Improved Time and Space Complexities for Transposition Invariant String Matching

Gonzalo Navarro [a,1] Szymon Grabowski [b] Veli Mäkinen [c]
Sebastian Deorowicz [d,2]

[a] *Center for Web Research, Dept. of Computer Science, University of Chile, Chile.*

[b] *Computer Engineering Department, Technical University of Łódź, Poland.*

[c] *Department of Computer Science, University of Helsinki, Finland.*

[d] *Institute of Computer Science, Silesian University of Technology, Poland.*

Abstract

Given strings $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$ over a finite alphabet $\Sigma \subset \mathbb{Z}$ of size $O(\sigma)$, and a *distance* $d()$ defined among strings, the *transposition invariant version* of $d()$ is $d^{\mathrm{t}}(A, B) = \min_{t \in \mathbb{Z}} d(A+t, B)$, where $A+t = (a_1+t)(a_2+t)\ldots(a_m+t)$. Distances $d()$ of most interest are Levenshtein distance and indel distance (the dual of the Longest Common Subsequence), which can be computed in $O(mn)$ time. Recent algorithms compute $d^{\mathrm{t}}(A, B)$ in $O(mn \log\log \min(m, n))$ time for those distances. In this paper we show how those complexities can be reduced to $O(mn \log\log \sigma)$. Furthermore, we reduce the space requirements from $O(mn)$ to $O(\sigma^2 + \min(m, n))$.

Key words: longest common subsequence, edit distance, music sequence comparison, transposition invariance, sparse dynamic programming

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.

[1] Supported by Millennium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile.

[2] Supported by Silesian University of Technology Research Project BK-203/RAu2/2004.

Interest in transposition invariant string matching problems has recently arisen in the field of music information retrieval (MIR) [2,6,7]. 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. Therefore, the problem of determining similarity of two strings under transposition invariance is of interest in music retrieval. It also finds applications in time series comparison [1], image comparison [3], and other areas [9].

Let m and n be the lengths of the two strings to compare and σ the size of their alphabets. The basic LCS and Levenshtein distance computation algorithms (without transposition invariance) require $O(mn)$ time and $O(\min(m,n))$ space. In a recent work [11], $O(mn \log\log \min(m,n))$ time algorithms are presented to compute the transposition invariant versions of these distances. Albeit the time penalty over the versions that do not handle transposition invariance is mild, these algorithms require much space, $O(mn)$. In this paper we show how those algorithms can be improved to $O(mn \log\log \sigma)$ time and $O(\sigma^2 + \min(m,n))$ space (on Levenshtein distance) or $O(\sigma + \min(m,n))$ (on LCS). This is an important improvement when the strings to compare are large in comparison to their alphabet, which happens in several applications.

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 $1 \leq i \leq m, 1 \leq j \leq n$.

The Levenshtein distance [8] $ed(A,B)$ between strings A and B is the mini-

mum number of character insertions, deletions, and substitutions, necessary to make them equal. This distance can be computed in $O(mn)$ time [13] time with the classical recurrence:

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

so that $ed(A,B) = D(m,n)$. The indel distance $id(A,B)$ is a variant of the Levenshtein distance where substitutions of characters are forbidden. It is computed in similar fashion in $O(mn)$ time as follows [16]:

$$\begin{array}{rcl} D(i,0) & = & i, \quad D(0,j) \quad = \quad j \\ D(i,j) & = & \textbf{if } a_i = b_j \textbf{ then } D(i-1,j-1) \\ & & \textbf{else } 1 + \min(D(i-1,j), D(i,j-1)), \end{array} \tag{2}$$

so that $id(A,B) = D(m,n)$. The length of the longest common subsequence, $lcs(A,B)$, is an important similarity measure, and it is the dual of the indel distance, that is, $lcs(A,B) = \frac{1}{2}(m+n-id(A,B))$.

A useful alternative formulation of these distance computation problems is to see them as a shortest path problem on a graph. The graph contains one node for each matrix cell. For $id(A,B)$, there are (horizontal) edges of cost 1 that connect every cell $(i,j-1)$ to (i,j), as well as (vertical) edges of cost 1 that connect every cell $(i-1,j)$ to (i,j). Whenever $a_i = b_j$, there is also a (diagonal) zero-cost cell that connects $(i-1,j-1)$ to (i,j). It is not hard to see that $D(m,n)$ is the minimum path cost that connects cell $(0,0)$ to cell (m,n). For $ed(A,B)$ this graph has also diagonal edges of cost 1 from every cell $(i-1,j-1)$ to (i,j).

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, given a distance $d(A,B)$ whose value is maximum when no characters match between A and B (which is the case of $ed(A,B)$ and $id(A,B)$), to compute the *transposition invariant version* of $d(A,B)$:

$$d^{\mathrm{t}}(A,B) \quad = \quad \min_{t \in \mathbb{Z}} d(A+t,B) \quad = \quad \min_{t \in [-\sigma,\sigma]} d(A+t,B),$$

where the latter equality is due to the fact that transpositions t outside the range $[-\sigma,\sigma]$ will not match any character of A to B, and thus $d(A+t,B)$ is maximum for those t.

For the cases of $ed(A,B)$ and $id(A,B)$, the computation of $d^{\mathrm{t}}(A,B)$ can be done naively in $O(mn\,\sigma)$ time [7] 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 for both distances [11]. Most recently [5], an algorithm that backtracks over the set of possible transpositions obtains $O((mn + \log\log\sigma)\log\sigma)$ in the best case and $O((mn + \log\sigma)\sigma)$ in the worst case.

In this paper we make use of the results on sparse dynamic programming [11] to obtain $O(mn \log\log\sigma)$ time algorithms for transposition invariant Levenshtein and indel distances. The idea is to split the dynamic programming matrix D into submatrices and apply sparse dynamic programming at each submatrix. The result is better than all previous work if $\sigma < \min(m,n)$. Moreover, the sparse dynamic programming algorithms [11] require $O(mn)$ space, whereas our algorithms need only $O(\sigma^2 + \min(m,n))$.

3 Using Sparse Dynamic Programming

The results in [11] resort to sparse dynamic programming to solve the transposition invariant distance computation problem. For each transposition $t \in [-\sigma,\sigma]$, the dynamic programming matrix D_t is computed to match $A + t$ against B. The key idea is that, for each transposition t, there are only a few matches between $A + t$ and B, that is, $a_i + t = b_j$. Added over all possible t values, there are exactly mn matches. Sparse dynamic programming algorithms compute $D_t(m,n)$ by considering only the *matching cells* (i,j) such that $a_i + t = b_j$. If there are r matches already spotted and in order, the algorithms developed in [11] require $O(r \log\log \min(m,n))$ time. Summed over all transpositions t, the cost becomes $O(mn \log\log \min(m,n))$.

There is a prior preprocessing work that classifies each cell (i,j) according to the transposition it belongs to. Hence we initialize $2\sigma + 1$ empty lists for $t \in [-\sigma,\sigma]$ and traverse the cells in the desired order, appending each cell (i,j) to the list for $t = b_j - a_i$. The extra cost of this preprocessing is $O(\sigma + mn)$.

Sparse dynamic programming is based on the following lemmas regarding the computation of $id(A,B)$ or $ed(A,B)$, whose basic version is proved in [11]. We prove slightly stronger versions here, by following the original proofs. In addition to the concept of matching cells $M = \{(i,j),\ a_i = b_j\}$, we define the *input cells* of matrix D as $In = \{(i,0),\ 0 \le i \le m\} \cup \{(0,j),\ 0 \le j \le n\}$ and the *output cells* as $Out = \{(i,n),\ 0 \le i \le m\} \cup \{(m,j),\ 0 \le j \le n\}$.

Lemma 1 *Let $D(0\ldots m, 0\ldots n)$ be the matrix that computes distance $id(A,B)$. Let M, In, and Out be the set of matching, input, and output cells of D, respectively. Then, any matching cell $(i,j) \in M$ can be computed using the*

recurrence

$$D(i,j) = \min\{D(i',j') + i - i' + j - j' - 2,\ (i',j') \in M \cup In, i' < i, j' < j\}, \quad (3)$$

and any output cell $(i,j) \in Out$ *can be computed using the above formula for* $D(i+1, j+1)$.

PROOF. Let us regard the computation of matrix D as a shortest path computation on a graph. Every path from an input cell to a matching cell (i,j), that is, to the target of a zero-cost edge, can be divided into two parts: (a) from the input cell until a cell (i',j') that is the target of the last zero-cost edge traversed before reaching (i,j), and (b) from cell (i',j') until cell (i,j). The path from (i',j') to (i,j) moves first to $(i-1, j-1)$ traversing only horizontal and vertical cost-1 edges, and then moves for free from $(i-1, j-1)$ to (i,j) (Eq. (1)). Overall, $(i-1) - i'$ vertical and $(j-1) - j'$ horizontal edges are traversed, for a total cost of $i - i' + j - j' - 2$. Hence the cost of this particular path is $D(i',j') + i - i' + j - j' - 2$. M contains all the cells that are targets of zero-cost edges, and therefore minimizing over all cells $(i',j') \in M$ yields the optimal cost, except for the possibility that the optimal path does not use any zero-cost edge before (i,j). This last possibility is covered by letting any input cell $(i',j') \in In$ that can influence $(i-1, j-1)$ participate in the minimization. The output cells $(i,j) \in Out$ can be obtained by pretending that $(i+1, j+1)$ is the target of a zero-cost edge, as the above computation effectively determines $D(i,j) = D(i+1, j+1)$. □

Lemma 2 *Let* $D(0 \ldots m, 0 \ldots n)$ *be the matrix that computes distance* $ed(A,B)$. *Let* M, In, *and* Out *be the set of matching, input, and output cells of* D, *respectively. Then, any matching cell* $(i,j) \in M$ *can be computed using the recurrence*

$$D(i,j) = \min \begin{cases} \{D(i',j') + j - j' - 1,\ (i',j') \in M \cup In, i' < i, j' - i' < j - i\} \\ \{D(i',j') + i - i' - 1,\ (i',j') \in M \cup In, j' < j, j' - i' \geq j - i\} \end{cases}, \quad (4)$$

and any output cell $(i,j) \in Out$ *can be computed using the above formula for* $D(i+1, j+1)$.

PROOF. Following the proof of Lemma 1 it is enough to show that the minimum path cost to reach cell $(i-1, j-1)$ from match point (i',j') is $j - j' - 1$ when $j' - i' < j - i$, and $i - i' - 1$ otherwise. The reason is that, in both cases, we use as many diagonal edges as possible and the rest are horizontal or vertical edges, depending on the case. □

The sparse dynamic programming algorithms in [11] operate via a data structure where cell values can be inserted and other cell values can be queried, considering only the values of already inserted cells and assuming that all the unknown cells are not matching cells. The values in $M \cup In$ are traversed in *reverse column-by-column order*, where cell (i', j') precedes (i, j) if $j' < j$, or if $j' = j$ and $i' > i$. This guarantees correctness and simplifies the operation of the algorithms (condition $j' < j$ in Eqs. (3) and (4) is automatically satisfied when $i' < i$). For each matching cell (i, j), a query to the data structure is performed in order to get the minimum over the relevant (i', j') cells, and then the resulting value, computed according to Lemma 1 or 2, is inserted as the value for cell (i, j).

These data structures permit inserting and querying an arbitrary number of cells, albeit for correct results we require that insertions and queries are performed in reverse column-by-column order. Let r be the number of cells inserted or queried, then the data structures used perform the r operations in $O(r \log\log \min(m, n))$ time.

In this paper we will use these algorithms over submatrices of D. These submatrices will have their input cells In initialized at arbitrary values, and we will want to obtain the values of all their output cells Out. For this sake, all the cells in $In \cup M$ will be inserted and all the cells in $M \cup Out$ will be computed with the proper query. Cells in In will be inserted with their initial values (at the proper time according to the reverse column-by-column order); those (i, j) in M will be inserted after querying for the minimum over relevant (i', j') values; and the output cells will be obtained (at the proper time) via the proper query, but instead of inserting them we will use their values to compute the bottom and rightmost borders of the matrix.

Figure 1 gives the pseudocode. The data structure is $\mathcal{S}$. After initialization, it permits adding cell values $D(i, j) = d$ using $\mathcal{S}.\mathbf{Add}(i, j, d)$, and computing cell values according to Eq. (3) or (4) using $\mathcal{S}.\mathbf{Compute}(i, j)$. This processing takes time $O(r \log\log \min(m, n))$, where $r = |M \cup In \cup Out|$.

4 The Algorithm

We compute a dynamic programming matrix $D_t(0 \ldots m, 0 \ldots n)$ for each transposition $t \in [-\sigma, \sigma]$, which corresponds to $d(A + t, B)$. We divide the dynamic programming matrices $D_t(0 \ldots m, 0 \ldots n)$ into $O(mn/k^2)$ *blocks* of $k \times k$ cells. Blocks will be labeled (r, s), for $0 \leq r < \lceil m/k \rceil$ and $0 \leq s < \lceil n/k \rceil$, corresponding to $D_t(kr + 1 \ldots kr + k, ks + 1 \ldots ks + k)$. The bottom and rightmost blocks may not be full but we ignore that for simplicity.

FillMatrix (D, In, M, Out)

1. $\mathcal{S}$.**Initialize**( )
2. **for** $(i,j) \in M \cup In \cup Out$ in reverse column-by-column order **do**
3. **if** $(i,j) \in In$ **then** $\mathcal{S}$.**Add**$(i, j, D(i,j))$
4. **else if** $(i,j) \in M$ **then** $\mathcal{S}$.**Add**$(i, j, \mathcal{S}$.**Compute**$(i,j))$
5. **else** // $(i,j) \in Out$ and it actually refers to $(i-1, j-1)$
6. $D(i-1, j-1) \leftarrow \mathcal{S}$.**Compute**$(i,j)$

Fig. 1. Algorithm to fill matrix D with input, matching, and output cells In, M, and Out, respectively, already sorted in reverse column-by-column order. Cells $(i,j) \in In$ are already computed in $D(i,j)$. Cells $(i,j) \in Out$ are shifted by one, as they are used to compute $D(i-1, j-1)$.

We compute all the D_t matrices simultaneously, row by row of blocks, each row from left to right. When we compute each block, we assume that its input cells (top row and leftmost column) are already computed, and after the computation we write its output cells (bottom row and rightmost column). Note that the input cells do not belong to the block itself.

Let us focus on the computation of a single block (r,s). The initial values of the input cells (top row and column) $D_t(kr, ks+j)$ and $D_t(kr+i, ks)$, for $0 \le i,j \le k$, are already known: Either $D_t(0, ks+j) = ks+j$ and $D_t(kr+i, 0) = kr+i$ (because of Eq. (1) or (2)), or they have already been computed as output cells of previously processed blocks ($D_t(kr, ks+j) = D_t(k(r-1)+k, ks+j)$, in block $(r-1, s)$, and $D_t(kr+i, ks) = D_t(kr+i, k(s-1)+k)$, in block $(r, s-1)$).

Therefore, each block (r,s) is processed by sparse dynamic programming according to the algorithms of the previous section (Figure 1), in $O(r \log\log k)$ per transposition. Since there are $2k-1$ input cells and $2k-1$ output cells per transposition, and the matching cells add up k^2 over all transpositions, we have $O((\sigma k + k^2)\log\log k)$ time per block. To this we must add the (negligible) $O(\sigma + k^2)$ time to collect transpositions and sort the cells. Figure 2 gives the pseudocode to compute a single block and Figure 3 the whole scheme.

By adding the above complexity over all the $O(mn/k^2)$ blocks, we get $O(mn(\sigma/k + 1)\log\log k)$, which is optimized in complexity for $k = \sigma$, to obtain $O(mn \log\log \sigma)$. That is, we split the matrix into $\sigma \times \sigma$ blocks. Note that we are assuming $\sigma \le \min(m,n)$, as otherwise the optimal block size cannot be attained. We return later to this case.

The space requirements have also decreased significantly. Whereas the original algorithms [11] require $O(mn)$ space, we use $O(k^2)$ space to process each block. If we process the matrix row by row, then we need only to remember the values of the bottom row cells of the previous row of blocks in order to process the

ComputeBlock (A, B, D, r, s, k)

1. **for** $t \in [-\sigma, \sigma]$ **do** M_t.**Initialize**( )
2. **for** $j \in 1 \ldots k$ **do**
3. **for** $i \in k \ldots 1$ **do**
4. $M_{b_{ks+j}-a_{kr+i}}$.**Add**(i, j)
5. In.**Initialize**( )
6. **for** $i \in k \ldots 0$ **do** In.**Add**$(i, 0)$
7. **for** $j \in 1 \ldots k$ **do** In.**Add**$(0, j)$
8. Out.**Initialize**( )
9. **for** $j \in 1 \ldots k-1$ **do** Out.**Add**$(k+1, j+1)$
10. **for** $i \in k \ldots 1$ **do** Out.**Add**$(i+1, k+1)$
11. **for** $t \in [-\sigma, \sigma]$ **do FillMatrix**$(D_t(kr \ldots kr+k, ks \ldots ks+k), In, M_t, Out)$

Fig. 2. Algorithm to compute block (r, s) for all transpositions t. Lines 1–10 deal with the initializations of sets In, M_t, and Out, and line 11 does the real sparse dynamic programming over the block (extended to include its input cells).

ComputeAll (A, B, m, n, k)

1. **for** $t \in [-\sigma, \sigma]$ **do**
2. **for** $i \in 0 \ldots m$ **do** $D_t(i, 0) \leftarrow i$
3. **for** $j \in 1 \ldots n$ **do** $D_t(0, j) \leftarrow j$
4. **for** $r \in 0 \ldots \lceil m/k \rceil - 1$ **do**
5. **for** $s \in 0 \ldots \lceil n/k \rceil - 1$ **do**
6. **ComputeBlock**(A, B, D, r, s, k)
7. **Return** $\min_{t \in [-\sigma, \sigma]} D_t(m, n)$

Fig. 3. Algorithm to compute $d^{\mathrm{t}}(A, B)$. It assumes that k divides m and n, otherwise some obvious but cumbersome twists are required.

current row of blocks (as well as the rightmost cells of the preceding block in the current row). This amounts to $O(n)$ space. Now, we can switch A and B if $m < n$, so as to ensure $O(\min(m, n))$ space. Overall, the space requirement is $O(k^2 + \min(m, n))$, which is $O(\sigma^2 + \min(m, n))$ if we choose $k = \sigma$.

It is possible to distinguish among transpositions that appear in a block from those that do not. For the latter, the output cells can be filled in $O(k)$ time both for indel and Levenshtein distance, using the algorithms for "different letter boxes" of [10]. This, however, does not improve the complexity of our algorithm.

5 Conclusions

We have presented an $O(mn \log\log \sigma)$ time algorithm to compute indel and Levenshtein distance with transposition invariance. The algorithm is simple and builds over a previous $O(\sigma + mn \log\log \min(m,n))$ solution, whose large $O(mn)$ space complexity is also lowered here to $O(\sigma^2 + \min(m,n))$. Our algorithm applies to the case $\sigma < \min(m,n)$, where it is better than any existing solution.

For the case $\min(m,n) \leq \sigma < \max(m,n)$, we can adapt our algorithm so that, even when it does not improve the time complexity of [11], it can greatly reduce its space. Assume w.l.o.g. $m \leq n$. We can partition the matrix into a horizontal strip of $m \times k$ blocks. Only m input/output cells need to be read/written by each block. The optimal choice is again $k = \sigma$, which yields $O(mn \log\log \min(m,n))$ time and $O(\sigma \min(m,n))$ space.

Just like in [11], the algorithms are easily extended to the search problem, where one seeks for all the substrings of B that are similar enough to A. Only a small change in the initial matrix conditions is necessary, $D_t(0,j) = 0$ for $1 \leq j \leq n$, to ensure that $D_t(m,j) = \min_{j'<j} d(A + t, b_{j'} \ldots b_j)$. Since our algorithms compute the whole bottom rows of the matrices, it is easy to detect all the positions j where the (transposition invariant) distance between A and a substring of B ending at j is below some desired threshold.

Non-contiguous alphabets $\Sigma \subset \mathbb{Z}$ can be handled by assuming that the alphabet is $\Sigma' = [\min(\Sigma), \max(\Sigma)]$, or even better, $\Sigma' = [\min\{a_i, b_j\}, \max\{a_i, b_j\}]$. Very sparse cases can be better addressed by considering only transpositions $t \in \{b_j - a_i\}$. This set can be collected in $O(\sigma' + mn)$ or in $O(mn \log(mn))$ time, and its size can be as bad as $\sigma'' = O(mn)$. This technique permits managing general alphabets, not only integer ones, but the usefulness of our approach depends on the size σ''.

Alternatively, for sparse integer alphabets, we can insert all the existing transpositions in a van Emde Boas tree [14,15] and then collect them all with the *successor* operation, in overall time $O(mn \log\log \sigma')$. This requires additional space $O(\sigma')$ (which is allocated but not necessarily written), which can be reduced to $O(\sigma'^{\varepsilon})$ for any constant $\varepsilon > 0$ [4]. By using randomization the space can be brought back to $O(mn)$ [12]. The idea is also relevant for the case of a contiguous alphabet $[0, \sigma]$, where the original algorithm [11] can be made $O(mn(\log\log \sigma + \log\log \min(m,n)))$ instead of $O(\sigma + mn \log\log \min(m,n))$. Yet another solution, if $O(mn)$ space is available, is to radix-sort all the mn transpositions by successive stages of $O(\log \max(m,n))$ bits each, to obtain $O(mn \log(\sigma')/\log \max(m,n))$ time.

Another kind of matching relaxation of interest in music retrieval is the so-

called δ-matching, where characters a_i and b_j match for all transpositions $b_j - a_i - \delta \leq t \leq b_j - a_i + \delta$. In [11] this is handled in $O(\delta mn \log\log \min(m,n))$ time. By choosing $k = \sigma/\delta$, we achieve $O(\delta mn \log\log(\sigma/\delta))$ time.

It is not clear whether it is possible to achieve $O(mn)$ time on transposition invariant distance computation, so that no penalty in complexity is paid for the transposition invariance. This has been achieved for a distance that permits only deletions [11] (episode matching), but it remains open for indel and Levenshtein distances. Other open problems are to obtain $O(mn$ polylog $\sigma)$ complexity for the case of α-limited distances, where the distance between two consecutive matching characters in A and B cannot exceed α. Complexities of the form $O(mn$ polylog $m)$ are obtained in [11], but the matrix partitioning technique cannot be immediately applied to that case.

References

[1] B. Bollobás, G. Das, D. Gunopulos, and H. Mannila. Time-series similarity problems and well-separated geometric sets. *Nordic Journal of Computing*, 8(4):409–423, 2001.

[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] K. Fredriksson, V. Mäkinen, and G. Navarro. *Rotation and Lighting Invariant Template Matching.* In *Proc. LATIN 2004*, pp. 39–48. LNCS v. 2976, 2004.

[4] D. Johnson. A priority queue in which initialization and queue operations take time $O(\log\log D)$. *Math. Systems Theory* 15, 295–209, 1982.

[5] 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.

[6] K. Lemström and J. Tarhio. Searching monophonic patterns within polyphonic sources. In *Proc. RIAO 2000*, pp. 1261–1279 (vol 2), 2000.

[7] K. Lemström and E. Ukkonen. Including interval encoding into edit distance based music comparison and retrieval. In *Proc. AISB 2000*, pp. 53–60, 2000.

[8] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady* 6:707–710, 1966.

[9] H. Mannila and H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *Proc. 1st International Conference on Knowledge Discovery and Data Mining (KDD'95)*, AAAI Press, pp. 210–215, 1995.

[10] V. Mäkinen, G. Navarro, and E. Ukkonen. Approximate matching of run-length compressed strings. *Algorithmica* 35:347–369, 2003.

[11] 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, To appear in *Journal of Algorithms*. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/ti_matching.ps.gz`.

[12] K. Mehlhorn and S. Näher. Bounded ordered dictionaries in $O(\log \log N)$ time and $O(n)$ space. *Information Processing Letters* 35, 183–189, 1990.

[13] P. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms*, 1(4):359–373, 1980.

[14] P. van Emde Boas, R. Kaas, E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Systems Theory*, 10:99–127, 1977.

[15] 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.

[16] R. Wagner and M. Fisher. The string-to-string correction problem. *J. of the ACM* 21(1):168–178, 1974.