Algorithms for Transposition Invariant String Matching (Extended Abstract)

Veli Mäkinen[1]*, Gonzalo Navarro[2]**, and Esko Ukkonen[1]*

[1] Department of Computer Science, P.O Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, Finland.
{vmakinen,ukkonen}@cs.helsinki.fi

[2] Center for Web Research, Department of Computer Science, University of Chile
Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl

Abstract. Given strings A and B over an alphabet $\Sigma \subseteq \mathbb{U}$, where $\mathbb{U}$ is some numerical universe closed under addition and subtraction, and a distance function $d(A, B)$ that gives the score of the best (partial) matching of A and B, the *transposition invariant distance* is $\min_{t\in\mathbb{U}}\{d(A+t, B)\}$, where $A + t = (a_1 + t)(a_2 + t)\ldots(a_m + t)$. We study the problem of computing the transposition invariant distance for various distance (and similarity) functions d, that are different versions of the edit distance. For all these problems we give algorithms whose time complexities are close to the known upper bounds without transposition invariance. In particular, we show how sparse dynamic programming can be used to solve transposition invariant problems.

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, 11, 12]. 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. Often the durations are omitted, too, since it is usually possible to recognize the melody from a sequence of pitches. In this paper, we study distance measures for pitch sequences (of monophonic music) and their computation.

In general, edit distance measures can be used for matching two pitch sequences. There are, however, a couple of properties related to music that should

* Supported by the Academy of Finland under grant 22584.

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

be taken into account. Transposition invariance is one of those; the same melody is perceived even if the pitch sequence is shifted from one key to another. Another property is the continuity of the alignment; the size of the gaps between matches should be limited, since one long gap in a central part can make a crucial difference in perception. Small gaps should be accepted, e.g. for removing "decorative" notes.

We study how these two properties can be embedded in evaluating the edit distance measures. The summary of our results is given in Section 3.

In the full version of this paper [15] we also study non-gapped measures of similarity under transposition invariance (like Hamming distance), that have other applications besides MIR. Also a more error-tolerant version of edit distances is studied there, where two characters can match if their distance is at most a given constant δ.

2 Definitions

Let Σ be a finite numerical alphabet, which is a subset of some universe $\mathbb{U}$ that is closed under addition and subtraction. Let $A = a_1a_2 \ldots a_m$ and $B = b_1b_2 \ldots b_n$ be two *strings* over Σ^*, i.e. the *symbols* (*characters*) a_i, b_j of the two strings are in Σ for all $1 \leq i \leq m, 1 \leq j \leq n$. We will assume w.l.o.g. that $m \leq n$, since the distance measures we consider are symmetric. String A' is a *substring* of A if $A' = A_{i \ldots j} = a_i \ldots a_j$ for some $1 \leq i \leq j \leq m$. 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 following measures can be defined. The length of the *longest common subsequence (LCS)* of A and B is $\mathrm{lcs}(A, B) = \max\{|S| \mid S \sqsubseteq A, S \sqsubseteq B\}$. The *edit distance* [13, 18, 16] between A and B is the minimum number of edit operations that are needed to convert A into B. Particularly, in the unit cost *Levenshtein distance* d_{L} the set of edit operations consists of insertions $\epsilon \to b$, deletions $a \to \epsilon$, and substitutions $a \to b$ $(a \neq b)$ of one alphabet symbol; here $a, b \in \Sigma$, ϵ is the empty string, and *identity operations* (*matches*) $a \to a$ are not charged. If the substitution operation is forbidden, we get a distance d_{ID}, which is actually a dual problem of evaluating the LCS; it is easy to see that $d_{\mathrm{ID}}(A, B) = m + n - 2 \cdot \mathrm{lcs}(A, B)$. For convenience, we will mainly use the minimization problem d_{ID} (not lcs) in the sequel. If only deletion for symbols of B are allowed, we get a distance d_{D}.

String A is a *transposed copy* of B (denoted by $A =^t B$) if $B = (a_1 + t)(a_2 + t) \cdots (a_m + t) = A + t$ for some $t \in \mathbb{U}$. Definitions for a transposed substring and a transposed subsequence can be stated similarly. The transposition invariant versions of the above distance measures d_* where $* \in \{\mathrm{L}, \mathrm{ID}, \mathrm{D}\}$ can now be defined as $d_*^{\mathrm{t}}(A, B) = \min_{t \in \mathbb{U}} d_*(A + t, B)$.

We also define α–limited versions of the edit distance measures, where the distance (gap) between two consecutive matches is limited by a constant $\alpha > 0$, i.e. if $(a_{i'}, b_{j'})$ and (a_i, b_j) are consecutive matches, then $|i - i' - 1| \leq \alpha$ and $|j - j' - 1| \leq \alpha$. We get distances $d_{\mathrm{L}}^{\mathrm{t},\alpha}$, $d_{\mathrm{ID}}^{\mathrm{t},\alpha}$, and $d_{\mathrm{D}}^{\mathrm{t},\alpha}$.

The *approximate string matching problem*, based on the above distance functions, is to find the minimum distance between A and any substring of B. In this case we call A the *pattern* and denote it $P_{1...m} = p_1 p_2 \cdots p_m$, and call B the *text* and denote it $T_{1...n} = t_1 t_2 \cdots t_n$, and usually assume that $m << n$. The *exact string matching problem* is a special case, where all the exact occurrences of P in T are searched. The two string matching problems are also called *search problems*.

In particular, if distance d_{D} is used in approximate string matching, we obtain a problem known as *episode matching* [14, 5]. It can also be stated as follows: Find the shortest substring of the text that contains the pattern as a subsequence.

Our complexity results are different depending on the form of the alphabet Σ. We will distinguish two cases. An *integer* alphabet is any alphabet $\Sigma \subset \mathbb{Z}$. For integer alphabets, $|\Sigma|$ will denote $\max(\Sigma) - \min(\Sigma) + 1$. A *real* alphabet will be any other $\Sigma \subseteq \mathbb{R}$. For any string $A = a_1 \ldots a_m$, we will call $\Sigma_A = \{a_i \mid 1 \leq i \leq m\}$ the alphabet of A.

3 Related Work and Summary of Our Results

The first thing to notice is that the problem of *exact* transposition invariant string matching is extremely easy to solve; one can use the relative encoding of both the pattern ($p'_1 = p_2 - p_1, p'_2 = p_3 - p_2, \ldots$) and the text ($t'_1 = t_2 - t_1, t'_2 = t_3 - t_2, \ldots$), and use the whole arsenal of the methods developed for exact string matching. Unfortunately, this relative encoding seems to be of no use when the exact comparison is replaced by an approximate one.

Transposition invariance (as far as we know) was introduced in the string matching context in the work of Lemström and Ukkonen [12]. They proposed transposition invariant longest common subsequence (LCTS) as a measure of similarity between two monophonic music (pitch) sequences. They gave a descriptive nick name for the measure: "Longest Common Hidden Melody". As the alphabet of pitches is some limited integer alphabet $\Sigma \subset \mathbb{Z}$, the transpositions that have to be considered are $\mathbb{T} = \{b - a \mid a, b \in \Sigma\}$. This gives a brute force algorithm for computing the length of the LCTS [12]: Compute $\mathrm{lcs}(A + t, B)$ using $O(mn)$ dynamic programming for each $t \in \mathbb{T}$. The runtime of this algorithm is $O(|\Sigma| mn)$, where typically $|\Sigma| < 256$. In the general case, where Σ could be unlimited, one could instead use the set of transpositions $\mathbb{T}' = \{b - a \mid a \in A, b \in B\}$. This is because some characters must match in any meaningful transposition. The size of $\mathbb{T}'$ could be mn, which gives $O(m^2 n^2)$ worst case time for real alphabets. Thus it is both of practical and theoretical interest to improve this algorithm.

We will also use a brute force approach as described above, but since most transpositions produce sparse instances of the dynamic programming problem, we can use specialized sparse dynamic programming algorithms to get good worst case bounds. Moreover, we show a connection between the resulting sparse dynamic programming problems and semi-static range minimum queries. We obtain simple yet efficient algorithms for the edit distance measures.

For LCS (and thus for d_{ID}) there already exists Hunt-Szymanski [10] type (sparse dynamic programming) algorithms whose time complexities depend on the number r of matching character pairs between the compared strings. The complexity of the Hunt-Szymanski algorithm is $O(r \log m)$ after time $O(n \log m)$ for preprocessing. As we show that the sum of values r over all different transpositions is mn, we get the bound $O(mn \log m)$ for the transposition invariant LCS. This requires constructing the sets of matching character pairs for each transposition; we show that this preprocessing can be done in time $O(mn)$ plus an additive term that depends on the alphabet size. Using improved algorithms for LCS [1, 7] yield $O(mn \log \log n)$ bound for the transposition invariant LCS. We improve this to $O(mn \log \log m)$ by giving a new sparse dynamic programming algorithm for LCS. Our algorithm can also be generalized to the case where gaps are limited by a constant α, giving time $O(mn \log(\alpha m))$ for evaluating $d_{\mathrm{ID}}^{\mathrm{t},\alpha}(A, B)$.

Eppstein et al. [7] have proposed sparse dynamic programming algorithms for more complex distance computations such as the Wilbur-Lipman fragment alignment problem [19]. Also the unit cost Levenshtein distance can be evaluated using these techniques [9]. Using this algorithm, the transposition invariant case can be solved in time $O(mn \log \log n)$. However, the algorithm does not generalize to the case of α-limited gaps, and thus we develop an alternative solution that is based on semi-static range minimum queries. This gives us $O(mn \log^2 n \log \log m)$ for evaluating $d_{\mathrm{L}}^{\mathrm{t},\alpha}(A, B)$.

Finally, we give a new sparse dynamic programming algorithm with running time $O(r)$ for episode matching. This further gives us an $O(mn)$ method for the transposition invariant episode matching.

Table 1 compares our results for transposition invariant problems with the known upper bounds for these problems without transposition invariance.

Table 1. Upper bounds for string matching without and with transposition invariance. We have not added, for clarity, the preprocessing times of Lemma 2, which depend on alphabet terms. In some rare cases on real alphabet the preprocessing might take $O(mn \log m)$. The bounds are given for distance evaluation, unless otherwise stated. The corresponding bounds for the transposition invariant search problems are obtained by replacing the factors $\log n$, $\log \log n$ and $\log(\alpha m)$ with $\log m$, $\log \log m$, and $\log m$, respectively.

distance	without transposition invariance	with transposition invariance
exact (searching)	$O(m+n)$	$O(m+n)$
d_{ID}	$O(mn/\log m)$ [4]	$O(mn \log \log m)$
d_{ID}^{α}	$O(\alpha^2 mn)$ (naive DP)	$O(mn \log(\alpha m))$
d_{L}	$O(mn/\log m)$ [4]	$O(mn \log \log n)$
d_{L}^{α}	$O(\alpha^2 mn)$ (naive DP)	$O(mn \log^2 n \log \log m)$
d_{D} (searching)	$O(mn/\log m)$ [5]	$O(mn)$
d_{D}^{α} (searching)	$O(mn)$ [6, 3]	$O(mn)$

4 Computation of Transposition Invariant Edit Distances

Let us first review how the edit distances can be computed using dynamic programming [13, 18, 16]. Let $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$. One can evaluate an $(m+1) \times (n+1)$ matrix (d_{ij}), $0 \le i \le m, 0 \le j \le n$, using the recurrence

$$d_{i,j} = \min(d_{i-1,j-1} + w(a_i \to b_j), d_{i-1,j} + w(\epsilon \to a_i), d_{i,j-1} + w(b_j \to \epsilon)), \quad (1)$$

with initialization $d_{i,0} = i$ for $0 \le i \le m$ and $d_{0,j} = j$ for $0 \le j \le n$. Weight function w gives the costs for the different edit operations. Matrix (d_{ij}) can be evaluated (in some suitable order, like row–by–row or column–by–column) in $O(mn)$ time, each value $d_{i,j}$ giving the weighted edit distance of $A_{1\ldots i}$ and $B_{1\ldots j}$. Especially, value d_{mn} equals the weighted edit distance of A and B.

Algorithms for distances d_{ID}, d_{L}, and d_{D} can now be stated by giving proper weights for the operations. For d_{ID} we have weights $w(a_i \to b_j) = (\text{ if } a_i = b_j \text{ then } 0 \text{ else } \infty)$, and $w(\epsilon \to b_j) = w(a_i \to \epsilon) = 1$. For d_{L}, the above substitution cost ∞ is replaced by 1. For d_{D}, other costs are as in d_{ID}, but $w(a_i \to \epsilon) = \infty$.

The corresponding search problems can be solved by assigning zero to the values in the first row of (d_{ij}) (recall that we identify pattern $P = A$ and text $T = B$). To find the best approximate match, we take $\min_{0 \le j \le n} d_{m,j}$.

4.1 The Relation between Sparseness and Transposition Invariance

Let M be the set of matching character pairs between strings A and B, i.e. $M = M(A,B) = \{(i,j) \mid a_i = b_j, 1 \le i \le m, 1 \le j \le n\}$. Let $r = r(A,B) = |M(A,B)|$. Let us redefine $\mathbb{T}$ to be the set of those transpositions that make some characters match between A and B, that is $\mathbb{T} = \{b_j - a_i \mid 1 \le i \le m, 1 \le j \le n\}$. One could compute the above edit distances and solve the search problems by running the corresponding recurrences (1) over all pairs $(A+t, B)$, where $t \in \mathbb{T}$. On integer alphabet this takes $O(|\Sigma| mn)$ time, and $O(|\Sigma_A||\Sigma_B| mn)$ time on real alphabet. This kind of a procedure can be significantly speeded up by using suitable "sparse dynamic programming" algorithms, instead.

Lemma 1 *If an algorithm computes a distance $d(A,B)$ in $O(g(r(A,B))f(m,n))$ time, where g is a concave function, then the transposition invariant distance $d^{\mathrm{t}}(A,B) = \min_{t \in \mathbb{T}} d(A+t, B)$ can be computed in $O(g(mn)f(m,n))$ time, by repeating the algorithm for all $t \in \mathbb{T}$.*

Proof. Let $r_t = r(A+t, B)$ be the number of matching character pairs between $A+t$ and B. Then

$$\sum_{t \in \mathbb{T}} g(r_t) f(m,n) = f(m,n) \sum_{t \in \mathbb{T}} g\left(\sum_{i=1}^{m} |\{j \mid a_i + t = b_j, 1 \le j \le n\}|\right)$$
$$\le f(m,n) g\left(\sum_{i=1}^{m} \sum_{t \in \mathbb{T}} |\{j \mid a_i + t = b_j, 1 \le j \le n\}|\right) = f(m,n) g(mn). \qquad \square$$

The rest of the section devotes to developing algorithms whose running times depend on r.

4.2 Preprocessing

As a first step, we need a way of constructing the match set M sorted in an order that enables sparse evaluation of matrix (d_{ij}). We use *column–by–column order* defined as follows: $j' < j$ or ($j' = j$ and $i' > i$).[3] The match set corresponding to a transposition t will be called $M_t = \{(i,j) \mid a_i + t = b_j\}$.

Lemma 2 *The* match sets $M_t = \{(i,j) \mid a_i + t = b_j\}$*, each sorted in column–by–column order, for all transpositions* $t \in \mathbb{T}$*, can be constructed for an integer alphabet in time* $O(|\Sigma| + mn)$*, and for a real alphabet in time* $O(m \log |\Sigma_A| + n \log |\Sigma_B| + |\Sigma_A||\Sigma_B| \log(\min(|\Sigma_A|, |\Sigma_B|)) + mn)$*. Both bounds can be achieved using* $O(mn)$ *space.*

Proof. (Sketch) For an integer alphabet, $O(|\Sigma| + mn)$ time is obtained by proceeding naively and by using array indexing to get M_t to which each pair (i,j) has to be added.

For a real alphabet, we can construct in $O(m \log |\Sigma_A| + n \log |\Sigma_B|)$ time two binary search trees $\mathcal{T}_A$ and $\mathcal{T}_B$ for characters in A and B, respectively, where each leaf maintains a list of consecutive positions in A (or in B) where that character occurs. Now, consider matching a leaf (character a) of $\mathcal{T}_A$ with a leaf (character b) of $\mathcal{T}_B$. Then, the product of the corresponding position lists gives the part of the match set M_{b-a}, where transposition $t = b - a$ is caused by matching a with b. The same transposition can be caused by several pairs of characters. Thus, when all the $|\Sigma_A||\Sigma_B|$ partial match sets are produced, we can merge the sets that correspond to the same transposition. This can be done in $O(|\Sigma_A||\Sigma_B| \log(\min(|\Sigma_A|, |\Sigma_B|)) + mn)$ time: First, sort the set of partial match sets based on the transposition they belong to by merging $\min(|\Sigma_A|, |\Sigma_B|)$ lists that are already in order (e.g. for each fixed $a \in \Sigma_A$, transpositions $b - a$, $b \in \Sigma_B$, are obtained in increasing order from $\mathcal{T}_B$). This produces a sequence of partial match sets, where the consecutive sets having the same transposition should be merged. Merging can be done efficiently by traversing B and adding a new column j to each match set $M_{b_j - a}$, $a \in \Sigma_A$; each such match set can be found in constant time by maintaining suitable pointers during the sorting phase. □

The preprocessing can be made more space-efficient with some penalty in the time requirement; the details can be found in [15].

4.3 Computing the Transposition Invariant Longest Common Subsequence

The fastest existing sparse dynamic programming algorithm for LCS is due to Eppstein et al. [7]. Improving an algorithm of Apostolico and Guerra [1], they achieved running time $O(d \log \log \min(d, \frac{mn}{d}))$, where $d \leq r$ is the number of dominant matches (see, e.g. [1] for a definition). Using this algorithm with Lemma 1, we get the bound $O(mn \log \log n)$ for the transposition invariant case.

[3] Note that our definition differs from the usual column–by–column order in condition $i' > i$. This is to simplify the algorithms later on.

The existing algorithms do not, however, extend to the case of α–limited gaps. We will give here an efficient algorithm for LCS that generalizes to this case. We will also use the same technique for developing an efficient algorithm for the Levenshtein distance with α–limited gaps. Moreover, by replacing the data structure used in this algorithm by a more efficient one, we can achieve $O(r \log\log m)$ complexity, which gives $O(mn \log\log m)$ for the transposition invariant LCS (which is better than the previous bound, since $m \leq n$).

Recall the set of matching character pairs $M = \{(i,j) \mid a_i = b_j\}$. Let $\bar{M} = M \cup \{(0,0),(m+1,n+1)\}$. Using basic properties of the dynamic programming matrix related to d_{ID} one obtains the following sparsity lemma.

Lemma 3 *Distance $d_{\mathrm{ID}}(A,B)$ can be computed by evaluating $d_{i,j}$ for $(i,j) \in \bar{M}$ using the recurrence*

$$d_{i,j} = \min\{d_{i',j'} + i - i' + j - j' - 2 \mid (i',j') \in \bar{M}, i' < i, j' < j\}, \qquad (2)$$

with initialization $d_{0,0} = 0$. Value $d_{m+1,n+1}$ equals $d_{\mathrm{ID}}(A,B)$.

The obvious strategy to use the above lemma is to keep the already computed values $d_{i',j'}$ in a data structure such that their minimum can be retrieved efficiently when computing the value of the next $d_{i,j}$. One difficulty here is that the values stored are not comparable as such since we want the minimum just after $i - i' + j - j' - 2$ is added. This can be solved by storing the *path invariant* values $d_{i',j'} - i' - j'$ instead. Then, after retrieving the minimum value, one can add $i + j - 2$ to get the correct value for $d_{i,j}$. To get the minimum value $d_{i',j'} - i' - j'$ from range $(i',j') \in [-\infty, i) \times [-\infty, j)$, we need a dynamic data structure supporting one-dimensional range queries (our column–by–column traversal order guarantees that all query points are in range $[-\infty, j)$). In addition, the range query should not be output sensitive; it should only report the minimum value, not all the points in the range.

A balanced binary tree can be used as a such data structure. We can use the row number i' as a sort key, and store values $d(i',j') - i' - j'$ in the leaves. Then we can maintain in each internal node the minimum of the values $d(i',j') - i' - j'$ in its subtree. This is summarized in the following well-known lemma.

Lemma 4 *A* balanced binary tree $\mathcal{T}$ *supports the following operations in $O(\log n)$ amortized time, where n is the amount of elements currently in the tree.*

$p = \mathcal{T}.Insert(k,v)$ **:** *Inserts value v into the tree with key k, and returns a pointer p to the inserted leaf.*

$\mathcal{T}.Delete(p)$ **:** *Deletes the leaf pointed by p.*

$v = \mathcal{T}.Minimum(I)$ **:** *Returns the minimum of values that have key in the one-dimensional range I.*

We are ready to give the algorithm. Initialize a balanced binary tree $\mathcal{T}$ by adding the value of $d_{0,0} - i - j = 0$ with key $i = 0$: $\mathcal{T}.Insert(0,0)$. Proceed with the match set $\bar{M} \setminus \{(0,0)\}$ that is sorted in column–by–column order and make the following operations at each pair (i,j):

(1) Take the minimum value from $\mathcal{T}$ whose key is smaller than the current row number i: $d = \mathcal{T}.Minimum([-\infty, i))$. Add $i + j - 2$ to this value: $d \leftarrow d + i + j - 2$.
(2) Add the current value d minus the current row number i and current column number j into $\mathcal{T}$ with the current row number as a key: $\mathcal{T}.Insert(i, d-i-j)$.

Finally, after cell $(m+1, n+1)$ has been processed, we have that $d_{\mathrm{ID}}(A, B) = d$.

The above algorithm works correctly: The column–by–column evaluation and the range query restricted by the row number in $\mathcal{T}$ guarantee that the $i' < i$ and $j' < j$ conditions hold. Clearly, the time complexity is $O(r \log r)$.

The queries that we use are semi-infinite. It follows that we can use the result in Lemma 7 (see next section) to obtain $\log\log m$ query time, since our query range is $[0, m]$.

Let us now consider the case with α–limited gaps. The only change we need in our algorithms is to make sure that, in order to compute $d_{i,j}$, we only take into account the matches that are in the range $(i', j') \in [i - \alpha - 1, i) \times [j - \alpha - 1, j)$. What we need to do is to change the range $[-\infty, i)$ into $[i - \alpha - 1, i)$ in $\mathcal{T}$, as well as to delete elements in column $\leq j - \alpha - 1$ after processing elements in column j. The former is easily accomplished by using query $\mathcal{T}.Minimum([i - \alpha - 1, i))$ at phase (1) of the algorithm. The latter needs an auxiliary tree organized by values j, having pointers to the corresponding leaves of the primary tree; then we can use $\mathcal{T}.Delete(p)$ to delete each element p found by a query $[-\infty, j - \alpha - 1]$ from the auxiliary tree. Notice that we can not use the reduction to $\log\log m$ anymore, since the query ranges are no longer semi-infinite. However, since there are at most αm values stored at a time in $\mathcal{T}$, we have $O(\log(\alpha m))$ query time.

By using Lemma 1 and the above algorithms, we get the following result.

Theorem 5 *The transposition invariant distance $d^{\mathrm{t}}_{\mathrm{ID}}(A, B)$ can be computed in $O(mn \log\log m)$ time and $O(mn)$ space. The corresponding search problem can be solved in $O(mn \log\log m)$ time and in $O(m^2)$ space. For the distance $d^{\mathrm{t},\alpha}_{\mathrm{ID}}(A, B)$, the space requirements remain the same, but the time bounds are $O(mn \log(\alpha m))$ for distance computation and $O(mn \log m)$ for searching. The preprocessing bounds of Lemma 2 need to be added to these bounds.*

We refer to [15] for details on how to achieve the above bounds for the search problems.

4.4 Computing the Transposition Invariant Levenshtein Distance

The Levenshtein distance d_{L} has a sparsity property similar to the one given for d_{ID} in Lemma 3. Recall that $\bar{M} = M \cup \{(0, 0), (m + 1, n + 1)\}$, where M is the set of matching character pairs. The following lemma is easy to prove.

Lemma 6 *Distance $d_{\mathrm{L}}(A, B)$ can be computed by evaluating $d_{i,j}$ for $(i, j) \in \bar{M}$ using the recurrence*

$$d_{i,j} = \min \begin{cases} \{d_{i',j'} + j - j' - 1 \mid (i', j') \in \bar{M}, i' < i, j' - i' < j - i\} \\ \{d_{i',j'} + i - i' - 1 \mid (i', j') \in \bar{M}, j' < j, j' - i' \geq j - i\} \end{cases} \qquad (3)$$

with initialization $d_{0,0} = 0$*. Value* $d_{m+1,n+1}$ *equals* $d_{\mathrm{L}}(A, B)$*.*

This relation is more complex than the one for d_{ID}. In the case of d_{ID} we could store values $d_{i',j'}$ in a comparable format (by storing $d_{i',j'} - i' - j'$ instead) so that the minimum from index range $i' < i, j' < j$ could be retrieved efficiently. For d_{L} there does not seem to be such a comparable format, since the path length from (i', j') to (i, j) may be either $i - i' - 1$ or $j - j' - 1$.

Let us call the two index ranges for i' and j' in the above lemma as the *lower region* $(i' < i, j' - i' < j - i)$ and the *upper region* $(j' < j, j' - i' \geq j - i)$. Our strategy is to maintain separate data structures for both regions. Each value $d_{i',j'}$ will be stored in both structures in such a way that the stored values in each structure are comparable.

Let $\mathcal{L}$ denote the data structure for the lower region and $\mathcal{U}$ the data structure for the upper region. If we store values $d_{i',j'} - j'$ in $\mathcal{L}$, we can take the minimum over those values plus $j - 1$ to get the value of $d_{i,j}$. However, we want this minimum over a subset of values stored in $\mathcal{L}$, i.e. over those $d_{i',j'} - j'$ whose coordinates satisfy $i' < i, j' - i' < j - i$. Similarly, if we store values $d_{i',j'} - i'$ in $\mathcal{U}$, we can take minimum over those values whose coordinates satisfy $j' < j, j' - i' \geq j - i$, plus $i - 1$ to get the value of $d_{i,j}$ (the actual minimum is then the minimum of upper region and the lower region).

What is left to be explained is how the minima of subsets of $\mathcal{L}$ and $\mathcal{U}$ can be obtained. For the upper region, we can use the same structure as for d_{ID}; if we keep values $d_{i',j'} - i'$ in a balanced binary tree $\mathcal{U}$ with keys $j' - i'$, we can make one-dimensional range search to locate the minimum of values $d_{i',j'} - i'$ whose coordinates satisfy $j' - i' \geq j - i$. The column–by–column traversal guarantees that $\mathcal{U}$ only contains values $d_{i',j'} - i'$ for whose coordinates hold $j' < j$. Thus, the upper region can be handled efficiently.

The problem now is the lower region. We could use row–by–row traversal to handle this case efficiently, but then we would have the symmetric problem with the upper region. No traversal order will allow us to limit to one-dimensional range searches in both regions simultaneously; we will need two-dimensional range searches in one of them. Let us consider the two-dimensional range search for the lower region. We would need a query that retrieves the minimum of values $d_{i',j'} - j'$ whose coordinates satisfy $i' < i, j' - i' < j - i$. We make a coordinate transformation to make this triangle region into a rectangle; we map each value $d_{i',j'} - j'$ into an xy-plane to coordinate $i', j' - i'$. What we need in this plane, is a rectangle query $[-\infty, i) \times [-\infty, j - i)$, i.e. a semi-infinite query in both coordinates. We will specify in Lemma 7 an abstract data structure for $\mathcal{L}$ that supports this operation. Such a structure, given in [8], is basically a two-dimensional range tree, such that the secondary trees are replaced by priority queues. The underlying idea is that one-dimensional range minimum queries can be answered using a priority queue. Moreover, these queries can be done in the rank space, and efficient implementations exist for the required operations on an integer range [17].

Lemma 7 ([8]) *There is a data structure $\mathcal{R}$ that, after $O(n \log n)$ time preprocessing, supports the following operations in amortized $O(\log n \log\log n)$ time and $O(n \log n)$ space, where n is the number of elements in the structure:*

$\mathcal{R}.Update(x, y, v)$ **:** *Update value at coordinate x, y to v (under the condition that the current value must be larger than v).*

$v = \mathcal{R}.Minimum(I)$ **:** *Retrieve the minimum of values from range I, where I is semi-infinite in both coordinates.*

In the one-dimensional case, where the keys are in the integer range $[1, u]$, semi-infinite queries can be answered in $O(\log\log u)$ time.

We are now ready to give a sparse dynamic programming algorithm for the Levenshtein distance. Initialize a balanced binary tree $\mathcal{U}$ for the upper region by adding the value of $d_{0,0} - i = 0$ with key $i = 0$: $\mathcal{U}.Insert(0, 0)$. Initialize a data structure $\mathcal{L}$ for the lower region ($\mathcal{R}$ of Lemma 7) with the triples (i, j, ∞) such that $(i, j) \in \bar{M}$. Update value of $d_{0,0} - j = 0$ with keys $i = 0$ and $j - i = 0$: $\mathcal{L}.Update(0, 0, 0)$. Proceed with the match set $\bar{M} \setminus \{(0, 0)\}$ that is sorted in column–by–column order and make the following operations at each pair (i, j):

(1) Take the minimum value from $\mathcal{U}$ whose key is larger or equal to the current diagonal $j - i$: $d' = \mathcal{U}.Minimum([j - i, \infty])$. Add $i - 1$ to this value: $d' \leftarrow d' + i - 1$.
(2) Take the minimum value from $\mathcal{L}$ inside the rectangle $[-\infty, i) \times [-\infty, j - i)$: $d'' = \mathcal{L}.Minimum([-\infty, i) \times [-\infty, j - i))$. Add $j - 1$ to this value: $d'' \leftarrow d'' + j - 1$.
(3) Choose the minimum of d' and d'' as the current value $d = d_{i,j}$.
(4) Add the current value d minus i into $\mathcal{U}$ with key $j - i$: $\mathcal{U}.Insert(j - i, d - i)$.
(5) Add the current value d minus j into $\mathcal{L}$ with keys i and $j - i$: $\mathcal{L}.Update(i, j - i, d - j)$.

Finally, after cell $(m + 1, n + 1)$ has been processed, we have that $d_{\mathrm{L}}(A, B) = d$.

The correctness of the algorithm should be clear from the above discussion. The time complexity is $O(r \log r \log\log r)$ (r elements are inserted and updated in the lower region structure, and r times it is queried). The space usage is $O(r \log r)$. We can reduce the time complexity to $O(r \log r \log\log m)$ since the $\log\log n$ factor in Lemma 7 is actually $\log\log u$, where $1 \ldots u$ is the the range of values added to the secondary structure. We can implement the structure in Lemma 7 so that $u = m$.

Using this algorithm, the transposition invariant distance can be evaluated in $O(mn \log n \log\log m)$ time. This is, by a $\log n$ factor, worse than what can be achieved by using an $O(r \log\log \min(r, mn/r))$ time algorithm of Eppstein et al. [7].

However, the advantage of our range query approach is that we can now easily evaluate the distance with α–limited gaps. Consider the lower region. We need the minimum over the values whose coordinates (i', j') satisfy $i' \in [i - \alpha - 1, i)$, $j' \in [j - \alpha - 1, j)$, and $j' - i' \in [-\infty, j - i)$. We map each $d_{i',j'} - j'$ into three-dimensional coordinate $(i', j', j' - i')$. The data structure of Lemma 7 can be generalized to answer three-dimensional range minimum queries when the ranges are semi-infinite in each coordinate [8]. We can use query $\mathcal{R}.Minimum([-\infty,i)\times[j-\alpha-1,\infty]\times[-\infty,j-i))$ when computing the value of $d_{i,j}$ from the lower region, since $i'\geq i-\alpha-1$ when $j'-i'<j-i$, and column–by–column order guarantees that $j'<j$. The upper region case is now symmetric and can be handled similarly. The data structure $\mathcal{R}$ can be implemented so that we get overall time complexity $O(r\log^2 r\log\log m)$.

Using Lemma 1 with the above algorithms (and with the algorithm of Eppstein et al.), we obtain the following result for the transposition invariant case.

Theorem 8 *Transposition invariant Levenshtein distance $d_{\mathrm{L}}^{\mathrm{t}}(A,B)$ can be computed in $O(mn\log\log n)$ time and in $O(mn)$ space. The corresponding search problem can be solved in $O(mn\log\log m)$ time and $O(m^2)$ space. For the distance $d_{\mathrm{L}}^{\mathrm{t},\alpha}(A,B)$ the time requirements are $O(mn\log^2 n\log\log m)$ and $O(mn\log^2 m\log\log m)$, and space requirements $O(mn\log^2 n)$ and $O(m^2\log^2 m)$, respectively, for distance computation and for searching. The preprocessing bounds of Lemma 2 need to be added to these bounds.*

Again, we refer to [15] for details on how to achieve the above bounds for the search problems.

4.5 Transposition Invariant Episode Matching

Finally we look at the episode matching problem and the $d_{\mathrm{D}}^{\mathrm{t}}$ distance, which has a simple sparse dynamic programming solution. Recall that $\bar{M}=M\cup\{(0,0),(m+1,n+1)\}$, where M is the set of matching character pairs. The following lemma for d_{D} is easy to prove.

Lemma 9 *Distance $d_{\mathrm{D}}(A,B)$ can be computed by evaluating $d_{i,j}$ for $(i,j)\in\bar{M}$ using the recurrence*

$$d_{i,j}=\min\{d_{i-1,j'}+j-j'-1\mid(i-1,j')\in\bar{M}\},\tag{4}$$

with initialization $d_{0,0}=0$. Value $d_{m+1,n+1}$ equals $d_{\mathrm{D}}(A,B)$.

Consider an algorithm that traverses the match set $\bar{M}$ in the column–by–column order. We will maintain for each row a value $c(i)$ that gives the largest $j'<j$ such that $a_i=b_{j'}$, and a value $d(i)=d_{i,j'}$. First, initialize these values to ∞, except that $c(0)=0$ and $d(0)=0$. Let $(i,j)\in\bar{M}$ be the current pair whose value we need to evaluate. Then $d=d_{i,j}=d(i-1)+j-c(i-1)-1$. We can now update the values of the current row: $c(i)=j$ and $d(i)=d$. This takes overall $O(|\bar{M}|)=O(|M|)$ time. (Preprocessing time $O(n\log|\Sigma_A|+|M|)$ for constructing M needs to be added to this.)

The above algorithm generalizes to the search problem and to the episode matching problem by implicitly initializing values $c(0)=j-1$ and $d(0)=0$ for the values in the first row. The problem of α–limited gaps can also be handled easily; we simply avoid updating $d(i)$ as defined when $j-c(i-1)-1>\alpha$. In this case we set $d(i)=\infty$. Using Lemma 1 we get:

Theorem 10 *The transposition invariant episode matching problem can be solved in $O(mn)$ time. The same bound applies in the case of α–limited gaps. The preprocessing bounds of Lemma 2 need to be added to these bounds.*

References

1. A. Apostolico and C. Guerra. The longest common subsequence problems revisited. *Algorithmica* 2:315–336, 1987.
2. T. Crawford, C.S. 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, C. Makris, W. Rytter, A. Tsakalidis, and K. Tsichlas. Approximate string matching with gaps. *Nordic Journal of Computing* 9(1):54–65, Spring 2002.
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. G. Das, R. Fleischer, L. Gasieniec, D. Gunopulos, and J. Kärkkäinen. Episode matching. In *Proc. CPM'97*, LNCS 1264, Springer, pp. 12–27, 1997.
6. M.J. Dovey. A technique for "regular expression" style searching in polyphonic music. In *Proc. ISMIR 2001*, pp. 179–185, October 2001.
7. D. Eppstein, Z. Galil, R. Giancarlo, and G. F. Italiano. Sparse dynamic programming I: linear cost functions. *J. of the ACM* 39(3):519–545, July 1992.
8. H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. *Proc. STOC'84*, pp. 135–143, 1984.
9. Z. Galil and K. Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science* 92:49–76, 1992.
10. J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, May 1977.
11. K. Lemström and J. Tarhio. Searching monophonic patterns within polyphonic sources. In *Proc. RIAO 2000* , pp. 1261–1279 (vol 2), 2000.
12. K. Lemström and E. Ukkonen. Including interval encoding into edit distance based music comparison and retrieval. In *Proc. AISB 2000*, pp. 53–60, 2000.
13. V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady* 6:707–710, 1966.
14. H. Mannila and H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *Proc. KDD'95*, AAAI Press, pp. 210–215, 1995.
15. V. Mäkinen, G. Navarro, and E. Ukkonen. Algorithms for Transposition Invariant String Matching. *TR/DCC-2002-5*, Dept. of CS, Univ. Chile, July 2002, "ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/ti_matching.ps.gz"
16. P. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *J. of Algorithms*, 1(4):359–373, 1980.
17. P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Proc. Letters* 6(3):80–82, 1977.
18. R. Wagner and M. Fisher. The string-to-string correction problem. *J. of the ACM* 21(1):168–173, 1974.
19. W. J. Wilbur and D. J. Lipman. The contect-dependent comparison of biological sequence. *SIAM J. Appl. Math.* 44(3):557-567, 1984.