled in overall $O(n)$ time [12]. Hence the search complexity stays at $O(mn\sqrt{\sigma})$. The matrix should be filled column by column to guarantee $O(m\sigma)$ space. To find substrings of B with long enough LCS to A we use the *indel distance*, which is the dual of the LCS, $id(A, B) = m + n - 2 \cdot LCS(A, B)$. The indel distance can be dealt with essentially as the LCS computation and permits searching in $O(mn \log \sigma)$ time.

It is not clear whether it is possible to achieve $O(mn)$ time on transposition invariant distance computation, so that no penalty is paid for the transposition invariance. No useful lower bounds are known. It would also be interesting to try to extend our algorithms to other more complex distances of interest in music retrieval, such as weighted edit distances. We have made use of particular properties of the LCS and Levenshtein distance to obtain our complexities, so it is not clear what can be achieved for other distances.

References

1. H. Bunke and J. Csirik. An improved algorithm for computing the edit distance of run-length coded strings. *Information Processing Letters*, 54(2):93–96, 1995.
2. T. Crawford, C. Iliopoulos, and R. Raman. String matching techniques for musical similarity and melodic recognition. *Computing in Musicology* 11:71–100, 1998.
3. M. Crochemore, C. Iliopoulos, Y. Pinzon, and J. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. *Information Processing Letters*, 80(6):279–285, 2001.
4. M. Crochemore, G. Landau, and M. Ziv-Ukelson. A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. In *Proc. SODA 2002*, pp. 679–688. ACM-SIAM, 2002.
5. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
6. J. Hunt and T. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, May 1977.
7. K. Lemström, G. Navarro, and Y. Pinzon. Practical algorithms for transposition-invariant string matching. *Journal of Discrete Algorithms (JDA)*, 2004. Elsevier Science. To appear. Abstract in *Proc. SPIRE'04*, LNCS, to appear.
8. K. Lemström and J. Tarhio. Searching monophonic patterns within polyphonic sources. In *Proc. RIAO 2000*, pp. 1261–1279 (vol 2), 2000.
9. K. Lemström and E. Ukkonen. Including interval encoding into edit distance based music comparison and retrieval. In *Proc. AISB 2000*, pp. 53–60, 2000.
10. V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady* 6:707–710, 1966.

11. V. Mäkinen, G. Navarro, and E. Ukkonen. Approximate matching of run-length compressed strings. *Algorithmica* 35:347–369, 2003.
12. V. Mäkinen, G. Navarro, and E. Ukkonen. Algorithms for transposition invariant string matching. In *Proc. STACS'03*, LNCS 2607, pp. 191–202, 2003. Full version as Technical Report TR/DCC-2002-5, Dept. of Comp. Science, Univ. of Chile, July 2002, `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/ti_matching.ps.gz`. To appear in *Journal of Algorithms*.
13. P. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms*, 1(4):359–373, 1980.