Transposition Invariant String Matching ⋆

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

[a] *Department of Computer Science, P.O Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, Finland.*

[b] *Center for Web Research, Department of Computer Science, University of Chile Blanco Encalada 2120, Santiago, Chile.*

Abstract

Given strings $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$ 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, including *Hamming distance*, *longest common subsequence (LCS)*, *Levenshtein distance*, and their versions where the exact matching condition is replaced by an approximate one. For all these problems we give algorithms whose time complexities are close to the known upper bounds without transposition invariance, and for some we achieve these upper bounds. In particular, we show how sparse dynamic programming can be used to solve transposition invariant problems, and its connection with multidimensional range-minimum search. As a byproduct, we give improved sparse dynamic programming algorithms to compute LCS and Levenshtein distance.

Key words: edit distance, music sequence comparison, transposition invariance, sparse dynamic programming, range-minimum searching.

⋆ Preliminary version appeared as [34].

Email addresses: vmakinen@cs.helsinki.fi (Veli Mäkinen), gnavarro@dcc.uchile.cl (Gonzalo Navarro), ukkonen@cs.helsinki.fi (Esko Ukkonen).

[1] Supported by the Academy of Finland under grant 22584.

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

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 string.

Interest in transposition invariant string matching problems has recently arisen in the field of music information retrieval (MIR) [11,28,29]. In music analysis and retrieval, one often wants to compare two music pieces to test how similar they are. One way to do this is to define a distance measure between the corresponding note sequences. Transposition invariance is one of the properties that such a distance measure should fulfill to reflect a human sense of similarity. There are other application areas where transposition invariance is useful, like time series comparison [5], image comparison [21], and others (see Section 3).

In this paper we study how transposition invariance can be achieved when evaluating some of the classical distance measures for strings. We focus on measures that have been used in practice and have applications in MIR. We are interested in the intrinsic difficulty of the problem, focusing on the essential aspects and in worst case complexities. Our aim is to build a foundation on top of which one can develop practical improvements such as good average cases, threshold-sensitive computation, bit-parallel simulation, four-russians techniques, filtering approaches, and so on.

We show that several transposition invariant string matching problems can be reduced to sparse dynamic programming, and demonstrate the connection between the latter and multidimensional range-minimum searching. In some cases our new sparse dynamic programming techniques are inferior compared to the best existing solutions, but in other cases we give improved solutions to well known problems such as sparse computation of longest common subsequences and Levenshtein distance. Moreover, our techniques are flexible and can be successfully extended to cases of interest that cannot be handled by the best current algorithms, for example to distances where matching characters cannot be too far apart. As a result, we show that all the distance measures studied allow including transposition invariance without a significant increase in the asymptotic running times (in most cases we pay polylogarithmic penalty factors).

The paper is organized as follows. In Section 2 we give the main definitions we use, including the string similarity measures we focus on. In Section 3 we cover related work and give at the same time motivations for some of the string matching problems addressed. In Section 4 we summarize our main

results. Section 5 is devoted to the so-called "edit distances" (where characters in both strings can be ignored) and Section 6 to the simpler distances where all characters must be aligned one by one. Finally, Section 7 gives our conclusions and future work directions.

2 Definitions

Let Σ be a numerical alphabet, which is a subset of some totally ordered universe $\mathbb{U}$ that is closed under addition and subtraction. Let $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$ be two *strings* in Σ^*, that is, $a_i, b_j \in \Sigma$ 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 study are symmetric [3]. String A' is a *substring* of A if $A' = A_{i...j} = a_i \ldots a_j$ 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 between two strings A and B. These measures can be found in any standard text book of string algorithms, see for example [17,25]. 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 dual problem of computing LCS is to compute distance d_{ID}, which is the minimum number of character insertions and deletions necessary to convert A into B (or vice versa). The duality is clear since $d_{\mathrm{ID}}(A, B) = m + n - 2 \cdot \mathrm{lcs}(A, B)$. For convenience, we will mainly use the minimization problem d_{ID} (not lcs) in the sequel. If we permit character substitutions in addition to insertions and deletions, the result is the unit cost *Levenshtein distance* d_{L} [30]. This is a particular case of more complex distances that assign a different cost to each operation and minimize the total cost of operations [39,35]. Finally, if only deletions of characters of B are allowed, we get distance d_{D}. We call d_{ID}, d_{L} and d_{D} collectively "edit distances".

When $m = n$, the following distances can also be defined. The *Hamming distance* d_{H} between strings A and B is $d_{\mathrm{H}}(A, B) = |\{i \mid a_i \neq b_i, 1 \leq i \leq m\}|$. The *sum of absolute differences distance* d_{SAD} between A and B is $d_{\mathrm{SAD}}(A, B) = \sum_{i=1}^{m} |a_i - b_i|$. The *maximum absolute difference distance* d_{MAD} between A and B is $d_{\mathrm{MAD}}(A, B) = \max\{|a_i - b_i| \mid 1 \leq i \leq m\}$. Note that d_{SAD} is in fact the Manhattan metric (l_1 norm) and d_{MAD} is the maximum metric (l_∞ norm) when we interpret A and B as points in m-dimensional Euclidean space.

String A is a *transposed copy* of B (denoted by $A =^t B$) if $B = A + t = (a_1 + t)(a_2 + t) \cdots (a_m + t)$ for some $t \in \mathbb{U}$. The transposition invariant versions of the above distance measures d_* where $* \in \{\mathrm{ID}, \mathrm{L}, \mathrm{D}, \mathrm{H}, \mathrm{SAD}, \mathrm{MAD}\}$ can

[3] Except for d_{D}, but in this case it is necessary that $m \leq n$.

now be stated as $d^{\mathrm{t}}_*(A,B) = \min_{t\in\mathbb{U}} d_*(A+t,B)$.

So far our definitions allow either only exact (transposition invariant) matches between some characters ($d^{\mathrm{t}}_{\mathrm{ID}}, d^{\mathrm{t}}_{\mathrm{L}}, d^{\mathrm{t}}_{\mathrm{D}}, d^{\mathrm{t}}_{\mathrm{H}}$), or approximate matches between all characters ($d^{\mathrm{t}}_{\mathrm{SAD}}, d^{\mathrm{t}}_{\mathrm{MAD}}$). To relax these conditions, we introduce a constant $\delta > 0$. We write $a =^{\delta} b$ when $|a-b| \leq \delta$, $a, b \in \Sigma$. By replacing the equalities $a = b$ with $a =^{\delta} b$, we get more error-tolerant versions of the distance measures: $d^{\mathrm{t},\delta}_{\mathrm{ID}}, d^{\mathrm{t},\delta}_{\mathrm{L}}$, $d^{\mathrm{t},\delta}_{\mathrm{D}}$, and $d^{\mathrm{t},\delta}_{\mathrm{H}}$. Similarly, by introducing another constant $\kappa > 0$, we can define distances $d^{\mathrm{t},\kappa}_{\mathrm{SAD}}, d^{\mathrm{t},\kappa}_{\mathrm{MAD}}$ such that the κ largest differences $|a_i - b_i|$ are discarded.

We can also define α–limited versions of the edit distance measures, where the distance (gap) between any two consecutive matching characters in A or B is limited by a constant $\alpha > 0$. That is, if in order to obtain $d(A,B)$ characters $a_{i_1}, a_{i_2}, \ldots, a_{i_r}$ match $b_{j_1}, b_{j_2}, \ldots, b_{j_r}$, while the others are inserted, deleted or substituted (depending on the distance), then $i_\ell - i_{\ell-1} - 1 \leq \alpha$ and $j_\ell - j_{\ell-1} - 1 \leq \alpha$ for all $1 < \ell \leq r$. We get distances $d^{\mathrm{t},\delta,\alpha}_{\mathrm{ID}}, d^{\mathrm{t},\delta,\alpha}_{\mathrm{L}}$, and $d^{\mathrm{t},\delta,\alpha}_{\mathrm{D}}$.

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 $P_{1\ldots m} = p_1 p_2 \cdots p_m$, and call B the *text* and denote it $T_{1\ldots n} = t_1 t_2 \cdots t_n$, and usually assume that $m << n$. A closely related problem is the *thresholded search problem* where, given P, T, and a threshold value $k \geq 0$, one wants to find all the text positions j_r such that $d(P, T_{j_l \ldots j_r}) \leq k$ for some j_l. We will refer collectively to these two closely related problems as the *search problem*.

In particular, if distance d_{D} is used in approximate string matching, we obtain a problem known as *episode matching* [31,18], which can also be stated as follows: Find the shortest substring of the text that contains the pattern as a subsequence. Another search problem related to d_{SAD} and d_{MAD} is called "(δ,γ)–matching" [7], where one wants to find all occurrences j_r such that $d_{\mathrm{MAD}}(P, T_{j_r-m+1\ldots j_r}) \leq \delta$ and $d_{\mathrm{SAD}}(P, T_{j_r-m+1\ldots j_r}) \leq \gamma$.

Our complexity results will vary depending on the form of the alphabet Σ. We will distinguish two cases. An *integer* alphabet is any finite alphabet $\Sigma \subset \mathbb{Z}$. For integer alphabets, $|\Sigma|$ will denote $\max(\Sigma) - \min(\Sigma) + 1$. A *general* alphabet will be any other Σ, finite or not, and we will omit any reference to $|\Sigma|$. We will only assume that Σ is totally ordered and closed under addition and subtraction (a good example to fix ideas is $\Sigma = \mathbb{R}$). On the other hand, 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. In these cases we will use $|\Sigma_A| = \max(\Sigma_A) - \min(\Sigma_A) + 1 \leq |\Sigma|$ when Σ_A is taken as an integer alphabet. On general alphabets, $|\Sigma_A| \leq m$ will denote the cardinality of the set Σ_A.

3 Related Work and Motivation

We start by noticing that the problem of *exact* transposition invariant string matching is extremely easy to solve. For the comparison problem, the only possible transposition is $t = b_1 - a_1$. For the search problem, 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 methods developed for exact string matching. Unfortunately, this relative encoding seems to be of no use when the exact string 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 [29]. They proposed, among other measures, transposition invariant longest common subsequence (LCTS) as a measure of similarity between two music (pitch) sequences. They gave a descriptive nickname 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 [29]: Compute $\mathrm{lcs}(A + t, B)$ using $O(mn)$ dynamic programming for each $t \in \mathbb{T}$. The running time of this algorithm is $O(|\Sigma|mn)$, where typically $|\Sigma| = 128$. In the general case, where Σ can be unlimited, one could instead use the set of transpositions $\mathbb{T}' = \{b - a \mid a \in \Sigma_A, b \in \Sigma_B\}$. This is because some characters must match in any meaningful transposition. The size of $\mathbb{T}'$ could be mn, which gives $O(m^2n^2)$ worst case time for general alphabets. Thus it is of both practical and theoretical interest to improve this algorithm.

The Levenshtein distance allows substituting a note by some other note. A natural extension would be to make the cost of a substitution operation depend on the distance between the notes. This is however problematic since there is no natural way of defining costs of insertions and deletions in this setting. We have chosen an alternative approach: A tolerance $\delta > 0$ is allowed for matching pitch levels. This can be used to allow matches between pitch levels that are relatively close. In practice, one could use different values δ for each pitch level to better reflect musical closeness.

While the LCS and the edit distance in general are useful tools for comparing two sequences that represent whole musical pieces, simpler measures could be used in the search problem. An especially suitable relaxation of the LCS is episode matching [31,18]. Assume that the pattern is (a discretized version of a signal) given by humming. The goal is to search for the matching musical pieces in a large music database. The pattern obtained by humming would usually contain the melody in its simplest form, but the searched occurrences in the music database might additionally contain some "decorative" notes, which

were forgotten by the person humming the piece. Episode matching would find the occurrences that contain the fewest decorative notes. This is a good objective, since an occurrence with a large number of additional notes would not be recognized as the same piece of music. A version of episode matching has been proposed in the context of MIR [**?**,13], where the number of these additional notes between two matching pitches is limited by a constant. This variant, as well as the original problem, can be solved using dynamic programming in $O(mn)$ time. Including transposition invariance has not been considered. We will study this problem and "matching with α–limited gaps" in general, where an additional restriction to the d_{ID}, d_{L} and d_{D} distances is that the gap between two consecutive matching characters is limited by an integer $\alpha > 0$. This aims at avoiding seriously distorted occurrences where, although the total number of extra notes is a small fraction of the whole string, they are all concentrated in the same place, so that a human would not recognize both strings as variants of the same melody. Moreover, such restrictions become necessary in other types of edit distances, see for example the edit distances for point-patterns developed in [33]. Here we will only concentrate on the α-limitation on well-known distance measures, since this is enough to demonstrate the key techniques.

Even simpler measures have been proposed for the search problem. These include variants of d_{H}^{δ}, d_{SAD} and d_{MAD}, such as the (δ, γ)–matching problem [7,12,15,16], where occurrences should have limited d_{MAD} and d_{SAD} distances to the pattern, simultaneously. Algorithms for exact string matching can be generalized to this special case, and bit-parallel algorithms can be applied [7,16]. These algorithms are fast in the average case and in practice, but their worst case is still $O(mn)$. In fact, for $\delta = \infty$ the problem is known as the weighted k-mismatches problem [32], for which it has long been an open question whether the quadratic bound can be improved. We will not answer that here, but we will show that within the same bounds one can solve the harder problem where transposition invariance is included.

So far we have discussed problems for monophonic musical sequences. Polyphonic music is much more challenging. Usually one would be interested in finding occurrences of a monophonic pattern in a polyphonic music. The basic approach would be to separate polyphonic music into parallel monophonic pitch sequences (each instrument separately). This case can be handled easily by applying algorithms for monophonic music. This would however lose the melodies that "jump" between instruments. To find these melodies one should represent the polyphonic music as a sequence of subsets of pitch levels. The exact matching is in this case called subset matching, for which novel (but impractical) algorithms have been developed [8–10]. To allow transposition invariance, one could simulate these algorithms with each possible transposition. The time complexity would then be $O(|\Sigma| s \log^2 s)$ [10], where s is the sum of the subset sizes. A practical approach has been taken by Lemström

and Tarhio [28], who developed a fast filter for the problem with transposition invariance, as well as a simple verification algorithm that has running time $O(|\Sigma|n + sm)$. We note that the edit distance problems can easily be adapted to the case in which the text consists of subsets. A more robust extension of episode matching for polyphonic music, where the number of jumps is controlled, was also studied [27].

Other applications for transposition invariance can be found, for example, in image processing and time series comparison. In image comparison, one could for example use the sum of absolute differences to find approximate occurrences of a template pattern inside a larger image. This measure is used, for instance, by Fredriksson in his study of rotation invariant template matching [21]. Transposition invariance would mean "lighting invariance" in this context. As images usually contain a lot of noise, the measure where κ largest differences can be discarded could be useful. We study the combination of rotation and lighting invariances in a subsequent paper [22].

In time series comparison, many of the measures mentioned can be used. In fact, episode matching was first introduced in this context [31]. Recently, a problem closely related to transposition invariant LCS was studied by Bollobás et al. [5]. They studied a more difficult problem where not only transposition (translation), but also scaling was allowed. They also allowed a tolerance between matched values, but did not consider transpositions alone. Our algorithms could be useful to improve these results, as dynamic programming algorithms are used as a black box in their techniques, and we give improved (sparse) dynamic programming algorithms.

4 Summary of Results

Our results are two-fold. For evaluating the easier distance measures ($d_{\mathrm{H}}^{\mathrm{t},\delta}, d_{\mathrm{SAD}}^{\mathrm{t},\kappa}, d_{\mathrm{MAD}}^{\mathrm{t},\kappa}$) we achieve almost the same bounds that are known without transposition invariance. These results are achieved by noticing that the optimum transposition can be found without evaluating the distances for each possible transposition.

For the more difficult measures ($d_{\mathrm{ID}}^{\mathrm{t},\delta,\alpha}$, $d_{\mathrm{L}}^{\mathrm{t},\delta,\alpha}$, and $d_{\mathrm{D}}^{\mathrm{t},\delta,\alpha}$) we still need to compute the distances for each possible transposition. This would be costly if the standard dynamic programming algorithms for these problems were used. However, we show that *sparse* dynamic programming algorithms can be used to obtain much better worst case bounds. Then we show the connection between the resulting sparse dynamic programming problems and multidimensional range-minimum queries. We obtain simple yet efficient algorithms for these distances.

For LCS (and thus for d_{ID}) there already exists Hunt-Szymanski [26] 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 $O(r \log n)$ once the matching pairs are given in correct order. As the sum of values r over all different transpositions is mn, we get the bound $O(mn \log n)$ for the transposition invariant case. Later improvements [2,20] permit reducing this complexity to $O(mn \log \log n)$ time (see Section 5.2). We improve this to $O(mn \log \log m)$ by giving a new $O(r \log \log \min(m, r))$ sparse dynamic programming algorithm for LCS. This algorithm can also be generalized to the case where gaps are limited by a constant α, giving $O(mn \log m)$ time for evaluating $d_{\mathrm{ID}}^{\mathrm{t},\alpha}(A, B)$.

Eppstein et al. [20] have proposed sparse dynamic programming algorithms for more complex distance computations such as the Wilbur-Lipman fragment alignment problem [40,41]. The unit cost Levenshtein distance can also be solved using these techniques [24]. Using this algorithm, the transposition invariant case can be solved in $O(mn \log \log n)$ time. However, the algorithm does not generalize to the case of α-limited gaps, and thus we develop an alternative solution that is based on two-dimensional semi-static range minimum queries. This gives us $O(mn \log^2 m)$ time for evaluating $d_{\mathrm{L}}^{\mathrm{t},\alpha}(A, B)$. However, we develop in passing an improved $O(r \log \log m)$ sparse dynamic programming algorithm for Levenshtein distance, which permits computing $d_{\mathrm{L}}^{\mathrm{t}}$ in $O(mn \log \log m)$ time. Also, we note that our algorithm to compute $d_{\mathrm{L}}^{\mathrm{t},\alpha}(A, B)$ can be applied to the case without transpositions, where it is still $O(mn \log^2 m)$, and hence better than the existing $O(\alpha mn)$ time algorithm [33] for $\alpha = \Omega(\log^2 m)$.

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

The search problems on the edit distances can be solved in general within the same time bounds of the distance computation problems. For the simpler distances, on the other hand, our only solution is to evaluate them at every text position.

Table 1 gives a simplified list of upper bounds that are known for these problems without transposition invariance. Table 2 gives the achieved upper bounds for the transposition invariant versions of these problems.

We start by describing our solutions to the edit distances, since they are the main emphasis of this paper. Then we briefly give the other results for Hamming distance and related measures.

distance	distance evaluation	searching
exact	$O(m)$	$O(m+n)$
d_{H}	$O(m)$	$O(n\sqrt{m \log m})$ [1]
d_{H}^{δ}	$O(m)$	$O(mn)$
d_{SAD}	$O(m)$	$O(mn)$
d_{MAD}	$O(m)$	$O(mn)$
(δ,γ)–matching	$O(m)$	$O(mn)$
$d_{\mathrm{ID}}, d_{\mathrm{L}}$	$O(mn/\log m)$	$O(mn/\log m)$ [14]
$d_{\mathrm{ID}}^{\delta,\alpha}$	$O(mn)$	$O(mn)$ [33]
$d_{\mathrm{L}}^{\delta,\alpha}$	$O(\alpha mn)$	$O(\alpha mn)$ [33]
d_{D}	$O(mn/\log m)$	$O(mn/\log m)$ [18]
$d_{\mathrm{D}}^{\delta,\alpha}$	$O(mn)$	$O(mn)$ [13]

Table 1
Upper bounds for string matching without transposition invariance. We omit bounds that depend on the threshold k in the search problems.

distance	distance evaluation	searching
exact	$O(m)$	$O(m+n)$
$d_{\mathrm{H}}^{\mathrm{t},\delta}$	$O(m \log m)$	$O(mn \log m)$
$d_{\mathrm{SAD}}^{\mathrm{t},\kappa}$	$O(m+\kappa \log \kappa)$	$O((m+\kappa \log \kappa)n)$
$d_{\mathrm{MAD}}^{\mathrm{t},\kappa}$	$O(m+\kappa \log \kappa)$	$O((m+\kappa \log \kappa)n)$
(δ,γ)–matching	$O(m)$	$O(mn)$
$d_{\mathrm{ID}}^{\mathrm{t},\delta}$	$O(\delta mn \log\log m)$	$O(\delta mn \log\log m)$
$d_{\mathrm{ID}}^{\mathrm{t},\delta,\alpha}$	$O(\delta mn \log m)$	$O(\delta mn \log m)$
$d_{\mathrm{L}}^{\mathrm{t},\delta}$	$O(\delta mn \log\log m)$	$O(\delta mn \log\log m)$
$d_{\mathrm{L}}^{\mathrm{t},\delta,\alpha}$	$O(\delta mn \log^2 m)$	$O(\delta mn \log^2 m)$
$d_{\mathrm{D}}^{\mathrm{t},\delta,\alpha}$	$O(\delta mn)$	$O(\delta mn)$

Table 2
Our upper bounds for transposition invariant string matching. On an integer alphabet, term $m \log m$ in $d_{\mathrm{H}}^{\mathrm{t},\delta}$ can be replaced by $|\Sigma|+m$, and $\kappa \log \kappa$ by $|\Sigma|+\kappa$. We have not added, for clarity, the preprocessing time of Theorem 2 for the edit distance measures. Finally, δ should be understood as $(2\delta+1)/\mu$, where μ is the minimum difference between any two different $a_i - b_j$ values ($\mu = 1$ on integer alphabets).

5 Transposition Invariant Edit Distances

Let us first review how the edit distances can be computed using dynamic programming [30,39,35]. Let $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$. For d_{ID}, evaluate an $(m+1) \times (n+1)$ matrix (d_{ij}), $0 \le i \le m$, $0 \le j \le n$, using the recurrence

$$d_{i,j} = \min((\textbf{if } a_i = b_j \textbf{ then } d_{i-1,j-1} \textbf{ else } \infty), d_{i-1,j} + 1, d_{i,j-1} + 1), \quad (1)$$

with initialization $d_{i,0} = i$ for $0 \le i \le m$ and $d_{0,j} = j$ for $0 \le j \le n$.

The matrix (d_{ij}) can be evaluated (in some suitable order, like row-by-row or column-by-column) in $O(mn)$ time, and the value d_{mn} equals $d_{\mathrm{ID}}(A, B)$.

A similar method can be used to calculate distance $d_{\mathrm{L}}(A, B)$. Now, the recurrence is

$$d_{i,j} = \min((d_{i-1,j-1} + \textbf{if } a_i = b_j \textbf{ then } 0 \textbf{ else } 1), d_{i-1,j} + 1, d_{i,j-1} + 1), \quad (2)$$

with initialization $d_{i,0} = i$ for $0 \le i \le m$ and $d_{0,j} = j$ for $0 \le j \le n$.

The recurrence for distance $d_{\mathrm{D}}(A, B)$, which is used in episode matching, is

$$d_{i,j} = \textbf{if } a_i = b_j \textbf{ then } d_{i-1,j-1} \textbf{ else } d_{i,j-1} + 1, \quad (3)$$

with initialization $d_{i,0} = \infty$ for $0 \le i \le m$ and $d_{0,j} = j$ for $1 \le j \le n$.

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

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 $d_{\mathrm{ID}}(A, B)$, 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 d_{L} this graph has also diagonal edges of cost 1 from every cell $(i-1, j-1)$ to (i, j). For d_{D}, the graph contains only the horizontal edges and the zero-cost diagonal edges. For searching, we add zero-cost edges connecting $(0, j-1)$ to $(0, j)$ for every j.

To solve our transposition invariant problems, we compute the distances in *all* required transpositions, but we use algorithms that are more efficient than the above basic dynamic programming solutions, such that the overall complexity does not exceed by much the worst case complexities of computing the distances for a single transposition.

Let M be the set of matching characters (also called match set) between strings A and B, that is, $M = M(A,B) = \{(i,j) \mid a_i = b_j, 1 \leq i \leq m, 1 \leq j \leq n\}$. The match set corresponding to a transposition t will be called $M_t = M(A+t,B) = \{(i,j) \mid a_i+t = b_j\}$. Let $r = r(A,B) = |M(A,B)|$. Let us define $\mathbb{T}$ to be the set of those transpositions that make some characters match between A and B, that is $\mathbb{T} = \{b_j - a_i \mid 1 \leq i \leq m, 1 \leq j \leq n\}$. One could compute the above edit distances and solve the search problems by running the above recurrences over all pairs $(A+t,B)$, where $t \in \mathbb{T}$. In an integer alphabet this takes $O(|\Sigma|mn)$ time, and $O(|\Sigma_A||\Sigma_B|mn) = O(m^2n^2)$ time in a general alphabet. This kind of procedure can be significantly sped up if the basic dynamic programming algorithms are replaced by suitable "sparse dynamic programming" algorithms.

Moreover, we are actually interested in computing the edit distances allowing approximate matches between the characters (recall the versions with parameter δ). To take these approximate matches into account, let us redefine our match set M_t as $M_t^\delta = \{(i,j) \mid |b_j - (a_i + t)| \leq \delta\}$.

We note that, if $\delta = 0$, then the sum of the sizes of all the match sets is mn, that is, $\sum_t |M_t| = mn$. However, if $\delta > 0$ then each cell may participate in more than one relevant transposition, and the total size of the match sets, $\sum_t |M_t^\delta|$, may perfectly exceed mn. On an integer alphabet, each cell can participate at most in $2\delta+1$ match sets, so the overall size is $\sum_t |M_t^\delta| \leq (2\delta+1)mn$. On a general alphabet, this is not enough. Let us call μ the smallest difference between two different relevant transpositions, then it holds $\sum_t |M_t^\delta| \leq (2\delta+1)mn/\mu$. Note that $\mu = 1$ on an integer alphabet.

Lemma 1 *If distance $d(A,B)$ can be computed in $O(g(r(A,B))f(m,n))$ time, where $g()$ is a concave increasing 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. The δ-tolerant distance $d^{\mathrm{t},\delta}(A,B) = \min_{t\in\mathbb{T}} d^\delta(A+t,B)$ can be computed in $O(g(\sum_t |M_t^\delta|)f(m,n))$ time.*

PROOF. For $\delta = 0$, let $r_t = |M_t| = r(A+t,B)$ be the number of matching character pairs between $A+t$ and B. Then

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

The case $\delta > 0$ is similar (change the order of the summations in the second line above, and $\sum_{t\in\mathbb{T}} M_t^\delta$ shows up). $\square$

The rest of the section is devoted to developing algorithms that depend on r. However, we start by considering how to obtain the sets $M_t = M(A+t, B)$.

5.1 Preprocessing

As a first step, we need a way of constructing the match sets M_t sorted in some order that enables sparse evaluation of matrix (d_{ij}).

We must be careful in constructing these match sets for all transpositions so that the overall preprocessing time does not exceed the time needed for the actual distance computations. For example, one could easily construct a match set by considering all the mn pairs (i,j) in any desired order and adding each pair (i,j) to $M_{b_j-a_i}$, first initializing it if the transposition $t = b_j - a_i$ did not previously exist. This method gives us $O(|\Sigma| + mn)$ time on an integer alphabet and $O(mn\log(|\Sigma_A||\Sigma_B|)) = O(mn\log n)$ on a general alphabet (by using a balanced tree of existing transpositions).

Let us now consider the case $\delta > 0$. Now each pair (a_i, b_j) defines a *range* of relevant transpositions, $[b_j - a_i - \delta, b_j - a_i + \delta]$. However, only at the extremes of those ranges the sets M_t^δ can change, so it is enough to consider two transpositions, $b_j - a_i - \delta$ and $b_j - a_i + \delta$, for each pair (a_i, b_j). Moreover, if $t' = t + \epsilon$ such that a range finishes between t and t' and all the rest stays the same, then $M_{t'}^\delta \subseteq M_t^\delta$, and because of the definitions of edit distances, $d(A+t, B) \le d(A+t', B)$ for any edit distance. This shows that it is enough to consider only the places where ranges start (or, symmetrically, all the places where ranges finish, but not both). Hence, we will compute M_t^δ for $t \in \{b_j - a_i - \delta\}$.

Theorem 2 *The* match sets $M_t^\delta = \{(i,j) \mid |b_j - (a_i + t)| \le \delta\}$, *each sorted in a desired cell order, for all relevant transpositions* $t \in \mathbb{T} = \{b - a - \delta, a \in \Sigma_A, b \in \Sigma_B\}$, *can be constructed in time* $O(|\Sigma| + (2\delta+1)mn)$ *on an integer alphabet, and*

in time $O(m \log |\Sigma_A| + n \log |\Sigma_B| + |\Sigma_A||\Sigma_B| \log(\min(|\Sigma_A|, |\Sigma_B|)) + \sum_{t \in \mathbb{T}} |M_t^\delta|)$ *on a general alphabet.*

PROOF. On an integer alphabet we can proceed naively to obtain $O(|\Sigma| + mn)$ time using array lookup to get the transposition $b_j - a_i$ where each pair (i, j) has to be added. For $\delta > 0$ each pair (i, j) is added to entries from $b_j - a_i - \delta$ to $b_j - a_i + \delta$, in $O(|\Sigma| + (2\delta + 1)mn)$ time.

The case of general alphabets is solved as follows.

(*i*) Start by obtaining the sets of different characters in A and B. Create a balanced binary search tree $\mathcal{T}_A$ where every character $a = a_i$ of A is inserted, maintaining for each such $a \in \Sigma_A$ a list $\mathcal{L}_a$ of the positions i of A, in increasing order, such that $a = a_i$. Do the same for B and $\mathcal{T}_B$. This costs $O(m \log |\Sigma_A| + n \log |\Sigma_B|)$.

(*ii*) Then, obtain a sorted list of all the relevant transpositions, with duplicates. Let us assume $|\Sigma_A| \leq |\Sigma_B|$ (otherwise do it the symmetric way). For each $a \in \mathcal{T}_A$, traverse all b in $\mathcal{T}_B$ in order and generate a list of increasing transpositions and their corresponding position lists $(b-a-\delta, \mathcal{L}_a, \mathcal{L}_b)$. Then merge the $|\Sigma_A|$ lists into a unique ordered list of relevant transpositions and positions, where there are possible duplicates in the $b - a - \delta$ values (but these are all contiguous). Since we choose the smaller alphabet to traverse the larger, this part costs $O(|\Sigma_A||\Sigma_B| \log(\min(|\Sigma_A|, |\Sigma_B|)))$ time.

(*iii*) Now, create list $\mathbb{T}$ of relevant transpositions and associate the set of positions M_t^δ to each $t \in \mathbb{T}$. We will need to fill simultaneously a matrix C of m rows and n columns, such that each cell $C_{i,j}$ points to the proper node M_t^δ in $\mathbb{T}$. Traverse the sorted list of transpositions and remove duplicate transpositions, appending a new node $t = b - a - \delta$ at the end of $\mathbb{T}$, where M_t^δ is stored, initially empty. At the same time, each time a list entry $(b - a - \delta, \mathcal{L}_a, \mathcal{L}_b)$ is processed, assign a pointer to M_t^δ at each cell $C_{i,j}$ for every $i \in \mathcal{L}_a$ and $j \in \mathcal{L}_b$. This costs $O(mn)$ since every cell of C will be visited exactly once.

(*iv*) Finally, fill the M_t^δ sets. Traverse matrix C in any desired order, and for each processed entry (i, j), add (i, j) to the set pointed to by $C_{i,j}$ (that is, $M^\delta_{b_j - a_i - \delta}$). This costs $O(mn)$. If $\delta > 0$ add entry (i, j) not only to $C_{i,j}$, but also move forward in the sorted list $\mathbb{T}$, adding entry (i, j) to next transpositions $b' - a'$ while $(b' - a') - (b - a) \leq 2\delta$. This costs $\sum_{t \in \mathbb{T}} |M_t^\delta|$. □

In the rest of this section, we will only consider explicitly the case $\delta = 0$ and develop algorithms that compute a distance $d(A, B)$ using a match set M_t. However, all algorithms can be used for computing the corresponding error tolerant distance $d^\delta(A + t, B)$ in a given transposition t by running them on M_t^δ instead of on M. All the complexities for $\delta = 0$ will include a term of the

form mn, which has to be replaced by $\sum_{t\in\mathbb{T}} |M_t^\delta| \leq (2\delta+1)mn/\mu$ if $\delta > 0$. Note that a simple upper bound on the preprocessing time for general alphabets is $O(mn \log m + n \log n)$ for $\delta = 0$ and $O(mn(\log m + (2\delta+1)/\mu) + n \log n)$ in general.

5.2 Computing the Longest Common Subsequence

For LCS (and thus for d_{ID}) there exist algorithms that depend on r. The classical Hunt-Szymanski [26] algorithm has running time $O(r \log n)$ if the set of matches M is already given in the proper order. Using Lemma 1 we can conclude that there is an algorithm for transposition invariant LCS that has time complexity $O(mn \log n)$. There are even faster algorithms for LCS: Eppstein et al. [20] improved an algorithm of Apostolico and Guerra [2] achieving running time $O(D \log\log \min(D, \frac{mn}{D}))$, where $D \leq r$ is the number of dominant matches (see, for example, [2] for a definition). Using this algorithm, we have the bound $O(mn \log\log n)$ for the transposition invariant case (note that this is a tight estimate, since it can be achieved when $D = \Theta(mn/D)$ at each transposition).

The existing sparse dynamic programming algorithms for LCS, however, do not extend to the case of α–limited gaps. We will give a simple but efficient algorithm for LCS that generalizes to this case. We will also use the same technique when developing an efficient algorithm for the Levenshtein distance with α–limited gaps. Moreover, by replacing the data structure used in the algorithm by a more efficient one described in Lemma 8, we can achieve $O(r \log\log m)$ complexity, which gives $O(mn \log\log m)$ for the transposition invariant LCS (this 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 $\bar{M} = M \cup \{(0,0),(m+1,n+1)\}$. We have the following sparsity property for d_{ID}.

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*

$$d_{i,j} = \min\{d_{i',j'} + i - i' + j - j' - 2 \mid (i',j') \in \bar{M}, i' < i, j' < j\}, \qquad (4)$$

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

PROOF. Let us regard again the computation of matrix d as a shortest path computation on a graph. Every path from cell $(0,0)$ to a cell (i,j) that is the target of a zero-cost edge can be divided into two parts: (i) from cell $(0,0)$ until a cell (i',j') that is the target of the last zero-cost edge traversed before reaching (i,j), and 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). 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 adding cell $(0, 0)$ to $\bar{M}$, with $d_{0,0} = 0$ (which is also a way to state that our paths must start at cell $(0, 0)$). Finally, as we wish to obtain value $d_{m,n}$, we could have added cell (m, n) to $\bar{M}$, but our reasoning applies only to cells that are target of zero-cost edges. Hence, we add cell $(m+1, n+1)$ as such a target, so $d_{m,n} = d_{m+1,n+1}$ is correctly computed. □

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 only 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 data structure supporting dynamic one-dimensional range minimum queries. To see that it is enough to use query range $[-\infty, i)$, notice that if we compute points (i, j) column–by–column (that is, for increasing j), each column from bottom to up (that is, for decreasing i), then the query points that are in the range $[-\infty, i)$ are also those in the range $[-\infty, j)$. We call this order the *reverse column–by–column order*: (i', j') precedes (i, j) if $j' < j$, or if $j' = j$ and $i' > i$.

Hence we need an efficient data structure where we can store the row numbers i' as the sort keys, and values $v(i') = d(i', j') - i' - j'$ associated to them, and query it by minimum values over a range of keys. Furthermore, we will need later to remove points from this data structure, so we want it to be dynamic. The following well-known lemma establishes the existence of such a data structure. Lacking any reference, we prove it.

Lemma 4 *There is a data structure $\mathcal{T}$ supporting the following operations in $O(\log n)$ time, where n is the amount of elements currently in the structure.*

$\mathcal{T}.Insert(k, v)$: Inserts value v into the structure with key k. If key k already exists, the value of the element is updated to v if v is smaller than the current value.

$\mathcal{T}.Delete(k)$: Deletes the element with key k.

$v = \mathcal{T}.Minimum(I)$: Returns the minimum of values whose keys are in the

one-dimensional range $I = [\ell, r]$.

PROOF. A modified balanced binary search tree (AVL, for example) organized by keys k and storing associated values $v(k)$ is a suitable data structure. Let us speak indistinctly of nodes and keys, and denote left and right children of a node k by k.left and k.right. This tree is augmented with a field $minv(k)$ stored at each node, where the minimum of values in the subtree rooted at k is maintained. The tree is easily updated when a new key k is inserted, as the only additional operation is to update the value $minv(k')$ of any traversed internal node k' to $\min(minv(k'), v)$. Once a node k is deleted, values $minv(k')$ in the path from the root to the parent of k need to be recomputed (if the deleted node is internal and hence replaced by a leaf, this update is done from the parent of the removed leaf). This updating is easy since $minv(k) = \min(v(k), minv(k.\text{left}), minv(k.\text{right}))$ is recomputed in constant time per node. For the same reason, $minv(k)$ values are also easily recomputed when the tree is rebalanced by rotations.

Minimum over ranges of keys $[\ell, r]$ are obtained as follows. The tree is searched for ℓ and r simultaneously until node s^* is reached where the search path splits. From s^*.left the search is continued with ℓ and at every node s where the search path of ℓ goes left, value $minv(s.\text{right})$ is compared to the minimum value obtained so far. Similarly, the search is continued with r at s^*.right, and at every node s where the search path of r goes right, value $minv(s.\text{left})$ is considered for updating the computed minimum. Also, the $v(k)$ values of nodes k visited are included in the minimization whenever $\ell \leq k \leq r$. A not so infrequent special case occurs when the search path splits before the root node, and hence node s^* does not exist. In this case both searches for ℓ and r start at the root node. □

We are ready to give the algorithm now. Initialize the tree $\mathcal{T}$ of Lemma 4 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 reverse column–by–column order and make the following operations at each such 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 current row and column number, $i + j$, into $\mathcal{T}$, with the current row number as its key: $\mathcal{T}.Insert(i, d - i - j)$.

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

The above algorithm works correctly: The reverse column–by–column evaluation and the range query restricted by the row number in $\mathcal{T}$ guarantee that

conditions $i' < i$ and $j' < j$ hold. The only point where the work on tree $\mathcal{T}$ deviates from what Lemma 3 requires is that new keys overwrite equal old keys. That is, if a new cell (i, j) is inserted, an old cell (i, j') is virtually removed if it existed. It is easy to see that the old cell is of no use once the new cell is inserted. Say that cell (i, j') obtained its value from cell (i_0, j_0), so that $d_{i,j'} = d_{i_0,j_0} + i - i_0 + j' - j_0 - 2$. Hence cell (i_0, j_0) is also a candidate to $d_{i,j} \leq d_{i_0,j_0} + i - i_0 + j - j_0 - 2$, so $d_{i,j} \leq d_{i',j'} + j - j'$. Now, assume a later cell (i'', j'') uses cell (i, j'), so that $d_{i'',j''} = d_{i,j'} + i'' - i + j'' - j' - 2$. But then it can also use cell (i, j) to obtain a smaller or equal value using $d_{i'',j''} = d_{i,j} + i'' - i + j'' - j - 2 \leq d_{i',j'} + i'' - i + j'' - j' - 2$. Note that this is simply a consequence of the fact that cell (i, j) dominates (i, j') [2].

Clearly, the time complexity of the algorithm is $O(r \log m)$, where $r = |M|$, since we can only have $m+1$ different row numbers stored in $\mathcal{T}$ at any moment. Figure 1 gives an example.

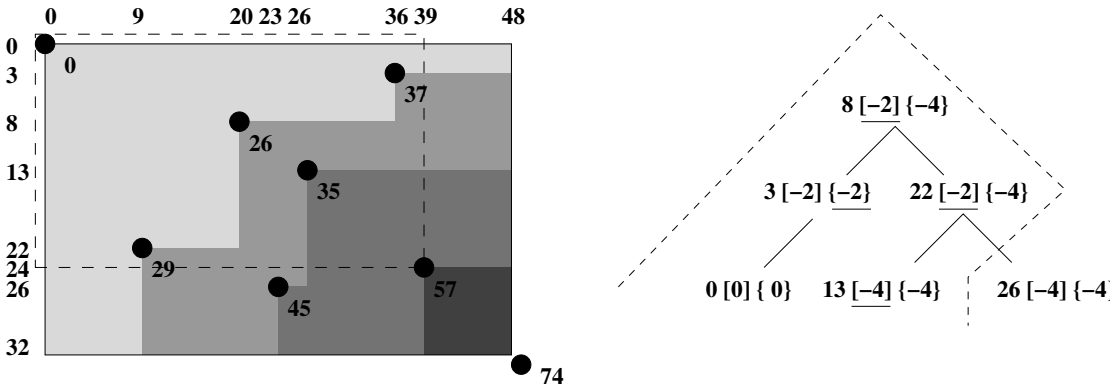

Fig. 1. Example of computation of d_{ID} on a sparse matrix. Black circles represent the matching pairs (i, j). Each such matrix position has an influence area represented by a gray rectangle (darker grays represent larger differences from the standard value $i + j$). Next to each position we represent the matrix value $d_{i,j}$ we compute. The value of interest is the lowest rightmost position. In particular, we depict the computation of the cell (24,39), for which we have to consider all the positions included in the dashed rectangle. On the right we show our tree data structure. Each node corresponds to a cell (i, j) and is represented as i $[v]$ $\{minv\}$, where i is the tree key, v is the value (meaning that the real cell value is $(i + j) + v$), and $minv$ is the minimum v value in the subtree. The search for cell (24,39) includes all the nodes below the dashed line, and it takes the minimum d over all the underlined values. Its new value is $d_{24,39} = d + 24 + 39 - 2 = 57$, so we will insert a new node with key 24 and value $57 - 24 - 39 = -6$ in the tree.

The algorithm also generalizes easily to the search problem: The 0 values in the first row can be added implicitly by using $d \leftarrow \min(i, d + i + j - 2)$ in step (1) above. Also, every value $d_{i,j} = d$ computed in step (2) above induces a value $d_{m,j+s} \leq d + (m - i) + s$ in the last row, which can be used either to keep the minimum $d_{m,j}$ value (in which case we consider only case $s = 0$), or to report all values $d_{m,j} \leq k$ in thresholded searching. In order to report occurrences

only once and in order, two arrays $S(1 \ldots n)$ and $E(1 \ldots n)$ of counters are maintained: The counters are initialized to zero, and at each pair $(i, j) \in \bar{M}$ such that $d_{i,j} + (m - i) \leq k$ we set $S(j) = S(j) + 1$ and $E(j + s) = E(j + s) + 1$ for the maximum s such that $d + (m - i) + s \leq k$. This marks the start and end points of the occurrences. Then it is easy to collect all the occurrences in $O(n)$ time by using $S()$ and $E()$ to keep track on how many ranges are active at any position j of the text.

The queries $[-\infty, i)$ we use are semi-infinite. We will show in Lemma 8 (Section 5.3) that the balanced binary search tree can be replaced by a more advanced data structure in this case. That is, semi-infinite queries for minimum and insertions can be supported in amortized $O(\log \log u)$ time, where $[1, u]$ is the integer range of keys that are inserted into the structure. In our case $u = m$, which gives us $O(\log \log m)$ query time. The next theorem follows immediately.

Theorem 5 *Given two strings $A = a_1 \ldots a_m$ and $B = b_1 \ldots b_n$, $m \leq n$, and the r cells (i, j) such that $a_i = b_j$ in reverse column–by–column order, then the LCS between A and B can be computed in time $O(r \log \log \min(r, m))$.*

Let us now consider the case with α–limited gaps. The only change we need in our algorithm 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 search range $[-\infty, i)$ into $[i - \alpha - 1, i)$ in $\mathcal{T}$, as well as to delete any elements in column $j - \alpha - 1$ after processing elements in column j. The former is easily accomplished by using query $\mathcal{T}.Minimum([i - \alpha - 1, i))$ at step (1) of the algorithm. The latter needs that we delete nodes from $\mathcal{T}$ when their columns become too old. More specifically, we maintain a pointer to the oldest (that is, smallest column) element in M that is still stored in $\mathcal{T}$. When we finish processing column j, we check whether the pointed cell is of the form $(i', j - \alpha - 1)$ for some i'. If it is, we remove key i' using $\mathcal{T}.Delete(i')$ and advance the pointer until the pointed cell belongs to a later column. Since the tracking takes constant time per cell of M, its effect in the complexity is negligible.

Note that it might be that key i' in $\mathcal{T}$ actually corresponds to a later column that has overwritten cell $(i', j - \alpha - 1)$. In this case we must advance the pointer but not delete the key. In order to check this, we also store in $\mathcal{T}$ nodes the current j' value corresponding to each key i'.

Notice that we cannot obtain $O(\log \log m)$ query time anymore, since the query ranges are no longer semi-infinite. On the other hand, we could have used two-dimensional queries instead of deleting points from $\mathcal{T}$ but, as shown in Lemma 8, the complexity would be worse. An illustration of the algorithm for LCS with α–limited gaps is given in Figure 2.

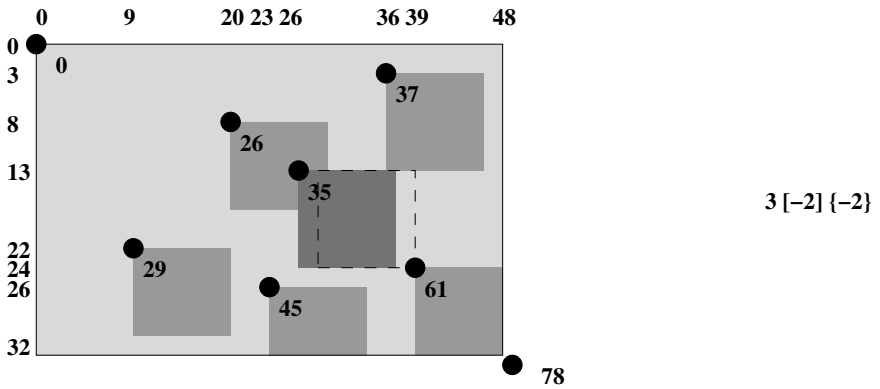

Fig. 2. Example of α–gapped computation of d_{ID} on a sparse matrix, for $\alpha = 10$. The same conventions of Figure 1 apply. The difference is that now the influence areas are restricted to width and height α, so we delete values which correspond to column numbers which are small enough to have become irrelevant and perform a two-sided range search over the tree, so only its middle part qualifies. In this example, the tree has only one element when computing cell (24,39), and it is outside the search range. In this case the value of the cell is $i + j - 2 = 61$.

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

Theorem 6 *The transposition invariant distance $d^{\mathrm{t}}_{\mathrm{ID}}(A, B)$ (or, equivalently, LCS) can be computed in $O(mn \log\log m)$ time. The corresponding search problem can be solved within the same time bound. For the distance $d^{\mathrm{t},\alpha}_{\mathrm{ID}}(A, B)$ the time bounds are $O(mn \log m)$ for distance computation and for searching. The preprocessing cost of Theorem 2 must be added to these bounds.*

The algorithms use $O(mn)$ space, since the overall size of the sets for different transpositions is mn (note that the algorithm itself needs only $O(m)$ space). This might be problematic especially for the search problem, when the two strings are of very different size.

We can achieve space complexity $O(m^2)$ in the search problem as follows. Divide the text into $O(n/m)$ segments of the form $T_{1\ldots 2m}$, $T_{m+1\ldots 3m}$, $T_{2m+1\ldots 4m}$, and so on. Run the whole algorithm (including generating the sets of transpositions) separately over those $O(n/m)$ text segments, one after the other. When processing text segment $T_{mi+1\ldots m(i+2)}$, report the matches found in the area $T_{m(i+1)+1\ldots m(i+2)}$. This way, each text position is processed twice and hence the complexity remains the same. The space, however, is that to process one text segment, $O(m^2)$. With respect to correctness, we remark that, given that cell $d_{i,j}$ receives value i from cell $(0, j)$, no column before $j - m$ can influence it (indeed, no column before $j - i$). Hence, in order to report correctly the matches in area $T_{m(i+1)+1\ldots m(i+2)}$ we only need to start m positions behind, thus processing area $T_{mi+1\ldots m(i+2)}$. This technique is rather general and can be applied to other edit distances as well.

In particular, in the case of α–limited gaps we can use the same technique

both for distance computation and for searching, since only the last α columns processed can affect current values. Hence we can compute $d_{\mathrm{ID}}^{\mathrm{t},\alpha}(A,B)$ using $O(\alpha m)$ space.

We recall that, when $\delta > 0$ and we consider distances $d_{\mathrm{ID}}^{\mathrm{t},\delta}$ and $d_{\mathrm{ID}}^{\mathrm{t},\delta,\alpha}$, all terms mn are replaced by $\sum_{t\in\mathbb{T}} |M_t^\delta|$ in the time and space complexities.

5.3 Computing the Levenshtein Distance

For the Levenshtein distance, there exists an $O(r \log\log \min(r, mn/r))$ sparse dynamic programming algorithm [20,24]. Using this algorithm, the transposition invariant case can be solved in $O(mn \log\log n)$ time. As with the LCS, this algorithm does not generalize to the case of α–limited gaps. We develop an alternative solution for the Levenshtein distance by generalizing our LCS range query approach. This new algorithm can be further generalized to solve the problem of α–limited gaps. On the other hand, we show that the sparse computation can be done in $O(r \log\log m)$ time.

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.

Lemma 7 *Distance $d_{\mathrm{L}}(A,B)$ can be computed by evaluating $d_{i,j}$ for $(i,j) \in \bar{M}$ using the recurrence*

$$d_{i,j} = \min \begin{cases} \{d_{i',j'} + j - j' - 1 \mid (i',j') \in \bar{M}, i' < i, j' - i' < j - i\} \\ \{d_{i',j'} + i - i' - 1 \mid (i',j') \in \bar{M}, j' < j, j' - i' \geq j - i\} \end{cases} \tag{5}$$

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

PROOF. Following the proof of Lemma 3 it is enough to show that the minimum path cost to reach cell $(i-1, j-1)$ from match point (i',j') is (i) $j - j' - 1$ when $j' - i' < j - i$, and (ii) $i - i' - 1$ 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 recurrence relation is now 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 $d_{i',j'} - i' - j'$ of $(i',j') \in [-\infty, i) \times [-\infty, j)$ 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 $j - j' - 1$.

Figure 3 illustrates the geometric setting implicit in (5). The lower region (below diagonal $j-i$) contains match points such that their extension by match (i,j) will add $j-j'-1$ to the score, and the upper region (above diagonal) contains match points such that their extension by match (i,j) will add $i-i'-1$ to the score. The score of the new match is computed as the minimum between the lowest possible score obtained by extending a match from the lower region and from the upper region. Therefore, each match will have its scores maintained in two structures, one structure representing scores to be extended as "lower region" scores, and other for "upper region" extensions.

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 $\mathcal{L}$, that is, 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 lower region minima.

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 search tree $\mathcal{U}$ with key $j'-i'$, 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 reverse column–by–column traversal guarantees that $\mathcal{U}$ only contains values $d_{i',j'}-i'$ whose coordinates satisfy $j'<j$. Thus, the upper region can be handled efficiently.

The problem 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 seems to allow us to limit to one-dimensional range searches in both regions simultaneously; we will need two-dimensional range searching 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 turn this triangle region into a rectangle: We map each value $d_{i',j'}-j'$ into an xy-plane at coordinate $i', j'-i'$. In this plane we perform a rectangle query $[-\infty,i)\times[-\infty,j-i)$. The following lemma, adapted from Gabow, Bentley and Tarjan [23], provides the required data structure for the lower region. We summarize some other related results in the same lemma that we will soon use in the α–limited case (we already referred to the one-dimensional result in the algorithm for d_{ID}).

Lemma 8 (Gabow, Bentley, Tarjan [23]) *There is a data structure $\mathcal{R}$ that stores a two-dimensional point-set S with a value associated to each point, and supports the following operations in amortized $O(\log n \log\log n)$ time after*

$O(n \log n)$ time preprocessing on S, where $n = |S|$:

- *$\mathcal{R}.Update(x, y, v)$: Update value of point $s = (x, y) \in S$ to v, under the condition (*) that the current value of s is larger than v.*
- *$v = \mathcal{R}.Minimum(I)$: Retrieve the minimum value from a range I of S, where I is semi-infinite at least in one fixed coordinate.*

There is another structure $\mathcal{P}$ that supports the same operations in $O(\log^2 n)$ time, where condition (*) does not need to hold, and search range I needs not be semi-infinite in either coordinate

Semi-infinite queries can be supported in $O(\log \log n)$ time in the one-dimensional case, if the point coordinates $s \in S$ are integers in the range $[1, n]$. In this case condition (*) must hold.

PROOF. We will review the proof of the $O(\log n \log \log n)$ bound [23] in order to cover the one-dimensional case and the closed range case.

The basic structure supporting operations in time $O(\log^2 n)$ is a range tree (see, for example, [3, Section 5]), where the secondary structures are replaced by the ones given in Lemma 4. The structure is a balanced (primary) search tree for the x-coordinate range searches, where each node w stores another (secondary) balanced tree for y-coordinate searches among the points that are stored in the subtree of w in the primary tree. As shown in Lemma 4, the secondary trees support minimum queries and unrestricted updates of values. To update a value, its node in the primary tree is found and then it is necessary to update the corresponding nodes in all the $O(\log n)$ secondary trees stored at the ancestors of the primary tree node. For range searching, we find in $O(\log n)$ time the $O(\log n)$ nodes of the primary tree whose subtrees cover the x-coordinate range, and then pay $O(\log n)$ time in each such node to find the minimum of points in the y-coordinate range. Hence, updating and searching can be done in $O(\log^2 n)$ time. Note that it is costly to maintain the invariants of the secondary trees contents upon rebalancing the primary tree, so insertions and deletions of points are not supported. Rather, the trees are built in a preprocessing stage in perfectly balanced form and stay with that shape. Preprocessing cost is proportional to the space needed by the data structure, which is $O(n \log n)$.

Let us then review how $O(\log \log n)$ time can be achieved in the one-dimensional case for integer point sets. As our query is w.l.o.g. $\min\{v(s) \mid s \in [-\infty, r)\}$, where $v(s)$ gives the value of s, it is enough to choose the minimum among those points s whose value $v(s)$ is the minimum in the range $[-\infty, s]$; these are called *left–to–right minima*. It is easy to see that other values $v(s)$ can never be the minimum in any range $[-\infty, r)$. Note that left–to–right minima form a decreasing sequence. The data structure of van Emde

Boas [37,38], which we will denote $\mathcal{Q}$, supports operations $\mathcal{Q}.insert(s)$ (inserts s into $\mathcal{Q}$), $\mathcal{Q}.delete(s)$ (deletes s from $\mathcal{Q}$), $\mathcal{Q}.successor(s)$ (returns the largest point stored in $\mathcal{Q}$ smaller than s), and $\mathcal{Q}.predecessor(s)$ (returns the smallest point stored in $\mathcal{Q}$ larger than s) in $O(\log\log n)$ time, where s is an integer in the range $[1, n]$. We will store only left–to–right minima from S in $\mathcal{Q}$. When inserting a new point s with value $v = v(s)$ into $\mathcal{Q}$, we first check that $v(\mathcal{Q}.predecessor(s)) > v(s)$, otherwise we do not insert s. If s is inserted, we repeat operation $\mathcal{Q}.delete(\mathcal{Q}.successor(s))$ until $v(\mathcal{Q}.successor(s)) < v(s)$. These operations guarantee that $v(\mathcal{Q}.predecessor(r))$ is the answer to our query $[-\infty, r)$. Note that it is possible to replace the value v of an already inserted point by a smaller value, by a process similar to insertion, but we cannot change v to a larger value.

The $O(\log n \log\log n)$ bound for the semi-infinite two-dimensional queries then follows easily by replacing the secondary trees of the range tree with data structures $\mathcal{Q}$: Consider a query $[l, r] \times [-\infty, t]$. We build the primary tree on the x-coordinates and the secondary trees on the y-coordinates. Instead of adding the y-coordinates as such, we use the rank of each point in the sorted order of the points where y-coordinate is used as the primary key and x-coordinate as the secondary key. To answer the query, we find the rank ρ of (t, ∞) (place where it would be inserted) in the sorted set of points by binary search in time $O(\log n)$, then query each of the $O(\log n)$ secondary structures Q found by the x-coordinate range search with $s = \mathcal{Q}.predecessor(\rho)$, and select the minimum $v(s)$. □

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 8) with the triples $(i, j - i, \infty)$ 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 reverse 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 than 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 $O(r \log r \log\log r)$ ($r = |M|$ elements are inserted and updated in the lower region structure, and r times it is queried). The space usage is $O(r \log r)$. Figure 3 gives an example.

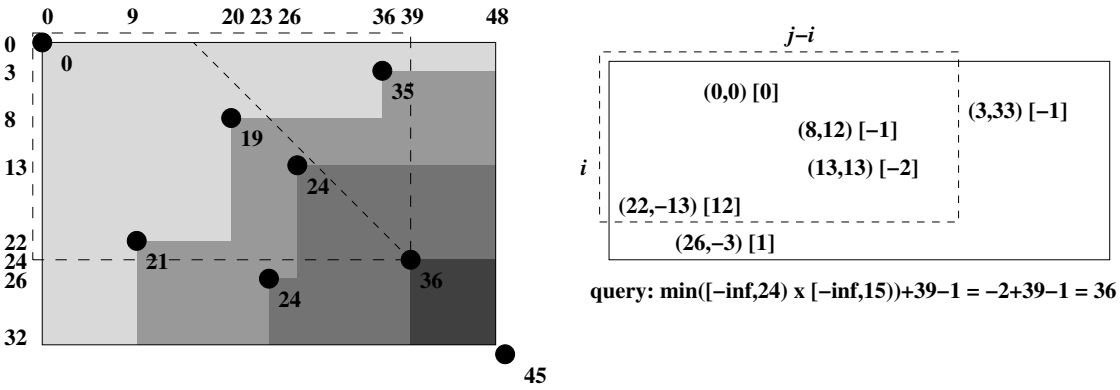

Fig. 3. Example of computation of d_{L} on a sparse matrix. The same conventions of Figure 1 apply. We distinguish in the matrix the lower and upper regions considered to solve cell (24,39). Since the upper region is handled just like for d_{ID}, we show on the right only the data structure of the lower region. It supports minimum operations over two dimensional ranges. Each relevant matrix position (i, j) is represented in the range search structure at position $(i, j-i)$. The value in brackets is $[y-j]$, where y is the value of cell (i, j). To solve cell (24,39) we take the minimum in the range $[-\infty, 24) \times [-\infty, 39-24)$ (inside the dashed rectangle on the right), which returns -2, and add $j-1$ to it to obtain 36. After this, point (24,15) will be updated to value $36 - 39 = -3$.

Actually, we can switch the roles of x and y in $\mathcal{L}$, so that the secondary structures are searched for i values. As explained in Section 5.2 we do not need to store different points with the same i coordinate in the secondary structures; it is enough to retain the last point inserted with coordinate i, since it dominates previous ones (that is, the new value we are inserting is never larger than the existing points with coordinate i). As we have shown in the proof of Lemma 8, the structure permits us replacing the value of a point with a new, smaller, one. Hence we can in fact store only unique coordinates in the range $0 \ldots m$, each associated to the last (that is, smallest) value $v(i)$ inserted so far. The advantage is that the time complexity becomes $O(r \log r \log\log \min(r, m))$. Moreover, we do not need to rank the points, but can directly search the i values.

The algorithm can be modified for the search problem similarly as d_{ID}, by implicitly adding values 0 in the first row of the current column and considering the effect of each computed $d_{i,j}$ value in the last row of the matrix. Now cell (i, j) induces values $d_{m,j+s} \leq d_{i,j} + \max(m-i, s)$. Applying the same text segmenting technique used for distance d_{ID} yields $O(r \log m \log\log m)$ time,

slightly better for our purposes than distance computation.

We show now a general technique to make distance computation $O(r \log m \log\log m)$ time as well. Segment the text into $O(r/m)$ regions, such that each text region contains between m and $2m$ cells in $\bar{M}$ (we must be flexible because there may be several cells in a column). Run the algorithm for each region separately, one after the other. At the end of each region, insert cells in $\bar{M}$ so that $\bar{M}$ covers all the cells of the last column of the region. Use those last values to initialize the data structure for the next region (via cell updates). This ensures continuity in the computation across regions. Overall we process at most $3r$ cells, and each region contains $O(m)$ cells, so the search time is $O(r \log m \log\log m)$. We observe that the same time complexity would be obtained if we used regions of $O(m^c)$ entries, for any constant c.

Using this algorithm, the transposition invariant Levenshtein distance computation, as well as the search problem, can be solved in $O(mn \log m \log\log m)$ time and $O(mn \log n)$ space. Note that in this case the space complexity is dominated by the data structure $\mathcal{L}$. Removing unnecessary elements (those that cannot give minima for the current column) is no longer possible, since the structure for the lower region is semi-static.

With the techniques used for splitting the text into regions, however, the data structure $\mathcal{L}$ needs only $O(m^2 \log m)$ space. Distance computation still needs $O(mn)$ additional space to store the transpositions. We cannot, as in the text segmenting approach used for searching, process the transpositions region by region to obtain $O(m^2)$ space, because this time region limits are different for each transposition and we need to remember the state of the computation for every different transposition.

We recall that the sparse dynamic programming algorithm by Eppstein et al. [20] is better than ours, $O(r \log\log r)$. Our text regions approach, however, permits improving Eppstein's algorithm. We can use the latter as a black box and apply it over text regions as with our algorithm. The result is given in the next theorem.

Theorem 9 *Given two strings $A = a_1 \ldots a_m$ and $B = b_1 \ldots b_n$, $m \leq n$, and the r cells (i, j) such that $a_i = b_j$ in reverse column–by–column order, then the Levenshtein distance between A and B can be computed in time $O(r \log\log m)$.*

Using this theorem, the time complexity for transposition invariant Levenshtein distance computation decreases to $O(mn \log\log m)$.

Our range query approach, although slower, has the advantage of letting us easily solve the case of α–limited gaps. First consider the easier upper 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 $j' - i' \geq j - i$. These can be simplified

to $j' < j$ (which comes for free with the reverse column–by–column order), $i' \geq i - \alpha - 1$ and $j' - i' \geq j - i$. We can use structure $\mathcal{R}$ of Lemma 8 to support minimum queries in the range $[i - \alpha - 1, \infty] \times [j - i, \infty]$. The lower region is more complicated. Its limiting conditions, $i' \in [i - \alpha - 1, i)$, $j' \in [j-\alpha-1, j)$, and $j'-i' < j-i$, can be simplified to $i' < i$, $j' \geq j-\alpha-1$ and $j'-i' < j-i$. Instead of resorting to three-dimensional searching, which would cost $O(\log^2 n \log\log n)$ [23], we use structure $\mathcal{P}$ of Lemma 8, which supports unlimited updates of values. Once moving from column j to $j+1$, we update each value in the secondary structures at column $j - \alpha - 1$ to ∞. As in the α–limited case of d_{ID}, we keep a pointer to the last active column in the match set M to determine which cells $(i', j-\alpha-1)$ have to be virtually deleted using $\mathcal{P}.Update(i', j - \alpha - 1 - i,' \infty)$. If we do this, condition $j' \geq j - \alpha - 1$ can be ignored, and $\mathcal{P}$ is built over the other two conditions and queried with range $[-\infty, i) \times [-\infty, j - i)$.

Again, text segmenting techniques can be used to maintain time complexities in $O(r \log^2 m)$. An illustration of the algorithm for Levenshtein distance with α–limited gaps is given in Figure 4.

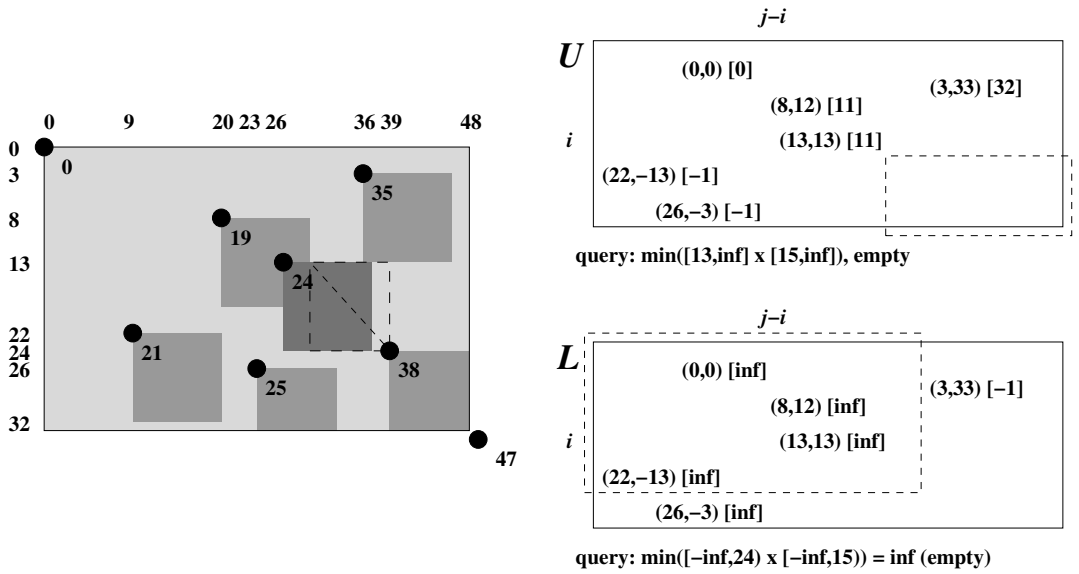

Fig. 4. Example of computation of α–gapped d_{L} on a sparse matrix. The same conventions of Figure 3 apply. On the right we show now both two-dimensional range search structures, $\mathcal{U}$ and $\mathcal{L}$. To solve cell (24,39) we take the minimum in the range $[24, \infty] \times [15, \infty]$ on $\mathcal{U}$ and $[-\infty, 24) \times [-\infty, 15)$ on $\mathcal{L}$. The area in $\mathcal{U}$ is empty, and that in $\mathcal{L}$ is virtually empty because we have set old column cell values to ∞.

Combining Lemma 1 with the above results, we obtain the following bounds for the transposition invariant case.

Theorem 10 *Transposition invariant Levenshtein distance $d_{\mathrm{L}}^{\mathrm{t}}(A, B)$ can be computed in $O(mn \log\log m)$ time. The corresponding search problem can be solved within the same time bounds. For the case of α–limited gaps, $d_{\mathrm{L}}^{\mathrm{t},\alpha}(A, B)$,*

the time requirements are $O(mn \log^2 m)$, both for distance computation and for searching. The preprocessing cost of Theorem 2 must be added to these bounds.

As before, the space complexity is $O(m^2 \log m)$ plus that of storing the sets M_t, that is, $O(mn)$ for distance computation and $O(m^2)$ for searching. Also, the α–limited version can be solved using $O(\alpha m)$ space. In case $\delta > 0$, the mn in the complexities becomes $\sum_{t \in \mathbb{T}} |M_t^\delta|$.

5.4 Episode Matching

To conclude the edit distance section we look at the episode matching problem and $d_{\mathrm{D}}^{\mathrm{t}}$ distance, which have 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 using similar arguments as in Lemma 3, since the last zero-cost edge in a path to (i,j) must be in row $i-1$.

Lemma 11 *Distance $d_{\mathrm{D}}(A,B)$ can be computed by evaluating $d_{i,j}$ for $(i,j) \in \bar{M}$ using the recurrence*

$$d_{i,j} = \min\{d_{i-1,j'} + j - j' - 1 \mid j' < j, (i-1, j') \in \bar{M}\}, \tag{6}$$

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 reverse column–by–column order. We maintain for each row i' a value $d(i')$ that gives the minimum $d_{i',j'} - j'$ value seen so far in that row among pairs $(i', j') \in \bar{M}$. First, initialize $d(0) = 0$ and $d(i) = \infty$ for $1 \le i \le m$. Let $(i,j) \in \bar{M}$ be the current pair whose value we need to evaluate. Then $d = d_{i,j}$ can simply be computed as $d = j - 1 + d(i-1)$, since $j - 1 + d(i-1) = j - 1 + \min\{d_{i-1,j'} - j' \mid j' < j, (i-1, j') \in \bar{M}\}$ (condition $j' < j$ holds because (i,j) precedes $(i-1, j)$ in reverse column–by–column order). After $d = d_{i,j}$ is computed, we can safely update the row minimum $d(i) = \min(d(i), d - j)$. The algorithm takes overall $O(|\bar{M}|) = O(r)$ time.

The above algorithm generalizes to the search problem (that is, to episode matching) by implicitly considering all values $d_{0,j}$ as zero for all j. That is, $d(0)$ is assumed to be $j - 1$ if a cell $d_{1,j}$ is being processed. The problem of α–limited gaps can also be handled easily. Let $c(i-1)$ give the last column j' where $d(i-1)$ has been updated (even if its value stayed the same). One easily notices that $c(i-1)$ is always the last match $(i-1, j')$ seen so far in that row. Therefore, 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 the following result.

Theorem 12 *The transposition invariant computation of distance $d_{\mathrm{D}}^{\mathrm{t}}(A, B)$, as well as transposition invariant episode matching, can be solved in $O(mn)$ time. The same bound applies in the case of α–limited gaps. The preprocessing cost of Theorem 2 must be added to these bounds.*

Note again that the algorithm needs only $O(m)$ space, but the overall space is $O(mn)$, because of the need to store the transpositions. It is interesting that in this case we cannot reduce the space to $O(m^2)$ for the search problem, as it is not true anymore that the previous m columns define the matrix contents. On the other hand, in the case of α–limited gaps we still can use $O(\alpha m)$ space.

6 Transposition Invariant Hamming Distance and Variants

So far we have seen that sparse dynamic programming is the key in solving transposition invariant distance computation problems. It could be used to solve other simpler distances such as Hamming distance. However, for such simpler distance measures, it is possible to find the optimal transposition directly, and do the distance computation only for that transposition. To demonstrate this, we consider in this section the computation of some error tolerant versions of Hamming, SAD and MAD distances, where the strings are aligned position-wise (a_i with b_i) and hence have the same length.

For this section, let us redefine $\mathbb{T} = \{t_i = b_i - a_i \mid 1 \leq i \leq m\}$ as the set of transpositions that make some characters a_i and b_i match. Note that the optimal transposition does not need, in principle, to be included in $\mathbb{T}$, but we will show that this is the case for $d_{\mathrm{H}}^{\mathrm{t}}$ and $d_{\mathrm{SAD}}^{\mathrm{t},\kappa}$. Note also that $|\mathbb{T}| = O(|\Sigma|)$ on an integer alphabet and $|\mathbb{T}| = O(m)$ in any case.

6.1 Hamming Distance

Let $A = a_1 \ldots a_m$ and $B = b_1 \ldots b_m$, where $a_i, b_i \in \Sigma$ for $1 \leq i \leq m$. We consider the computation of transposition invariant Hamming distance $d_{\mathrm{H}}^{\mathrm{t},\delta}(A, B)$. That is, we search for a transposition t maximizing the size of set $\{i \mid |b_i - (a_i + t)| \leq \delta, 1 \leq i \leq m\}$.

Theorem 13 *Given two numeric strings A and B, both of length m, there is an algorithm for computing distance $d_{\mathrm{H}}^{\mathrm{t},\delta}(A, B)$ in $O(|\Sigma| + m)$ time on an integer alphabet, or in $O(m \log m)$ time on a general alphabet.*

PROOF. It is clear that the Hamming distance is minimized for the transposition in $\mathbb{T}$ that makes the maximum number of characters match. What

follows is a simple voting scheme, where the most voted transposition wins. Since we allow a tolerance δ in the matched values, t_i votes for range $[t_i - \delta, t_i + \delta]$. Construct sets $S = \{(t_i - \delta, \text{“open”}) \mid 1 \leq i \leq m\}$ and $E = \{(t_i + \delta, \text{“close”}) \mid 1 \leq i \leq m\}$. Sort $S \cup E$ into a list I using order

$$(x', y') <^{\mathrm{H}} (x, y) \text{ if } x' < x \text{ or } (x' = x \text{ and } y' < y),$$

where “open”<“close”. Initialize variable $count = 0$. Do for $i = 1$ to $|I|$ if $I(i) = (x, \text{“open”})$ then $count = count+1$ else $count = count-1$. Let $maxcount$ be the largest value of $count$ in the above algorithm. Then clearly $d_{\mathrm{H}}^{\mathrm{t},\delta}(A, B) = m - maxcount$, and the optimal transposition is any value in the range $[x_i, x_{i+1}]$, where $I(i) = (x_i, *)$, for any i where $maxcount$ is reached. The complexity of the algorithm is $O(m \log m)$. Sorting can be replaced by array lookup when Σ is an integer alphabet, which gives the bound $O(|\Sigma| + m)$ for that case. □

6.2 Sum of Absolute Differences Distance

We shall first look at the basic case where $\kappa = 0$. That is, we search for a transposition t minimizing $d_{\mathrm{SAD}}(A + t, B) = \sum_{i=1}^{m} |b_i - (a_i + t)|$.

Theorem 14 *Given two numeric strings A and B, both of length m, there is an algorithm for computing distance $d_{\mathrm{SAD}}^{\mathrm{t}}(A, B)$ in $O(m)$ time on both integer and general alphabets.*

PROOF. Let us consider $\mathbb{T}$ as a multiset, where the same element can repeat multiple times. Then $|\mathbb{T}| = m$, since there is one element in $\mathbb{T}$ for each $b_i - a_i$, where $1 \leq i \leq m$. Sorting $\mathbb{T}$ in ascending order gives a sequence $t_{i_1} \leq t_{i_2} \leq \ldots \leq t_{i_m}$. Let t_{opt} be the optimal transposition. We will prove by induction that $t_{opt} = t_{i_{\lfloor m/2 \rfloor + 1}}$, that is, the optimal transposition is the median transposition in $\mathbb{T}$.

To start the induction we claim that $t_{i_1} \leq t_{opt} \leq t_{i_m}$. To see this, notice that $d_{\mathrm{SAD}}(A + (t_{i_1} - \epsilon), B) = d_{\mathrm{SAD}}(A + t_{i_1}, B) + m\epsilon$, and $d_{\mathrm{SAD}}(A + (t_{i_m} + \epsilon), B) = d_{\mathrm{SAD}}(A + t_{i_m}, B) + m\epsilon$, for any $\epsilon \geq 0$.

Our induction assumption is that $t_{i_k} \leq t_{opt} \leq t_{i_{m-k+1}}$ for some k. We may assume that $t_{i_{k+1}} \leq t_{i_{m-k}}$, since otherwise the result follows anyway. First notice that, independently of the value of t_{opt} in the above interval, the cost $\sum_{l=1}^{k} |b_{i_l} - (a_{i_l} + t_{opt})| + \sum_{l=m-k+1}^{m} |b_{i_l} - (a_{i_l} + t_{opt})|$ will be the same. Then notice that $\sum_{l=k+1}^{m-k} |b_{i_l} - (a_{i_l} + t_{i_{k+1}} - \epsilon)| = \sum_{l=k+1}^{m-k} |b_{i_l} - (a_{i_l} + t_{i_{k+1}})| + (m - 2k)\epsilon$, and $\sum_{l=k+1}^{m-k} |b_{i_l} - (a_{i_l} + t_{i_{m-k}} + \epsilon)| = \sum_{l=k+1}^{m-k} |b_{i_l} - (a_{i_l} + t_{i_{m-k}})| + (m - 2k)\epsilon$. This completes the induction, since we showed that $t_{i_{k+1}} \leq t_{opt} \leq t_{i_{m-k}}$.

The consequence is that $t_{i_k} \leq t_{opt} \leq t_{i_{m-k+1}}$ for maximal k such that $t_{i_k} \leq t_{i_{m-k+1}}$, that is, $k = \lceil m/2 \rceil$. When m is odd, it holds $m - k + 1 = k$ and there is only one optimal transposition, $t_{i_{\lceil m/2 \rceil}}$. When m is even, one easily notices that all transpositions t_{opt}, $t_{i_{m/2}} \leq t_{opt} \leq t_{i_{m/2+1}}$, are equally good. Finally, the median can be found in linear time [4]. □

To get a fast algorithm for $d_{\mathrm{SAD}}^{\mathrm{t},\kappa}$ when $\kappa > 0$ largest differences can be discarded, we need a lemma that shows that the distance computation can be incrementalized from one transposition to another. Let $t_{i_1}, t_{i_2}, \ldots, t_{i_m}$ be the sorted sequence of $\mathbb{T}$.

Lemma 15 *Once values S_j and L_j such that $d_{\mathrm{SAD}}(A + t_{i_j}, B) = S_j + L_j$, $S_j = \sum_{j'=1}^{j-1} t_{i_j} - t_{i_{j'}}$, and $L_j = \sum_{j'=j+1}^{m} t_{i_{j'}} - t_{i_j}$, are computed, the values of S_{j+1} and L_{j+1} can be computed in $O(1)$ time.*

PROOF. Value S_{j+1} can be written as

$$S_{j+1} = \sum_{j'=1}^{j} t_{i_{j+1}} - t_{i_{j'}} = \sum_{j'=1}^{j} t_{i_{j+1}} - t_{i_j} + t_{i_j} - t_{i_{j'}} = j(t_{i_{j+1}} - t_{i_j}) + S_j.$$

Similar rearranging gives

$$L_{j+1} = \sum_{j'=j+2}^{m} t_{i_{j'}} - t_{i_{j+1}} = (m - j)(t_{i_j} - t_{i_{j+1}}) + L_j.$$

Thus both values can be computed in constant time given the values of S_j and L_j (and $t_{i_{j+1}}$). □

Theorem 16 *Given two numeric strings A and B both of length m, there is an algorithm for computing distance $d_{\mathrm{SAD}}^{\mathrm{t},\kappa}(A, B)$ in $O(m + \kappa \log \kappa)$ time on both integer and general alphabets. On integer alphabets, time $O(|\Sigma| + m + \kappa)$ can also be obtained.*

PROOF. Consider the sorted sequence $t_{i_1}, t_{i_2}, \ldots, t_{i_m}$ as in the proof of Theorem 14. Clearly the candidates for the κ outliers (largest differences) are $M(k', k'') = \{t_{i_1}, \ldots, t_{i_{k'}}, t_{i_{m-k''+1}}, \ldots t_{i_m}\}$ for some $k' + k'' = \kappa$. The naive algorithm is then to compute the distance in all these $\kappa + 1$ cases: Compute the median of $\mathbb{T} \setminus M(k', k'')$ for each $k' + k'' = \kappa$ and choose the minimum distance induced by these medians. These $\kappa + 1$ medians can be found as follows: First select values $t_{\kappa+1}$ and $t_{m-\kappa}$ using the linear time selection algorithm [4]. Then collect and sort all values smaller than $t_{\kappa+1}$ or larger than

$t_{m-\kappa}$. After selecting the median $m_{0,\kappa}$ of $\mathbb{T} \setminus M(0,\kappa)$ and $m_{\kappa,0}$ of $\mathbb{T} \setminus M(\kappa,0)$, one can collect all medians $m_{k',k''}$ of $\mathbb{T} \setminus M(k',k'')$ for $k'+k''=\kappa$, since the $m_{k',k''}$ values are those between $m_{0,\kappa}$ and $m_{\kappa,0}$. The $\kappa+1$ medians can thus be collected and sorted in $O(m+\kappa\log\kappa)$ time, and the additional time to compute the distances for all of these $\kappa+1$ medians is $O(\kappa m)$. However, the computation of distances given by consecutive transpositions can be incrementalized using Lemma 15. First one has to compute the distance for the median of $\mathbb{T} \setminus M(0,\kappa)$, $d_{\mathrm{SAD}}(A+m_{0,\kappa},B)$, and then continue incrementally from $d_{\mathrm{SAD}}(A+m_{k',k''},B)$ to $d_{\mathrm{SAD}}(A+m_{k'+1,k''-1},B)$, until we reach the median of $\mathbb{T} \setminus M(\kappa,0)$, $d_{\mathrm{SAD}}(A+m_{\kappa,0},B)$ (this is where we need the medians sorted). Since the set of outliers changes when moving from one median to another, one has to add value $t_{i_{k'}} - t_{i_m}$ to S_m and value $t_{i_m} - t_{i_{k''}}$ to L_m, where S_m and L_m are the values given by Lemma 15 (here we need the outliers sorted). The time complexity of the whole algorithm is $O(m+\kappa\log\kappa)$. On an integer alphabet, sorting can be replaced by array lookup to yield $O(|\Sigma|+m+\kappa)$. □

6.3 Maximum Absolute Difference Distance

We consider now how $d^{\mathrm{t},\kappa}_{\mathrm{MAD}}$ can be computed. In case $\kappa=0$, we search for a transposition t minimizing $d_{\mathrm{MAD}}(A+t,B) = \max_{i=1}^{m} |b_i-(a_i+t)|$. In case $\kappa>0$, we are allowed to discard the k largest differences $|b_i-(a_i+t)|$.

Theorem 17 *Given two numeric strings A and B both of length m, there is an algorithm for computing distance $d^{\mathrm{t},\kappa}_{\mathrm{MAD}}(A,B)$ in $O(m+\kappa\log\kappa)$ time on both integer and general alphabets. On integer alphabets, time $O(|\Sigma|+m+\kappa)$ can also be obtained.*

PROOF. When $\kappa=0$ the distance is clearly $d^{\mathrm{t}}_{\mathrm{MAD}}(A,B) = (\max_i\{t_i\} - \min_i\{t_i\})/2$, and the transposition giving this distance is $(\max_i\{t_i\} + \min_i\{t_i\})/2$. When $\kappa>0$, consider again the sorted sequence $t_{i_1}, t_{i_2}, \ldots, t_{i_m}$ as in the proof of Theorem 14. Again the κ outliers are $M(k',k'')$ for some $k'+k''=\kappa$ in the optimal transposition. The optimal transposition is then the value $(t_{i_{m-k''}} + t_{i_{k'+1}})/2$ that minimizes $(t_{i_{m-k''}} - t_{i_{k'+1}})/2$, where $k'+k''=\kappa$. The minimum value can be computed in $O(\kappa)$ time, once the $\kappa+1$ smallest and largest t_i values are sorted. These values can be selected in $O(m)$ time and then sorted in $O(\kappa\log\kappa)$ time, or $O(|\Sigma|+\kappa)$ on integer alphabets. □

6.4 Searching

Up to now we have considered distance computation. Any algorithm to compute the distance between A and B can be trivially converted into a search

algorithm for P in T by comparing P against every text window of the form $T_{j-m+1\ldots j}$. Actually, we do not have any search algorithm better than this.

Lemma 18 *For distances $d_{\mathrm{H}}^{\mathrm{t},\delta}$, $d_{\mathrm{SAD}}^{\mathrm{t},\kappa}$, and $d_{\mathrm{MAD}}^{\mathrm{t},\kappa}$, if the distance can be evaluated in $O(f(m))$ time, then the corresponding search problem can be solved in $O(f(m)n)$ time.*

On the other hand, it is not immediate how to perform transposition invariant (δ, γ)–matching. We show how the above results can be applied to this case.

Note that one can find in $O(mn)$ time all the occurrences $\{j\}$ such that $d_{\mathrm{MAD}}^{\mathrm{t}}(P, T_{j-m+1\ldots j}) \leq \delta$, and all the occurrences $\{j'\}$ where $d_{\mathrm{SAD}}^{\mathrm{t}}(P, T_{j'-m+1\ldots j'}) \leq \gamma$. The (δ, γ)–matches are a subset of $\{j\} \cap \{j'\}$, but identity does not necessarily hold. This is because the optimal transposition can be different for $d_{\mathrm{MAD}}^{\mathrm{t}}$ and $d_{\mathrm{SAD}}^{\mathrm{t}}$.

What we need to do is to verify this set of possible occurrences $\{j\} \cap \{j'\}$. This can be done as follows. For each possible match $j'' \in \{j\} \cap \{j'\}$ one can compute limits s and l such that $d_{\mathrm{MAD}}(P+t, T_{j''-m+1\ldots j''}) \leq \delta$ for all $s \leq t \leq l$: If the distance $d = d_{\mathrm{MAD}}(P + t_{opt}, T_{j''-m+1\ldots j''})$ is given, then $s = t_{opt} - (\delta - d)$ and $l = t_{opt} + (\delta - d)$. On the other hand, note that $d_{\mathrm{SAD}}(P+t, T_{j''\ldots j''+m-1})$, as a function of t, is decreasing until t reaches the median of the transpositions, and then increasing. Thus, depending on the relative order of the median of the transpositions with respect to s and l, we only need to compute distance $d_{\mathrm{SAD}}(P + t, T_{j''-m+1\ldots j''})$ in one of them ($t = s$, $t = l$, or $t = t_{\lceil m/2 \rceil}$). This gives the minimum value for d_{SAD} in the range $[s, l]$. If this value is $\leq \gamma$, we have found a match.

One can see that using the results of Theorems 14 and 17 with $\kappa = 0$, the above procedures can be implemented so that only $O(m)$ time at each possible occurrence is needed. There are at most n occurrences to test.

Theorem 19 *Given two numeric strings P (pattern) and T (text) of lengths m and n, there is an algorithm for finding all the transposition invariant (δ, γ)–occurrences of P in T in $O(mn)$ time on both integer and general alphabets.*

7 Conclusions and Future Work

We have studied two techniques for solving transposition invariant string matching problems. The first technique, applicable to several "edit distance" measures, considered all the possible transpositions. However, since most transpositions produce sparse instances of the edit distance matrix, specialized algorithms could be used to solve these sparse instances efficiently. These kind

of algorithms already existed in the literature. We devised improved sparse dynamic programming algorithms in those cases (for example LCS and Levenshtein distance), as well as new ones when they did not exist (for example episode matching and α–limited gaps in all the distances). The problem of matching with α–limited gaps most clearly demonstrated the connection between sparse dynamic programming and range-minimum searching.

The second technique was to directly identify the optimal transposition and compute the distance in that transposition. This identification was shown to be efficiently computable for several distance measures where the i-th character of one string is compared only against the i-th character of the other.

In general, we found that including transposition invariance in the studied distances increases the time complexity only slightly, usually by a polylogarithmic factor.

To demonstrate the practicality of the developed methods, we implemented the transposition invariant LCS algorithm. This implementation is now included in the C-Brahms music retrieval engine [6].

An interesting remaining question is whether the log factors could be avoided to achieve $O(mn)$ for transposition invariant edit distances. For episode matching we achieved the $O(mn)$ bound, except that the preprocessing can (in very uncommon situations on general alphabets) take $O(mn \log m + n \log n)$ time. Independently, it would be nice to reduce preprocessing time to $O(mn)$, so that it can never affect the real dynamic programming complexities. The bottleneck is in sorting mn values of the form $b_j - a_i$, once the $\{a_i\}$ and the $\{b_j\}$ sequences, of length n and m, have been sorted. We could do it in $O(mn \log \min(m, n))$ time, but maybe it can be done better. Also, the space needed to arrange the transpositions for distance computation is $O(mn)$. We have been able to reduce all the other space complexities to small polynomials in m, so it would be interesting to do the same with the transpositions. We tried, with no result, to mix generation and processing of the cells. The problem is that there may be too many active transpositions at any time.

Also, we are confident that the search times for the easier measures that we studied can be improved at least in the average case. For the edit distance measures, algorithms that depend on the minimum (transposition invariant) distance can be derived. For example, an algorithm that processes only diagonal areas of the dynamic programming matrix [36] can be generalized to give time bounds like $O(|\mathbb{T}|dn)$, where $\mathbb{T}$ is the set of transpositions and $d = d_*^{\mathrm{t}}(A, B)$. This can be combined with the sparse evaluation to get an algorithm that is fast both in practice and in the worst case, $O(dn \log \log m)$. The challenge is to derive a similar bound for the search problem.

Finally, a more ambitious goal is to handle more general distance functions,

such as edit distances with substitution costs of the form $|b_j - a_i|$. Other related models are discussed in [33].

Acknowledgments

We thank the anonymous referees for their useful suggestions to improve this work.

References

[1] K. Abrahamson. Generalized string matching. *SIAM J. Computing*, 16(6):1039–1051, 1987.

[2] A. Apostolico and C. Guerra. The longest common subsequence problems revisited. *Algorithmica* 2:315–336, 1987.

[3] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2nd rev. ed. 2000.

[4] M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bounds for selection. *J. Computer and System Sciences*, 7:448–461, 1972.

[5] 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.

[6] C-BRAHMS. http://www.cs.helsinki.fi/group/cbrahms/demoengine/.

[7] E. Cambouropoulos, M. Crochemore, C.S. Iliopoulos, L. Mouchard, and Yoan J. Pinzón. Algorithms for computing approximate repetitions in musical sequences. In *Proc. 10th Australian Workshop on Combinatorial Algorithms, AWOCA'99*, R. Raman and J. Simpson, eds., Curtin University of Technology, Perth, Western Australia, pp. 129–144, 1999.

[8] R. Cole and R. Hariharan. Tree pattern matching and subset matching in randomized $O(n \log^3 m)$ time. In *Proc. 29th Annual Symposium on the Theory of Computing (STOC'97)*, pp. 66–75, 1997.

[9] R. Cole, R. Hariharan, and P. Indyk. Tree pattern matching and subset matching in deterministic $O(n \log^3 m)$ time. In *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99)*, pp. 245–254, 1999.

[10] R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proc. 34th Annual Symposium on the Theory of Computing (STOC'02)*, pp. 596–601, 2002.

[11] T. Crawford, C.S. Iliopoulos, and R. Raman. String matching techniques for musical similarity and melodic recognition. *Computing in Musicology* 11:71–100, 1998.

[12] M. Crochemore, C.S. Iliopoulos, T. Lecroq, and Y.J. Pinzón. Approximate string matching in musical sequences. In *Proc. Prague Stringology Club (PSC 2001)*, M. Baliik and M. Simanek, eds, Czech Technical University of Prague, pp. 26–36, DC-2001-06, 2001.

[13] 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, 2002.

[14] M. Crochemore, G. Landau, and M. Ziv-Ukelson. A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. In *Proc. 13th Symposium on Discrete Algorithms (SODA'02)*, pp. 679–688. ACM-SIAM, 2002.

[15] M. Crochemore, C.S. Iliopoulos, T. Lecroq, W. Plandowski, and W. Rytter. Three Heuristics for δ–Matching: δ–BM Algorithms. In *Proc. 13th Annual Symposium on Combinatorial Pattern Matching (CPM'02)*, Springer-Verlag LNCS 2373, pp. 178–189, 2002.

[16] M. Crochemore, C. Iliopoulos, G. Navarro, and Y. Pinzón. A bit-parallel suffix automaton approach for (δ, γ)–matching in music retrieval. In *Proc. 10th International Symposium on String Processing and Information Retrieval (SPIRE'03)*, LNCS, 2003. To appear.

[17] M. Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, 1984.

[18] G. Das, R. Fleischer, L. Gasieniec, D. Gunopulos, and J. Kärkkäinen. Episode matching. In *Proc. 8th Symposium on Combinatorial Pattern Matching (CPM'97)*, LNCS 1264, Springer, pp. 12–27, 1997.

[19] M. J. Dovey. A technique for "regular expression" style searching in polyphonic music. In *Proc. 2nd Annual International Symposium on Music Information Retrieval (ISMIR 2001)*, pp. 179–185, 2001.

[20] 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, 1992.

[21] K. Fredriksson. *Rotation Invariant Template Matching.* PhD Thesis, A-2001-3, Department of Computer Science, University of Helsinki, 139 pages, 2001.

[22] K. Fredriksson, V. Mäkinen, and G. Navarro. *Rotation and Lighting Invariant Template Matching.* Manuscript, June, 2003.

[23] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. *Proc. 16th ACM Symposium on Theory of Computing (STOC'84)*, pp. 135–143, 1984.

[24] Z. Galil and K. Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science* 92:49–76, 1992.

[25] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[26] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, May 1977.

[27] K. Lemström and V. Mäkinen. On Minimizing Pattern Splitting in Multi-track String Matching. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM'03)*, Springer-Verlag LNCS 2676, pp. 237–253, 2003.

[28] K. Lemström and J. Tarhio. Searching monophonic patterns within polyphonic sources. In *Proc. Content-Based Multimedia Information Access (RIAO 2000)*, pp. 1261–1279 (vol 2), Paris, France, 2000.

[29] K. Lemström and E. Ukkonen. Including interval encoding into edit distance based music comparison and retrieval. In *Proc. Symposium on Creative & Cultural Aspects and Applications of AI & Cognitive Science (AISB 2000)*, pp. 53–60, Birmingham, United Kingdom, 2000.

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

[31] 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.

[32] S. Muthukrishnan. New results and open problems related to non-standard stringology. In *Proc. 6th Annual Symposium on Combinatorial Pattern Matching (CPM'95)*, LNCS 937, pp. 298–317, 1995.

[33] V. Mäkinen. *Parameterized Approximate String Matching and Local-Similarity-Based Point-Pattern Matching.* PhD thesis manuscript, Report A-2003-6, Department of Computer Science, University of Helsinki, August 2003. To appear.

[34] V. Mäkinen, G. Navarro and E. Ukkonen. Algorithms for Transposition Invariant String Matching (Extended Abstract). In *Proc. 20th International Symposium on Theoretical Aspects of Computer Science (STACS 2003)*, Springer-Verlag LNCS 2607, pp. 191–202, 2003.

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

[36] E. Ukkonen. Algorithms for approximate string matching. *Information and Control* 64(1–3):100–118, 1985.

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

[38] 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.

[39] R. Wagner and M. Fisher. The string-to-string correction problem. *J. of the ACM* 21(1):168–173, 1974.

[40] W. J. Wilbur and D. J. Lipman. Rapid similarity searches of nucleic acid and protein data banks. In *Proc. Nat. Acad. Sci.*, USA, 80:726–730, 1983.

[41] W. J. Wilbur and D. J. Lipman. The context-dependent comparison of biological sequence. *SIAM J. Appl. Math.* 44(3):557-567, 1984.