Algorithms for 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 and B over an alphabet $\Sigma \subseteq \mathbb{U}$, where $\mathbb{U}$ is some numerical universe closed under addition and subtraction, and a distance function $d(A, B)$ that gives the score of the best (partial) matching of A and B, the *transposition invariant distance* is $\min_{t \in \mathbb{U}}\{d(A+t, B)\}$, where $A+t = (a_1+t)(a_2+t)\ldots(a_m+t)$. We study the problem of computing the transposition invariant distance for various distance (and similarity) functions d, including *Hamming distance*, *longest common subsequence (LCS)*, *edit 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.

Key words: edit distance, music sequence comparison, transposition invariance, sparse dynamic programming, orthant searching, range searching with activation, minimum searching over two dimensional ranges

---

1 Introduction

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

Interest in transposition invariant string matching problems has recently arisen in the field of music information retrieval (MIR) [11,23,24]. 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 [7], image comparison [18], etc. (see Section 3).

In this paper, we study how transposition invariance can be embedded in evaluating some of the classical distance measures for strings. We focus on measures that have been used in practice. 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 which one can develop practical improvements such as good average cases, bit-parallel computation,

[⋆] A short version of this paper is submitted to a conference.
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.

and so on.

Our principal result is that most of the distance measures studied allow including transposition without a significant increase in the asymptotic running times. The summary of our results is given in Section 4.

2 Definitions

Let Σ be a finite numerical alphabet, which is a subset of some universe $\mathbb{U}$ that is closed under addition and subtraction ($\mathbb{U}$ is either $\mathbb{Z}$ or $\mathbb{R}$ in the sequel, and Σ is called either *integer* or *real* alphabet, respectively). Let $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$ be two *strings* over Σ^*, i.e. $a_i, b_j \in \Sigma$ 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. String A' is a *substring* of A if $A' = A_{i \ldots j} = a_i \ldots a_j$ for some $1 \leq i \leq j \leq m$. String A'' is a *subsequence* of A, denoted by $A'' \sqsubseteq A$, if $A'' = a_{i_1} a_{i_2} \ldots a_{i_{|A''|}}$ for some indexes $1 \leq i_1 < i_2 < \cdots < i_{|A''|} \leq m$.

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

The following measures can also be defined when $m \neq n$. The length of the *longest common subsequence (LCS)* of A and B is $\mathrm{lcs}(A, B) = \max\{|S| \mid S \sqsubseteq A, S \sqsubseteq B\}$. The *edit distance* [25,34,29] between A and B is the minimum

number of edit operations that are needed to convert A into B. Particularly, in the unit cost *Levenshtein distance* d_{L} the set of edit operations consists of character insertions, deletions, and substitutions. If the substitution operation is forbidden, we get a distance d_{ID}, which is actually a dual problem of evaluating the LCS; it is easy to see that $d_{\mathrm{ID}}(A,B) = m + n - 2 \cdot \mathrm{lcs}(A,B)$. For convenience, we will mainly use the minimization problem d_{ID} (not lcs) in the sequel. If only deletion for characters of B are allowed, we get a distance d_{D}.

String A is a *transposed copy* of B (denoted by $A =^t B$) if $B = (a_1 + t)(a_2 + t) \cdots (a_m + t) = A + t$ for some $t \in \mathbb{U}$. Definitions for a transposed substring and a transposed subsequence can be stated similarly. The transposition invariant versions of the above distance measures d_* where $* \in \{\mathrm{H}, \mathrm{MAD}, \mathrm{SAD}, \mathrm{L}, \mathrm{ID}, \mathrm{D}\}$ can 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 ($d_{\mathrm{H}}^{\mathrm{t}}, d_{\mathrm{L}}^{\mathrm{t}}, d_{\mathrm{ID}}^{\mathrm{t}}, d_{\mathrm{D}}^{\mathrm{t}}$), or approximate match between all characters ($d_{\mathrm{MAD}}^{\mathrm{t}}, d_{\mathrm{SAD}}^{\mathrm{t}}$). To relax these conditions, we introduce a constant $\delta > 0$. We write $a =^\delta b$ when $|a - b| \le \delta$, $a, b \in \Sigma$. By replacing the equalities $a = b$ with $a =^\delta b$, we get more error-tolerant versions of the distance measures: $d_{\mathrm{H}}^{\mathrm{t},\delta}, d_{\mathrm{L}}^{\mathrm{t},\delta}$, $d_{\mathrm{ID}}^{\mathrm{t},\delta}$, and $d_{\mathrm{D}}^{\mathrm{t},\delta}$. Similarly, by introducing another constant $\kappa > 0$, we can define distances $d_{\mathrm{MAD}}^{\mathrm{t},\kappa}, d_{\mathrm{SAD}}^{\mathrm{t},\kappa}$ 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 two matches is limited by a constant $\alpha > 0$, i.e. if $(a_{i'}, b_{j'})$ and (a_i, b_j) are matches, then $|i - i' - 1| \le \alpha$ and $|j - j' - 1| \le \alpha$. We get distances $d_{\mathrm{L}}^{\mathrm{t},\delta,\alpha}, d_{\mathrm{ID}}^{\mathrm{t},\delta,\alpha}$, and $d_{\mathrm{D}}^{\mathrm{t},\delta,\alpha}$.

The *approximate string matching problem*, based on the above distance func-

tions, 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* [27,15]. It can also be stated as follows: Find the shortest substring of the text that contains the pattern as a subsequence.

Our complexity results are different depending on the form of the alphabet Σ. We will distinguish two cases. An *integer* alphabet is any alphabet $\Sigma \subset \mathbb{Z}$. For integer alphabets, $|\Sigma|$ will denote $\max(\Sigma) - \min(\Sigma) + 1$. A *real* alphabet will be any other $\Sigma \subseteq \mathbb{R}$ and we will omit any reference to $|\Sigma|$. 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 real alphabets, $|\Sigma_A| \leq m$ will denote the cardinality of the set Σ_A.

3 Related Work and Motivation

The first thing to notice is 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 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 [24]. 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 nick name for the measure: “Longest Common Hidden Melody”. As the alphabet of pitches is some limited integer alphabet $\Sigma \subset \mathbb{Z}$, the transpositions that have to be considered are $\mathbb{T} = \{b - a \mid a, b \in \Sigma\}$. This gives a brute force algorithm for computing the length of the LCTS [24]: Compute $\mathrm{lcs}(A + t, B)$ using $O(mn)$ dynamic programming for each $t \in \mathbb{T}$. The runtime of this algorithm is $O(|\Sigma|mn)$, where typically $|\Sigma| = 256$. In the general case, where Σ could be unlimited, one could instead use the set of transpositions $\mathbb{T}' = \{b - a \mid a \in A, b \in B\}$. This is because some characters must match in any meaningful transposition. The size of $\mathbb{T}'$ could be mn, which gives $O(m^2n^2)$ worst case time for real alphabets. Thus it is both of 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 when considering distance functions with the parameter δ; 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 [27,15]. 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 least decorative notes. This is a good objective, since an occurrence with 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 [16,13], where the number of these additional notes between two matches 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 matches is limited by an integer $\alpha > 0$.

Even simpler measures have been proposed for the search problem; these include variants of d_{H}^{δ}, d_{MAD} and d_{SAD} [8,12]. In the "(δ, γ)–matching problem", 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 $d_{\mathrm{SAD}}(P, T_{j_r-m+1\ldots j_r}) \leq \gamma$. Algorithms for exact string matching can be generalized to this special case, and bit-parallel algorithms can be applied [8,26].

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 [28], and it has long been an open question to improve the quadratic bound. We will not improve this bound 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 [10,9]. Novel (but impractical) algorithms have been developed for this problem [10,9]. To allow transposition invariance, one could simulate these algorithms with each possible transposition. The time complexity would then be $O(|\Sigma| s \log^3 m)$, where s is the sum of the subset sizes. A practical approach has been taken by Lemström and Tarhio [23], who develop a fast filter for the problem with transposition invariance; they also give a simple verification algorithm that has running time $O(|\Sigma| n + sm)$. To finish the MIR part, we note that the problems that lead to dynamic programming (like LCS, edit distance, episode matching) can easily be adapted to the case in which the text consists of subsets.

Other applications for transposition invariance can be found, e.g., 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, e.g., by Fredriksson in his study of rotation invariant template matching [18]. 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.

In time series comparison, many of the measures can be used. In fact, the episode matching was first introduced in this context [27]. Recently, a closely related problem to the transposition invariant LCS was studied by Bollobás et. al [7]. They studied a slightly more difficult problem where not only transposition (translation), but also scaling was allowed. They also allowed a tolerance between matched values, byt did not consider transpositions alone.

4 Summary of Results

Our results are two-fold. For evaluating the easier distance measures ($d_{\mathrm{H}}^{\mathrm{t},\delta}$, $d_{\mathrm{MAD}}^{\mathrm{t},\kappa}$, $d_{\mathrm{SAD}}^{\mathrm{t},\kappa}$) we achieve almost the same bounds that are known without the 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 ($d_{\mathrm{L}}^{\mathrm{t},\delta,\alpha}$, $d_{\mathrm{ID}}^{\mathrm{t},\delta,\alpha}$, 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 give much better worst case bounds. Then we show the connection between the resulting sparse dynamic programming problems and dynamic range minimum queries. We obtain simple yet efficient algorithms for the distances $d_{\mathrm{L}}^{\mathrm{t},\delta,\alpha}$, $d_{\mathrm{ID}}^{\mathrm{t},\delta,\alpha}$, and $d_{\mathrm{D}}^{\mathrm{t},\delta,\alpha}$.

For LCS (and thus for d_{ID}) there already exists Hunt-Szymanski [22] type (sparse dynamic programming) algorithms whose time complexities depend on the number r of matching pairs between the compared strings. The complexity of the Hunt-Szymanski algorithm is $O((r+n)\log n)$. 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 [2,17] yield $O(mn \log\log n)$ time. We improve this to $O(mn \log\log m)$ by giving a new sparse dynamic algorithm for LCS. This algorithm can also be generalized to the case where gaps are limited by a constant α, giving $O(mn \log n)$ for evaluating $d_{\mathrm{ID}}^{\mathrm{t},\alpha}(A, B)$.

Eppstein et. al. [17] have proposed sparse dynamic programming algorithms for more complex distance computations such as Wilbur-Lipman fragment alignment problem [35,36]. Also the unit cost Levenshtein distance can be solved using these techniques [20]. 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 semi-static range minimum queries. This gives us $O(mn \log^2 n \log\log m)$ for evaluating $d_{\mathrm{L}}^{\mathrm{t},\alpha}(A, B)$.

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

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 variants of these problems.

Table 1

Upper bounds for string matching without transposition invariance. We omit bounds that depend on the treshold k in the search problems. For $d_{\mathrm{ID}}^{\delta,\alpha}$ and $d_{\mathrm{L}}^{\delta,\alpha}$ we could not find existing algorithms; naive dynamic programming gives $O(\alpha^2 mn)$ for both, and our sparse dynamic programming algorithms give $O(mn \log n)$ and $O(mn \log^2 n \log\log m)$, respectively (bounds are simplified by assigning $r = mn$).

distance	distance evaluation	searching
exact	$O(m)$	$O(m+n)$
d_{H}	$O(m)$	$O(n\sqrt{m \log m})$ [1]
d_{H}^{δ}	$O(m)$	$O(mn)$
$d_{\mathrm{SAD}}^{\kappa}$	$O(m)$	$O(mn)$
$d_{\mathrm{MAD}}^{\kappa}$	$O(m)$	$O(mn)$
(δ,γ)–matching	$O(m)$	$O(mn)$
$d_{\mathrm{ID}}, d_{\mathrm{L}}$	$O(mn/\log m)$	$O(mn/\log m)$ [14]
d_{D}	$O(mn/\log m)$	$O(mn/\log m)$ [15]
$d_{\mathrm{D}}^{\delta,\alpha}$	$O(mn)$	$O(mn)$ [13]

5 Computation of $d_H^{\mathrm{t},\delta}$,$d_{\mathrm{SAD}}^{\mathrm{t},\kappa}$, and $d_{\mathrm{MAD}}^{\mathrm{t},\kappa}$

For this section, let $\mathbb{T} = \{t_i = b_i - a_i \mid 1 \leq i \leq m\} = \{t_i\}$ be the set of transpositions that make some characters in A and B match. Note that the

Table 2

Upper bounds for transposition invariant string matching. In integer alphabet, $\kappa \log \kappa$ can be replaced by $|\Sigma| + \kappa$, and $m \log m$ by $|\Sigma| + m$ for $d_{\mathrm{H}}^{\mathrm{t},\delta}$. Also, use $\delta + 1$ instead of δ and $\log(2 + x)$ instead of $\log x$ to get correct bounds for small δ and κ values. We have not added, for clarity, the size of the output in the (thresholded) search complexity, nor the preprocessing time in Lemma 10 for the edit distance measures. The bounds on these distances are valid in real alphabets provided we replace δ by δ/μ, where μ is the minimum distance between two characters in A or in B.

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	$O(m)$	$O(mn)$
$d_{\mathrm{ID}}^{\mathrm{t},\delta}$	$O(\delta mn \log \log m)$	$O(\delta mn \log \log m)$
$d_{\mathrm{ID}}^{\mathrm{t},\delta,\alpha}$	$O(\delta mn \log n)$	$O(\delta mn \log m)$
$d_{\mathrm{L}}^{\mathrm{t},\delta}$	$O(\delta mn \log \log n)$	$O(\delta mn \log \log m)$
$d_{\mathrm{L}}^{\mathrm{t},\delta,\alpha}$	$O(\delta mn \log^2 n \log \log m)$	$O(\delta mn \log^2 m \log \log m)$
$d_{\mathrm{D}}^{\mathrm{t},\delta,\alpha}$	$O(\delta mn)$	$O(\delta mn)$

optimal transposition does not need, in principle, to be included in $\mathbb{T}$, but we will show that this is the case for d_H^{t} and $d_{SAD}^{\mathrm{t},\kappa}$. Note also that $|\mathbb{T}| = O(|\Sigma|)$ in integer alphabet and $|\mathbb{T}| = O(m)$ in any case.

We consider first the computation of transposition invariant Hamming distance $d_H^{\mathrm{t},\delta}$. Let $A = a_1 \ldots a_m$ and $B = b_1 \ldots b_m$, where $a_i, b_i \in \Sigma$, $1 \leq i \leq m$.

Theorem 1 *One can compute $d_H^{\mathrm{t},\delta}(A, B)$ in $O(|\Sigma| + m)$ time with integer alphabet, or in $O(m \log m)$ time in real alphabet.*

PROOF. It is clear that the Hamming distance is minimized for the transposition in $\mathbb{T}$ that makes the maximal number of the characters match. What follows is a simple voting scheme, where the most voted t_i wins. Additionally, it is extended to match δ positions back and forth from each t_i. Let $\cdot$ denote a don't care element, and $p.x$ ($p.y$) denote the first (second) element of a pair $p = (\cdot, \cdot)$. 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

$$(x', y') <^h (x, y) : x' < x \text{ or } (x' = x \text{ and } y' < y),$$

where "open"<"close". Initialize variable $count = 0$. Do for $i = 1$ to $|I|$ if $I(i) = (\cdot, \text{"open"})$ 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 $[I(i).x, I(i+1).x]$, for any i where $maxcount$ is reached. The complexity of the algorithm is $O(m \log m)$. Sorting can be replaced by array indexing when Σ is an integer alphabet, which gives the bound $O(|\Sigma| + m)$ for that case. □

We shall first look at the basic case where $\kappa = 0$.

Theorem 2 *One can compute $d^{\mathrm{t}}_{\mathrm{SAD}}(A, B)$ in $O(m)$ time with both integer and real alphabet.*

PROOF. Sorting $\mathbb{T}$ in ascending order gives a sequence $t_{i_1}, t_{i_2}, \ldots, t_{i_m}$. Let t_{opt} be the optimal transposition, where $t_{i_{j-1}} \leq t_{opt} \leq t_{i_j}$ for some $1 < j \leq m$. The cases $t_{opt} \leq t_{i_1}$ or $t_{opt} > t_{i_m}$ can be discarded as we will see. We can rewrite $d_{\mathrm{SAD}}(A + t_{opt}, B)$ as follows:

$$d_{\mathrm{SAD}}(A + t_{opt}, B) = \sum_{j'=1}^{j-1} (t_{opt} - t_{i_{j'}}) + \sum_{j'=j}^{m} (t_{i_{j'}} - t_{opt}). \tag{1}$$

We have two cases (i) $j - 1 \leq m - j$, and (ii) $j - 1 > m - j$. In case (i) we can rearrange terms in (1) and get

$$d_{\mathrm{SAD}}(A + t_{opt}, B) = \sum_{j'=1}^{j-1} (t_{i_{m-j'+1}} - t_{i_{j'}}) + \sum_{j'=j}^{m-j+1} (t_{i_{j'}} - t_{opt}). \tag{2}$$

From equation (2) one can see that as long as there are terms in the second summation, one can increase t_{opt} so that the overall cost will decrease. This remains true even when we move from $t_{i_{j-1}} \leq t_{opt} \leq t_{i_j}$ to $t_{i_j} \leq t_{opt} \leq t_{i_{j+1}}$. If m is odd the value of t_{opt} can be increased until $t_{i_{(m+1)/2-1}} \leq t_{opt} \leq t_{i_{(m+1)/2}}$. Obviously $t_{opt} = t_{i_{(m+1)/2}}$ in that case. If m is even the value of t_{opt} can be increased until there are two terms left in the summation. Then the optimal transposition t_{opt} is any value between and including $t_{i_{m/2}}$ and $t_{i_{m/2+1}}$; they all produce the same cost. Case (ii) gives the same result, so we can conclude that it is enough to compute the distance with $t = t_{i_{\lfloor m/2 \rfloor + 1}}$. Sorting is not needed since $t_{i_{\lfloor m/2 \rfloor + 1}}$ can be found with a linear time median finding algorithm. $\square$

To get a fast algorithm for $d^{\mathrm{t},\kappa}_{\mathrm{SAD}}$ when $\kappa > 0$ mismatches are allowed, 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 3 *Given values j, 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 $L_j = \sum_{j'=j+1}^{m} t_{i_{j'}} - t_{i_j}$, 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

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

Similarly L_{j+1} can be written as

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

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

Theorem 4 *One can compute $d^{\mathrm{t},\kappa}_{\mathrm{SAD}}(A, B)$ in $O(m + \kappa \log \kappa)$ time with both integer and real alphabet.*

PROOF. Consider the sorted sequence $t_{i_1}, t_{i_2}, \ldots, t_{i_m}$ as in the proof of Theorem 2. Clearly the candidates for the κ outliers are $M(k', k'') = \{t_{i_1}, \ldots, t_{i_{k'}}, t_{i_{m-k''}}, \ldots t_{i_m}\}$ for some $k' + k'' = \kappa$. The naive algorithm is then to

compute the distance in all these $\kappa+1$ cases: Compute medians of $\mathbb{T}\setminus M(k',k'')$ and choose the minimum distance induced by these medians. These $\kappa+1$ medians can be found by first taking the median of $\mathbb{T}\setminus M(0,\kappa)$ and of $\mathbb{T}\setminus M(\kappa,0)$, and then passing over the set collecting and sorting all the values in between, as these are the medians of $\mathbb{T}\setminus M(k',k'')$. The $\kappa+1$ medians can thus be taken 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 3. First one has to compute the distance for the median of $\mathbb{T}\setminus M(0,\kappa)$, and then continue incrementally until we reach the median of $\mathbb{T}\setminus M(\kappa,0)$ (this is where we need the medians sorted). Since the set of mismatches 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 3. The time complexity of this algorithm is $O(m+\kappa\log\kappa)$. □

5.3 Maximum Absolute Difference Distance

We consider now how $d^{\mathrm{t},\kappa}_{\mathrm{MAD}}$ can be computed.

Theorem 5 *One can compute $d^{\mathrm{t},\kappa}_{\mathrm{MAD}}(A,B)$ in $O(m+\kappa\log\kappa)$ time with both integer and real alphabet.*

PROOF. When $\kappa=0$ the optimal 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 2. Again the κ outliers are $M(k',k'')$ for some $k'+k''=\kappa$ in the optimal transposition. For each choice, the distance

can be computed in $O(1)$ time (it is $(t_{i_{m-k''-1}} - t_{i_{k'+1}})/2$). The $O(\kappa)$ medians can be found in linear time, but they must be sorted in order to construct sets induced by mismatch sets incrementally from $M(0,\kappa)$ to $M(\kappa,0)$. Thus the complexity becomes $O(m + \kappa \log \kappa)$. □

Remark 6 *In integer alphabet, terms $\kappa \log \kappa$ in $d^{t,\kappa}_{SAD}$ and $d^{t,\kappa}_{MAD}$ could be replaced by $\kappa + |\Sigma|$, since the sorting could then be replaced by array indexing.*

5.4 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 a search algorithm better than this.

Lemma 7 *For distances $d^{t,\delta}_{H}$, $d^{t,\kappa}_{SAD}$, and $d^{t,\kappa}_{MAD}$, 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.

5.4.1 Transposition invariant (δ,γ)–matching.

Note that one can find in $O(mn)$ time all the occurrences $\{j\}$ such that $d^{t}_{MAD}(P, T_{j-m+1\ldots j}) \leq \delta$, and all the occurrences $\{j'\}$ where $d^{t}_{SAD}(P, T_{j'-m+1\ldots j'}) \leq \gamma$. 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 $d^{\mathrm{t}}_{\mathrm{MAD}}$ and $d^{\mathrm{t}}_{\mathrm{SAD}}$.

What we need to do is to verify this set of possible matches $\{j\} \cap \{j'\}$. 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 the function $d_{\mathrm{SAD}}(P+t, T_{j''\ldots j''+m-1})$, 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 the SAD distance in one of them ($t = s$, $t = l$, or $t = t_{\lfloor m/2 \rfloor + 1}$). This gives the minimum value for SAD in the range $[s, t]$. If this value is $\leq \gamma$, we have found a match.

One can see that using the results of Theorems 2 and 5 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.

Corollary 8 *One can find all the transposition invariant (δ, γ)–occurrences in $O(mn)$ time with both integer and real alphabet.*

6 Computation of $d^{\mathrm{t},\delta}_{\mathrm{ID}}$, $d^{\mathrm{t},\delta}_{\mathrm{L}}$, and $d^{\mathrm{t},\delta}_{\mathrm{D}}$

Let us first review how the edit distances can be computed using dynamic programming [25,34,29]. 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}), $0 \leq i \leq m$, $0 \leq j \leq n$, using the recurrence

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

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

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 the distance $d_{\mathrm{L}}(A, B)$. Now, the recurrence is

$$d_{i,j} = \min((d_{i-1,j-1} + \textbf{if } a_i = b_j \textbf{ then } 0 \textbf{ else } +1), d_{i-1,j} + 1, d_{i,j-1} + 1), \quad (4)$$

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

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

$$d_{i,j} = \textbf{if } a_i = b_j \textbf{ then } d_{i-1,j-1} \textbf{ else } d_{i,j-1} + 1, \quad (5)$$

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

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

For episode matching there is an alternative (and more useful) recurrence [15]

$$d_{i,j} = \textbf{if } a_i = b_j \textbf{ then } d_{i-1,j-1} \textbf{ else } d_{i,j-1}, \quad (6)$$

with initialization $d_{i,0} = \infty$ for $0 \leq i \leq m$ and $d_{0,j} = j$ for $1 \leq j \leq n$. The length of the best episode match is then $\min_{1 \leq j \leq n}\{|j - d_{m,j}|\}$.

To solve our transposition invariant problems, we could try to prove that only some transpositions need to be checked, as is the case with the problems in

the previous section. This does not seem to be possible with the more flexible measures of similarity studied here. Therefore we choose a different approach: 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 in one transposition.

Let M be the set of matching characters between strings A and B, i.e. $M = M(A,B) = \{(i,j) \mid a_i = b_j, 1 \leq i \leq m, 1 \leq j \leq n\}$. Let $r = r(A,B) = |M(A,B)|$. Let us redefine $\mathbb{T}$ in this section 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 integer alphabet this takes $O(|\Sigma|mn)$ time, and $O(|\Sigma_A||\Sigma_B|mn)$ time in real alphabet. This kind of procedure can be significantly speeded up if the basic dynamic programming algorithms are replaced by suitable "sparse dynamic programming" algorithms.

Lemma 9 *If an algorithm computes a distance $d(A,B)$ in $O(g(r(A,B))f(m,n))$ time, where g is a concave function, then the transposition invariant distance $d^{\mathrm{t}}(A,B) = \min_{t \in \mathbb{T}} d(A+t, B)$ can be computed in $O(g(mn)f(m,n))$ time.*

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

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

The rest of the section devotes to developing algorithms that depend on r.

6.1 Preprocessing

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

We must be careful in constructing these match sets for all transpositions so that the overall preprocessing time will 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 (such as column-by-column) 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 in integer alphabet and $O(mn \log(|\Sigma_A||\Sigma_B|)) = O(mn \log n)$ in real alphabet (by using a balanced tree of existing transpositions).

Also, one should pay attention to the space usage: The sum of all the sizes $|M_t|$ is $O(mn)$ space, which can be too much especially in the search problem. This can be reduced to $O(m^2)$ in the search problems for $d^{\mathrm{t}}_{\mathrm{ID}}$ and $d^{\mathrm{t}}_{\mathrm{L}}$; values in

column j cannot affect the values at column $j+2m$, and thus one can partition $B = T$ into substrings of length $3m$ so that the consecutive substrings overlap by m characters. Then one can run the algorithms over all pairs (A, B'), where B' is a substring described above. To achieve $O(m^2)$ space (with algorithms that depend on r), we need to be able to produce match sets for each (A, B') separately. For $d_{\mathrm{D}}^{\mathrm{t}}$ this trick does not apply, but as we will see, only the matches in the current column are needed.

Lemma 10 *The* match sets $M_t = \{(i, j) \mid a_i + t = b_j\}$, *each sorted in column-by-column order, for all transpositions* $t \in \mathbb{T}$, *can be constructed with the following complexities. On integer alphabet,* $O(|\Sigma| + mn)$. *On real alphabet* , $O(m \log |\Sigma_A| + n \log |\Sigma_B| + |\Sigma_A||\Sigma_B| \log(|\Sigma_A||\Sigma_B|) + mn)$. *Both bounds can be achieved using* $O(mn)$ *space. If* B *can be partitioned into* $O(n/p)$ *overlapping substrings of length* $O(p)$ *or a window of length* p *can be slid over* B, *we get* $O(mp)$ *space on integer alphabet and* $O(mp + |\Sigma_A||\Sigma_B|)$ *on real alphabet. The latter can be reduced to* $O(mp)$ *at a time cost of* $O(n|\Sigma_A| \log(|\Sigma_A||\Sigma_B|) + mn)$.

For the versions that relax the matching condition using parameter δ, *we get* $O(|\Sigma| + \delta mn)$ *on integer alphabet and* $O(m \log |\Sigma_A| + n \log |\Sigma_B| + |\Sigma_A||\Sigma_B| \log(|\Sigma_A||\Sigma_B|) + mn(\delta/\mu) \log(\delta/\mu))$ *on real alphabet, where* $\mu = \min\{|a_i - a_j| \mid 1 \leq i < j \leq m,\ a_i \neq a_j\} \cup \{|b_i - b_j| \mid 1 \leq i < j \leq n,\ b_i \neq b_j\}$. *For real alphabet and* $O(m^2)$ *space, the cost is* $O(n|\Sigma_A| \log(|\Sigma_A||\Sigma_B|) + mn(\delta/\mu) \log(\delta/\mu))$.

PROOF. In the integer case we can proceed naively to obtain $O(|\Sigma| + mn)$ time using array indexing to get the transposition 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 $O(|\Sigma| + \delta mn)$ time. If B is processed by blocks, the previous

block can be used to empty the lists created when processing it in $O(\delta m)$ instead of the $O(|\Sigma|)$ time that would be necessary for a full reinitialization, hence retaining the $O(|\Sigma| + \delta mn)$ complexity for this case too.

The case of real alphabets with $O(mn)$ memory is solved as follows. Let us first consider the case $\delta = 0$. Create a balanced 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 $O(m \log |\Sigma_A| + n \log |\Sigma_B|)$. In which follows we will speak indistinctly of characters of Σ_A (Σ_B) and nodes of $\mathcal{T}_A$ ($\mathcal{T}_B$). For each node a in $\mathcal{T}_A$ and b in $\mathcal{T}_B$, initialize $M_{b-a} = \emptyset$ and insert it into a tree of transpositions, $\mathcal{T}_{\mathbb{T}}$. At the same time, create a simple list $\mathcal{P}_b$ for each node b in $\mathcal{T}_B$ containing, for each node a of $\mathcal{T}_A$, a pointer to the node a of $\mathcal{T}_A$ and to the node $b - a$ in $\mathcal{T}_{\mathbb{T}}$. This takes $O(|\Sigma_A||\Sigma_B| \log(|\Sigma_A||\Sigma_B|))$ time, since $|\mathbb{T}| \leq |\Sigma_A||\Sigma_B|$. Finally, traverse all the lists of positions $\mathcal{L}_b$ of $\mathcal{T}_B$ in synchronization, getting consecutive positions j in B (this is done, e.g., by putting all the tree nodes b in a heap sorted by the first position in the list $\mathcal{L}_b$, extracting the smallest, and reinserting it with the next position in the list). For each extracted position j of B corresponding to a node b in $\mathcal{T}_B$, traverse its list of pairs $\mathcal{P}_b = \{(i,t) \in (\mathcal{T}_A \text{ node}, \mathcal{T}_{\mathbb{T}} \text{ node})\}$. For each such list element, add (i,j) to set M_t in $\mathcal{T}_{\mathbb{T}}$. This takes overall $O(n \log |\Sigma_B| + mn)$ time.

Let us consider now how we can modify the above algorithm for the case where we have to process B by blocks of length $O(p)$, so that we use less space. The point is to show that we can move from one block to the next fast, removing the positions of B that we leave behind and adding the new positions we reach. In order to remove the smallest position j from the structure described above, we have to locate $b = b_j$ in $\mathcal{T}_B$, remove the first element of $\mathcal{L}_b$, and then

traverse $\mathcal{P}_b$ removing the $|\Sigma_A|$ positions $(*, j)$ from the sets M_{b-a_*} in $\mathcal{T}_{\mathbb{T}}$. Since we have direct pointers for the latter operation, all this can be carried out in time $O(\log|\Sigma_B| + |\Sigma_A|)$. Since each character of B is removed only once, we get overall remotion time $O(n \log|\Sigma_B| + n|\Sigma_A|) = O(n \log|\Sigma_B| + mn)$, which does not affect our complexities. Insertion of a new character $b = b_j$ is similar: we locate b in $\mathcal{T}_B$, add j at the end of $\mathcal{L}_b$ and add $(*, j)$ to the sets M_{b-a_*} in $\mathcal{T}_{\mathbb{T}}$. The complexity is the same.

However, there is a detail that must be considered. In order to have $O(mp)$ space, we must ensure that, whenever a list $\mathcal{L}_b$ becomes empty, we delete $\mathcal{L}_b$ and $\mathcal{P}_b$ (indeed, the whole node b from $\mathcal{T}_B$). The same happens to an empty set M_t in $\mathcal{T}_{\mathbb{T}}$. However, this means that we may have to rebuild $\mathcal{P}_b$ for each new character b that is inserted, resulting in an overall cost of $O(|\Sigma_A| \log(|\Sigma_A||\Sigma_B|))$. This turns out to be larger than most of the other complexities and results in an overall time of $O(n|\Sigma_A| \log(|\Sigma_A||\Sigma_B|) + mn)$. Alternatively, we can leave those currently unused computations in $\mathcal{P}$ so as to retain our previous complexity, but the space in this case can reach $O(mp+|\Sigma_A||\Sigma_B|)$.

Finally, let us consider the case where $\delta > 0$. Note that now we have ranges of relevant transpositions rather than individual transpositions. Inside each range, the set of δ-matching pairs is the same. The first point is to note that there are at most $4|\Sigma_A||\Sigma_B|$ relevant ranges. Consider that each b_j of B induces a segment $[b_j - \delta, b_j + \delta]$ of alphabet values it matches. Imagine a sequence of these segments in increasing order (they are all of the same length). Now, for each a_i of A we consider the induced segment $[a_i - \delta, a_i + \delta]$. If we slide this segment in the alphabet range, the upper limit will touch all the $2|\Sigma_B|$ beginnings and endings of segments, and the same will happen to the lower limit. Except for those points, the set of matching pairs between A and

B cannot change. Hence there are at most $4|\Sigma_A||\Sigma_B|$ relevant transposition ranges.

We proceed as before, with the only difference that each pair (a_i, b_j) will produce a segment $[b_j - a_i - \delta, b_j - a_i + \delta]$ where (i, j) is active, so $\mathcal{T}_{\mathbb{T}}$ will store (equal length) segments, which can overlap. We will fill the values in $\mathcal{T}_{\mathbb{T}}$ as before; each pair (i, j) will be added to a single segment of $\mathcal{T}_{\mathbb{T}}$. The only new problem that appears is that, before, we had at the end all the M_t sets already computed in $\mathcal{T}_{\mathbb{T}}$ at the end, but now we are not yet ready.

We have to traverse $\mathcal{T}_{\mathbb{T}}$ in increasing order of range endpoints. For each new range endpoint (beginning or ending) we have a new transposition range to process. For each such range, we know which ranges of $\mathcal{T}_{\mathbb{T}}$ are currently open, and we merge all the (i, j) pairs of all the open ranges (the pairs are already sorted inside each node of $\mathcal{T}_{\mathbb{T}}$). This merging can be done at $O(\log(\delta/\mu))$ cost per element extracted, since there can be at most $O(\delta/\mu)$ overlapping transpositions. Since overall we produce $O(mn\delta/\mu)$ pairs, the extra cost over the above scheme is $O(mn(\delta/\mu)\log(\delta/\mu))$. $\square$

6.2 Computing the Longest Common Subsequence

For LCS (and thus for d_{ID}) there exist algorithms that depend on r. The classical Hunt-Szymanski [22] algorithm has running time $O(r \log n)$ if the set of matches M is already given in the proper order. Using Lemma 9 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 [2,17]; Eppstein et. al. [17] 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, e.g., [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 tight estimate, since it can be achieved when $D = O(mn/D)$ 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 Sect. 6.4, we can achieve $O(r \log\log m)$ complexity, which gives $O(mn \log\log m)$ for the transposition invariant LCS (which is better than the previous bound, since $m \leq n$).

We will need the following (sparsity) lemma to give a fast algorithm for d_{ID}. Let $(i', j') <^p (i, j)$ denote the partial order defined as $i' < i$ and $j' < j$.

Lemma 11 *The recurrence (3) can be replaced by*

$$d_{i,j} = \min \{\, d(i', j') + i - i' + j - j' + \textbf{\textit{if}}\, a_i = b_j\ \textbf{\textit{then}}\ -2\ \textbf{\textit{else}}\ 0 \\ \mid a_{i'} = b_{j'}, (i', j') <^p (i, j)\}, \tag{7}$$

where $d_{0,0} = 0$ *and* $a_0 = b_0$.

PROOF. Consider the evaluation of the matrix (d_{ij}) as a shortest path computation in which one can either proceed one cell down (cost 1), one cell to the right (cost 1) or one cell forward in the diagonal (cost ∞ if the corresponding characters do not match, otherwise 0). The paths that take only horizontal

and vertical steps from cell (i', j') to cell (i, j) have cost $i - i' + j - j'$. The paths that consist of one diagonal movement (from $(i-1, j-1)$ to (i, j)) and otherwise of horizontal and vertical movements (from (i', j') to $(i-1, j-1)$) from cell (i', j') to cell (i, j) have cost $i - i' - 1 + j - j' - 1$, when $a_i = b_j$. The paths that take more diagonal steps either have cost ∞ or pass through some cell $(i'', j'') \neq (i', j')$ such that $(i'', j'') <^p (i, j)$, $a_{i''} = b_{j''}$. Using induction, one can see that the path cost from (i'', j'') plus $d_{i'', j''}$ is always smaller or equal to the path cost from (i', j') plus $d_{i', j'}$. $\Box$

The obvious strategy to use the above lemma is to keep the already computed values $d(i', j')$ for each i', j' such that $a_{i'} = b_{j'}$ in some data structure so that their minimum can be retrieved efficiently when computing the value of $d(i, j)$. One difficulty here is that the values stored are not comparable as such since we want the minimum just after $i - i' + j - j' - 2$ is added. This can be solved by storing 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 from range $(i', j') \in [-\infty, i) \times [-\infty, j)$, we need a dynamic data structure that can support one-dimensional range queries (the column-by-column traversal order guarantees that all points are in range $[-\infty, j)$). In addition, the range query should not be output sensitive; it should only report the minimum value, not all the points in the range.

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

Lemma 12 *A* balanced binary tree $\mathcal{T}$ *supports the following operations in*

$O(\log n)$ amortized time, where n is the amount of elements inserted in the tree.

- *$\mathcal{T}.Insert(k, v)$: Inserts value v into tree with key k.*
- *$\mathcal{T}.Delete(v)$: Deletes all elements with value $\geq v$.*
- *$v = \mathcal{T}.Minimum(k, +)$: Returns the minimum of values that have key $> k$.*
- *$v = \mathcal{T}.Minimum(k, -)$: Returns the minimum of values that have key $< k$.*
- *$v = \mathcal{T}.Minimum(l, r)$: Returns the minimum of values that have key $> l$ and $< r$.*

PROOF. The balanced tree described above is easily updated when a new key k is inserted, as the only additional operation is to change the value v' of any traversed internal node by $\min(v, v')$. Deletion needs a parallel tree organized by value v, so that deleting all the values larger than v can be done by disconnecting $O(\log n)$ subtrees. This parallel tree stores pointers to the original tree $\mathcal{T}$, so we can remove also the nodes from $\mathcal{T}$. Since we remove all values larger than v, minimum values computed at internal nodes in $\mathcal{T}$ need not be updated. So the deletion of each node takes $O(\log n)$ time. Since one cannot delete more elements than those inserted, the amortized time for deletions is $O(\log n)$. Minimum over ranges of keys are obtained by taking the minimum value over the $O(\log n)$ nodes that are traversed when searching for the keys. For simplicity we will speak of the balanced tree $\mathcal{T}$, ignoring the fact that the data structure is composed of two trees. (We note, however, that deletions are strictly necessary only when matching with α–limited gaps.) □

We are ready to give the algorithm. To simplify the exposition, we assume first that there is only one match in each column. Now, initialize a balanced binary

tree $\mathcal{T}$ by adding the value of $d_{0,0} - i - j = 0$ with key $i = 0$ ($\mathcal{T}.Insert(0,0)$). Proceed with the match set M that is sorted in column-by-column order and make the following operations at each pair (i, j):

(1) Take the minimum value from $\mathcal{T}$ whose key is smaller than the current row number i ($d = \mathcal{T}.Minimum(i, -)$). Add $i + j - 2$ to this value ($d \leftarrow d + i + j - 2$).

(2) Add the current value d minus the current row number i and current column number j into $\mathcal{T}$ with the current row number as a key ($\mathcal{T}.Insert(i, d - i - j)$).

Finally, if $a_m = b_n$ then $d_{\mathrm{ID}}(A, B) = d$, otherwise $d_{\mathrm{ID}}(A, B) = \mathcal{T}.Minimum(m + 1) + m + n$.

One can easily see that the above algorithm works correctly; the column-by-column evaluation and the range query restricted by the row number in $\mathcal{T}$ guarantee that the $(i', j') <^p (i, j)$ condition holds, as long as there is only one match in each column. To remove the "one match per column assumption", one can simply batch and delay the insert-operations until all the minimum-operations in that column are executed [3].

Clearly, the time complexity is $O(r \log r)$. Figure 1 gives an example.

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

[3] We note, however, that including those cells as soon as they are computed does not alter the result, as Lemma 11 could have also been proved using the definition of $(i', j') <^p (i, j)$ as $i' < i$ and $j' \leq j$.

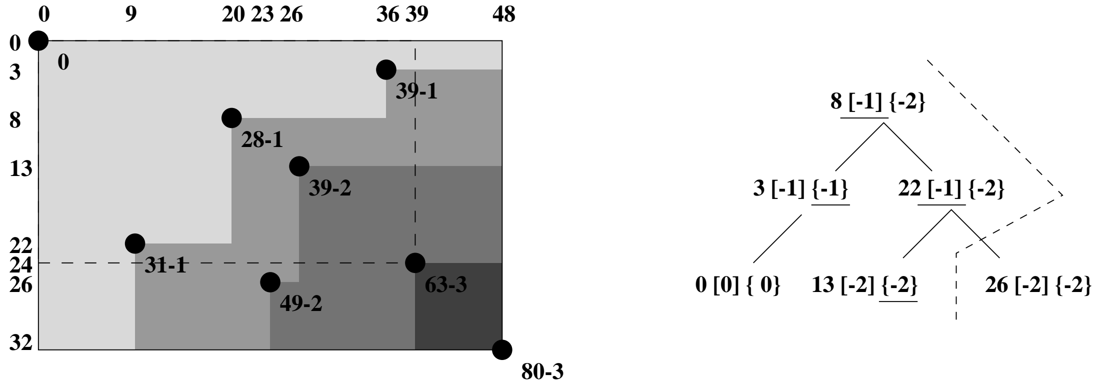

Fig. 1. Example of computation of d_{ID} on a sparse matrix. The black circles represent the matching pairs (i, j). Each such matrix position has an influence area represented by a gray rectangle (darker grays represent larger differences from the standard value $i + j$). Near to each position we represent the matrix value we compute, in the form $i+j-x$. 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 $[x]$ $\{y\}$, where i is the tree key, x means that the cell value is $i + j - x$, and y is the minimum x value in the subtree. The search for cell (24,39) includes all the nodes to the left of the dashed line, and has to take the minimum m over all the underlined values. Its new value is $24 + 39 + m - 1$.

$d_{m,j+s} \leq d + (m - i) + s$ in the last row, which can be used either to keep the minimum $d_{m,j}$ value, or to report all values $d_{m,j} \leq k$ in thresholded searching (each $d_{i,j}$ induces a range at last row where values are $\leq k$; after computing all values $d_{i,j}$, the last row can be traversed by keeping book on the active ranges in order to report each occurrence only once). The time complexity does not change except for the size of the output, but it can be improved since $n >> m$; we can delete those nodes that cannot give the minima, i.e., values d such that $\min(i, d + i + j - 2) = i$. This means that, before we process elements in column j, we can remove all the values $v \geq -j + 2$. The

running time becomes $O(r \log m)$ with $O(m^2)$ space, since this is the number of possibly relevant matches at any time.

We will show in Sect. 6.4 that the balanced binary tree can be replaced by a priority queue. Moreover an implementation of priority queue can be used that supports operations in $O(\log \log u)$ time, where $1 \ldots u$ is the range of values inserted in the structure. The structure does not store the values of $d_{i,j}$ but the row numbers i, and thus we can replace $\log n$ with $\log \log m$.

Let us now consider the case with α–limited gaps. There are couple of changes we need in our algorithms to make sure that, in order to compute $d_{i,j}$, we only take into account the matches that are in the range $(i', j') \in [i - \alpha - 1, i) \times [j - \alpha - 1, j)$. What we need to do is to change the range $[-\infty, i)$ into $[i - \alpha - 1, i)$ in $\mathcal{T}$, as well as to delete elements in column $\leq j - \alpha - 1$ after processing elements in column j. The former is easily accomplished by using query $\mathcal{T}.Minimum(i - \alpha - 2, i)$ at phase (1) of the algorithm. The latter needs an extra tree organized by j values, similar to the one used for the *Delete* operation. In fact, for searching, this tree can replace the one used for *Delete* and we would obtain the same running time, as the relevant α values cannot exceed m in the search problem. However, the reduction to priority queues does not work anymore, and the $\log \log m$ factor must be replaced by $\log n$ in the bounds.

There is one more complication in the case of α–limited gaps. If $\mathcal{T}.Minimum(i - \alpha - 2, i) = \infty$ and thus there is no match inside the query rectangle, we must delete substrings $A_{1 \ldots i-1}$ and $B_{1 \ldots j-1}$. In this case we must use update rule $d \leftarrow i + j - 2$ in phase (1) of the algorithm. Symmetric case happens in the end of the sequences (if (i, j) is the last match, then substrings

$A_{i+1...m}$ and $B_{j+1...n}$ must be deleted); when computing the value of $d_{\text{ID}}(A, B)$, we must take the minimum over $d_{i,j}+m-i+n-j$, where values $d_{i,j}$ are those computed during the execution of the algorithm. This minimum can easily be maintained during the execution of the algorithm.

An illustration of the algorithm for LCS with α–limited gaps is given in Figure 2.

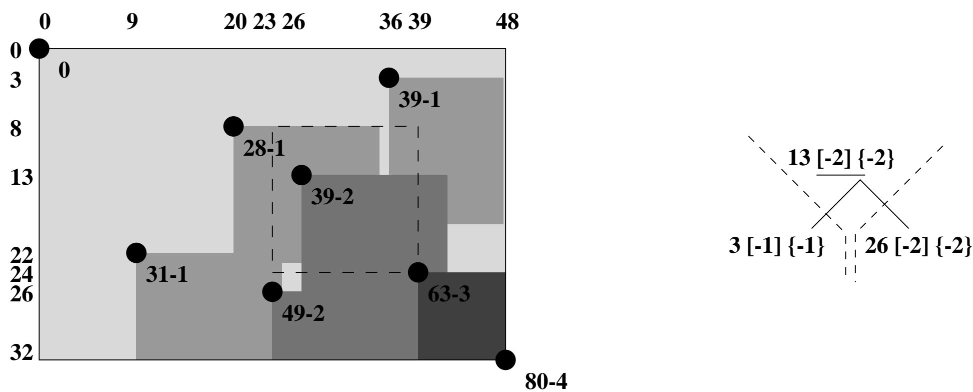

Fig. 2. Example of α–gapped computation of d_{ID} on a sparse matrix, for $\alpha = 15$. The same conventions of Figure 1 apply. The difference is that now the influence areas are restricted to width and height α, so we delete values with too small columns and perform a two-sided range search over the tree, so only its middle part qualifies. In this example, the final result does not change.

By using Lemma 9 and the above algorithms, we get the following result.

Theorem 13 *The transposition invariant distance* $d^{\text{t}}_{\text{ID}}(A, B)$ *can be computed in* $O(mn \log\log m)$ *time and* $O(mn)$ *space. The corresponding search problem can be solved in* $O(mn \log\log m)$ *time and in* $O(m^2)$ *space. For the case of* α*–limited gaps,* $d^{\text{t},\alpha}_{\text{ID}}(A, B)$*, the space requirements remain the same, but the time bounds are* $O(mn \log n)$ *for distance computation and* $O(mn \log m)$ *for searching. The preprocessing bounds in Lemma 10 need to be added to these bounds.*

Note that to achieve space complexity $O(m^2)$ we need to slide a window of length $2m$ over the text, and run preprocessing and computation in parallel so that all transpositions are evaluated in each window.

6.3 Computing the Levenshtein Distance

For the Levenshtein distance, there exists a $O(r \log\log \min(r, mn/r))$ sparse dynamic programming algorithm [17,20]. 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 range query approach to the LCS. This new algorithm can be further generalized to solve the problem of α–limited gaps.

The Levenshtein distance d_{L} has a sparsity property similar to the one given for d_{ID} in Lemma 11. The following lemma can be proven using similar arguments as in the proof of Lemma 11.

Lemma 14 *The recurrence (4) can be replaced by*

$$\begin{aligned} d_{i,j} = \min \{\, & d(i',j') + j - j' + \textbf{\textit{if}}\, a_i = b_j \textbf{\textit{ then }} -1 \textbf{\textit{ else }} 0 \\ & \mid a_{i'} = b_{j'}, i' < i, i - i' \leq j - j'\} \\ & \cup \{d(i',j') + i - i' + \textbf{\textit{if}}\, a_i = b_j \textbf{\textit{ then }} -1 \textbf{\textit{ else }} 0 \\ & \mid a_{i'} = b_{j'}, j' < j, i - i' > j - j'\}, \end{aligned} \tag{8}$$

where $d_{0,0} = 0$ *and* $a_0 = b_0$.

As with the LCS, our goal is to compute only values $d_{i,j}$ such that $a_i = b_j$. The recurrence relation is however much more complex than the one for d_{ID}. In the case of d_{ID} we could store values $d_{i',j'}$ (such that $a_{i'} = b_{j'}$) in a comparable

format (by storing $d_{i',j'}-i'-j'$ instead) so that the minimum of range $(i',j') <^p (i,j)$ 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 $j-j'-1$, when $a_i = b_j$.

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

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

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

The problem now is the lower region. We could use row-by-row traversal to handle this case efficiently, but then we would have the symmetric prob-

lem with the upper region. No traversal order will allow us to limit to one-dimensional range searches in both regions simultaneously; we will need two-dimensional range searches in one of them. Let us consider the two-dimensional range search for the lower region. We would need a query that retrieves the minimum of values $d_{i',j'} - j'$ whose coordinates satisfy $i' < i, j' - i' \leq j - i$. We make a coordinate transformation to make this triangle region into a rectangle; we map each value $d_{i',j'} - j'$ into an xy-plane to coordinate $i', j' - i'$. What we need in this plane, is a rectangle query $[-\infty, i) \times [-\infty, j - i)$. We will in Lemma 15 specify an abstract data structure for $\mathcal{L}$ that supports this operation, and will later in this section show that such a structure exists.

Lemma 15 *There is a data structure* $\mathcal{R}$ *that, after* $O(n \log n)$ *time preprocessing, supports the following operations in amortized* $O(\log n \log \log n)$ *time and* $O(n \log n)$ *space, where* n *is the number of elements in the structure:*

- $\mathcal{R}.Update(x, y, v)$*: Update value at coordinate* x, y *to* v *(under condition that the current value must be larger than* v*).*
- $v = \mathcal{R}.Minimum(l_1, l_2, -, -)$*: Retrieve the minimum of values whose* x*-coordinate is smaller than* l_1 *and* y*-coordinate is smaller than* l_2*.*

We are now ready to give the sparse dynamic programming algorithm for the Levenshtein distance. As with the algorithm for LCS, we first assume that there is only one match in each column, to simplify the exposition. 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 15) with the triples (i, j, ∞) such that $(i, j) \in M \cup \{(0, 0)\}$. 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 $M = \{(i, j) \mid a_i = b_j\}$ that is

sorted in column-by-column order and make the following operations at each pair (i, j):

(1) Take the minimum value from $\mathcal{U}$ whose key is larger or equal to the current diagonal $j - i$ ($d' = \mathcal{U}.Minimum(j - i - 1, +)$). Add $i - 1$ to this value ($d' \leftarrow d' + i - 1$).
(2) Take the minimum value from $\mathcal{L}$ inside the rectangle $[-\infty, i) \times [-\infty, j - i)$ ($d'' = \mathcal{L}.Minimum(i, j - i, -, -)$). 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, $d_{\mathrm{L}}(A, B) = \min(\mathcal{U}.Minimum(n - m - 1, +) + m, \mathcal{L}.Minimum(m + 1, n - m, -, -) + n)$.

The correctness of the algorithm should be clear from the above discussion. To remove the "one match per column assumption", one can batch and delay the insert-operations until all the minimum-operations in that column are executed, just like in the algorithm for LCS (again, we note that this is not really necessary). The time complexity is $O(r \log r \log \log r)$ (r elements are inserted and updated into the lower region structure, and r times it is queried). The space usage is $O(r \log r)$. We can reduce the time complexity to $O(r \log r \log \log m)$ since the $\log \log n$ factor in Lemma 15 is actually $\log \log u$, where $1 \ldots u$ is the the range of values added to the (secondary) structure (see Sect. 6.4). We can implement the structure in Lemma 15 so that $u = m$.

Figure 3 gives an example.

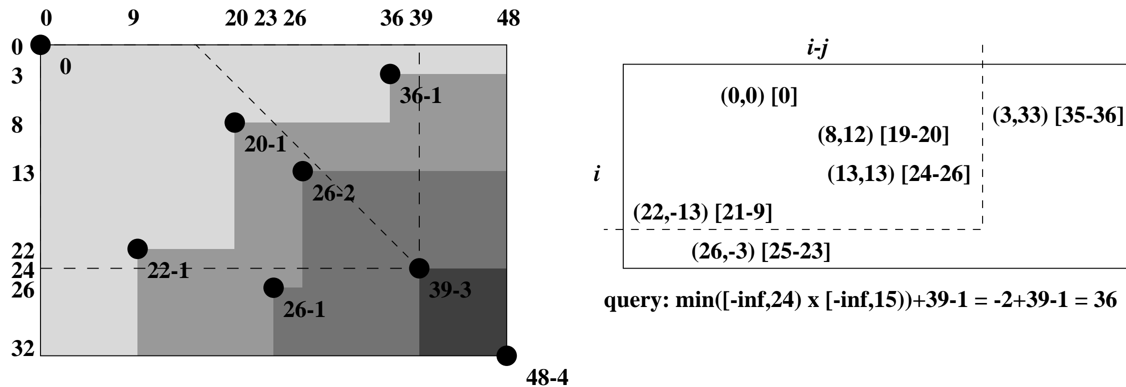

Fig. 3. Example of computation of d_{L} on a sparse matrix. The same conventions of Figure 1 apply. To represent the value of cell (i, j) we use the notation $a - x$, which indicates that this value was obtained from cell (i', j'), with value $a' - x'$, such that $a = a' + \max(i - i', j - j')$ and $x = x' - 1$. We also 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 $y = a - x$ 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.

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. However, removing unnecessary elements from the structures (those that can not give minima for the current column) is not anymore possible, since the structure for the lower region is semi-static; points can not be removed so that the structure would remain balanced. However, we can partition the text into $O(n/m)$ substrings of length $3m$ so that the consecutive substrings overlap by m characters. Then we can run the algorithm on each piece at a time, and no occurrences will be missed, since the values in column j can not affect

the values in column $j + 2m$. This gives $O(r \log m \log \log m)$ search time and $O(m^2 \log m)$ space usage.

Using this algorithm, the transposition invariant distance computation can be solved in $O(mn \log n \log \log m)$ time, and the search problem in $O(mn \log m \log \log m)$ time. These are, by a $\log n$ factor, worse than what can be achieved by using the algorithm of Eppstein et. al [17] (that algorithm can be also generalized to the search problem similarly as above).

However, the advantage of our range query approach is that we can now easily solve the case of α–limited gaps. Consider the lower region. We need the minimum over the values whose coordinates (i', j') satisfy $i' \in [i - \alpha - 1, i)$, $j' \in [j - \alpha - 1, j)$, and $j' - i' \in [-\infty, j - i)$. We map each $d_{i',j'} - j'$ into three dimensional space to coordinate $(i', j', j' - i')$. As we will show in the next subsection, the data structure of Lemma 15 can be generalized to answer three-dimensional (orthant) queries of the form $\mathcal{R}.Minimum(l_1, l_2, l_3, -, +, -)$ (minimum value of points whose first coordinate is smaller than l_1, second larger than l_2, and third smaller than l_3). We can use query $\mathcal{R}.Minimum(i, j - \alpha - 2, j - i, -, +, -)$ when computing the value of $d_{i,j}$ from the lower region, since $i' \geq i - \alpha - 2$ when $j' - i' \leq j - i$, and column-by-column order guarantees that $j' < j$. The upper region case is now symmetric and can be handled similarly. The data structure $\mathcal{R}$ can be implemented so that we get overall time complexity $O(r \log^2 r \log \log m)$. For the search problem, this can be reduced to $O(r \log^2 m \log \log m)$.

As in the case of LCS with α–limited gaps, we still need to consider separately the case where the query area contains no matches. Then we must delete/substitute substrings $A_{1\ldots i-1}$ and $B_{1\ldots j-1}$. In this case we must use up-

date rule $d \leftarrow \max(i,j)-1$ when computing the value of $d_{i,j}=d$. Symmetric case happens in the end of the sequences (if (i,j) is the last match, then substrings $A_{i+1\ldots m}$ and $B_{j+1\ldots n}$ must be deleted/substituted); when computing the value of $d_{\mathrm{L}}^{\alpha}(A,B)$, we must take the minimum over $d_{i,j}+\max(m-i,n-j)$, where values $d_{i,j}$ are those computed during the execution of the algorithm. This minimum can easily be maintained during the execution of the algorithm.

An illustration of the algorithm for Levenshtein distance with α–limited gaps is given in Figure 4.

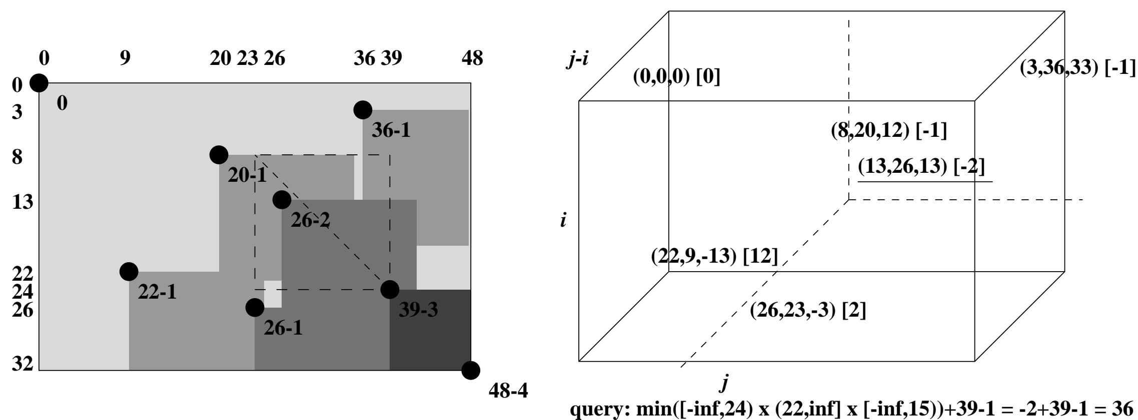

Fig. 4. Example of computation of α–gapped d_{L} on a sparse matrix. The same conventions of Figure 3 apply. On the right we show now the three-dimensional range search structure, where cell (i,j) is represented at position $(i,j,j-i)$ and its value is $[y-j]$, where y is the cell value. This time, a similar structure is needed for the upper area, but we have not represented it. To solve cell (24,39) we take the minimum in the range $[-\infty,24)\times(39-15-2,\infty]\times[-\infty,39-24)$. The area is that inside the dashed cube on the right, where we have underlined the only point that falls inside. This query returns -2, and we add $j-1$ to it to obtain 36.

Using Lemma 9 with the above algorithms, we obtain the following result for the transposition invariant case.

Theorem 16 *Transposition invariant Levenshtein distance $d_{\mathrm{L}}^{\mathrm{t}}(A,B)$ can be*

computed in $O(mn \log\log n)$ time and in $O(mn)$ space. The corresponding search problem can be solved in $O(mn \log\log m)$ time and $O(m^2)$ space. For the case of α–limited gaps, $d_L^{t,\alpha}(A, B)$, the time requirements are $O(mn \log^2 n \log\log m)$ and $O(mn \log^2 m \log\log m)$, and space requirements $O(mn \log^2 n)$ and $O(m^2 \log^2 m)$, respectively, for distance computation and for searching. The preprocessing bounds in Lemma 10 need to be added to these bounds.

6.4 Range Searching for Minima.

We will now describe the data structure $\mathcal{R}$ of Lemma 15. Let S be a *labeled* finite set of points in two-dimensional Euclidean space. The size of S is $n = |S|$. By "labeled" we mean that there is a function $\ell : S \to \mathbb{R}$ that gives a label $\ell(s)$ for each point $s \in S$. The *minimum label range query* problem is to retrieve the minimum label $\ell(s)$ over points $s \in S$ that belong to some query rectangle $[l, r] \times [b, t]$. Efficient solutions for this problem are given by Gabow, Bentley, and Tarjan [19]. We review these solutions here and give some alternative (easier to describe) solutions to keep our exposition as self-contained as possible.

When the set S is static, the one-dimensional case of the problem can be solved as follows [19]. Sort S in increasing order and construct an array $A[1 \ldots n]$ of the labels in that order. Then construct a Cartesian tree [33] on the array A, and preprocess the tree for *least common ancestor queries (LCA)*. Range minimum queries can now be answered by two binary searches on A to find the first i and the last j entry inside the query, and a least common ancestor query to find the minimum value among $A[i], A[i+1], \ldots A[j]$ in $O(1)$ time

[21]. See [4] for a more detailed description of the connection between range minimum queries and LCA.

The two-dimensional version can then be solved by first constructing a balanced binary tree with points in S as leaves and x-coordinate as the search key (actually this can be seen as a range tree [6]). Each internal node v of the tree contains a list of points in S (in order of y-coordinate); the lists are defined recursively as follows. Node v contains a subset of the points in the list of its parent such that the x-coordinate of each point is less than the parent's key if v is the left child, or such that the x-coordinate is greater or equal to the parent's key if v is the right child. An array A like above is constructed for each such list, and each of them is preprocessed to answer (discrete) minimum range queries in $O(1)$ time. The two-dimensional range query $[l, r] \times [b, t]$ can now be answered as follows. Find each node of the tree such that the associated point list is totally inside the x-range $[l, r]$, and whose parent's list is not. For each such node make two binary searches and a range minimum query to find the minimum value from the y-range $[b, t]$. The minimum over all these nodes is the minimum value from range $[l, r] \times [b, t]$. The overall search time is $O(\log^2 n)$, since there are at most $O(\log n)$ nodes whose lists must be queried, and each binary search takes at most $O(\log n)$ time. This can be further reduced to $O(\log n)$ by using *fractional cascading* (see e.g. [5]); the arrays of a parent and a child can be linked such a way that if the first and the last entries that belong to the query range in the parent array are known, then the corresponding entries in the child array can be found by following the links from the parent array. This has the effect that the binary searches are only needed in one node; in its subtree the entries are found by following the links.

So far we have discussed the static case. We would need a semi-static version,

where the labels of the points can be updated. This case can be handled by replacing the above arrays with balanced binary trees; each node of the primary x-coordinate search tree contains a secondary tree which is the balanced binary tree of Lemma 12 with y-coordinate as the key, and the label as the value. We can conclude that updates and two dimensional range queries for minimum can be supported in $O(\log^2 n)$ time in this structure. It is also easy to see that the structure can be constructed in $O(n \log n)$ time (we can sort the points in both x- and y-order, and then construct each binary tree in linear time).

What is left is to reduce $O(\log^2 n)$ to $O(\log n \log\log n)$. This improvement hardly can be achieved for the general case where the query rectangle is limited in all directions. However, we are interested in a query of the form $[-\infty, l) \times [-\infty, t)$ (this is called orthant searching [19]). Consider the one-dimensional case $[-\infty, l)$. We will show (following [19]) that it is enough to use a priority queue to solve this problem. First, it is enough to store those points s whose label is the minimum in the range $[-\infty, s]$. We keep these points (actually their indices in the sequential order) in a queue Q and associate the labels with the priorities. When inserting a new point p, we can test whether its label is smaller than the label of the point $s = Q.predecessor(p)$ that would precede it in the queue. If it is not, we do not insert the point. Otherwise we insert the point, and remove points $Q.successor(p), Q.successor(Q.successor(p)), \ldots$ until we find a point t whose label is smaller than the label of p. This guarantees that a range query $[-\infty, l)$ can be answered by $Q.predecessor(l)$.

These operations on a queue can be supported in $O(\log\log n)$ time (amortized time for insert) using the priority queue of Van Emde Boas [31,32]. Note that this $O(\log\log n)$ bound requires that the inserted values are in the

range $[1 \ldots n]$, which is the case here. Replacing the balanced binary tree of Lemma 12 with this priority queue, we have proven Lemma 15.

The general case of $d > 2$-dimensional orthant searching for minimum can be solved in $O(\log^{d-1} n \log \log n)$ time and in $O(n \log^{d-1} n)$ space, by constructing these range trees for higher dimensions recursively.

6.5 Episode Matching

Finally we look at the episode matching problem and the $d_{\mathrm{D}}^{\mathrm{t}}$ distance, which has a simple sparse dynamic programming solution. The following lemma for d_{D} is easy to prove.

Lemma 17 *The recurrence (5) can be replaced by*

$$d_{i,j} = d(i-1, j') + j - j' - 1, \tag{9}$$

where j' *is the largest* $j' < j$ *such that* $a_{i-1} = b_{j'}$, $d_{0,0} = 0$, *and* $a_0 = b_0$.

Consider an algorithm that traverses the match set $M = \{(i,j) \mid a_i = b_j\}$ in the column-by-column order. We will maintain for each row a value $c(i)$ that gives the largest $j' < j$ such that $a_i = b_{j'}$, and a value $d(i) = d_{i,j'}$. First, initialize these values to ∞, except that $c(0) = 0$ and $d(0) = 0$. Let $(i,j) \in M$ be the current pair whose value we need to evaluate. Then $d = d_{i,j} = d(i-1) + j - c(i-1) - 1$. We can now update the values of the current row: $c(i) = j$ and $d(i) = d$. One can easily see that the above recurrencies can be implemented using dynamic programming in $O(r)$ time, $r = |M|$ (preprocessing time for constructing M needs to be added to this).

The above algorithm generalizes to the search problem and to the episode matching problem by implicitly initializing values $c(0) = j - 1$ and $d(0) = 0$ for the values in the first row.

A similar algorithm can be derived from the recurrence (6), which also gives the start points of the occurrences without needing to backtrack the computation as when (9) is used.

Also the problem of α–limited gaps can be handled easily; we simply avoid updating $d(i)$ as defined when $j-c(i-1)-1 > \alpha$. In this case we set $d(i) = \infty$.

Theorem 18 *The episode matching problem can be solved in $O(|\Sigma|+m+n+r)$ time in integer alphabet and $O((m+n)\log|\Sigma_A|+r)$ time in real alphabet (both in $O(m+n+r)$ space). The transposition invariant episode matching problem can be solved in $O(mn)$ time. The same bound applies in the case of α–limited gaps. The preprocessing bounds in Lemma 10 need to be added to the bounds for the transposition invariant cases.*

6.6 Generalizations to $d^{t,\delta}$ Distances

If we allow a tolerance $\delta > 0$ between matched characters then $\sum_{t\in\mathbb{T}} r_t^\delta = (1 + 2\lfloor\delta\rfloor)mn$, where r_t^δ is the number of matched characters between $A + t$ and B (in integer alphabet), and $\mathbb{T} = \{b_j - a_i - \delta\} \cup \{b_j - a_i + \delta\}$. As in the proof of Lemma 10, it is easy to see that $\mathbb{T}$ is the relevant set of transpositions. Therefore we get complexities $O(\delta mn \log\log(\delta n))$, $O(\delta mn \log\log(\delta n))$, and $O(\delta mn)$ for $d_{\mathrm{ID}}^{t,\delta}$,$d_{\mathrm{L}}^{t,\delta}$, and $d_{\mathrm{D}}^{t,\delta}$, respectively. These finally reduce to $O(\delta mn \log\log n)$, $O(\delta mn \log\log n)$, and $O(\delta mn)$, because in the worst case every character may match every other, and in this case we would process m^2n^2 pairs of charac-

ters, and still $\log(m^2n^2) = O(\log n)$. In real alphabet the result still holds provided we replace δ by δ/μ, where $\mu = \min\{|a_i - a_j| \mid 1 \le i < j \le m,\ a_i \neq a_j\} \cup \{|b_i - b_j| \mid 1 \le i < j \le n,\ b_i \neq b_j\}$.

7 Conclusions and Future Work

We have studied two techniques for solving transposition invariant string matching problems. The first one was to 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.

The second technique, for more general "edit distance" measures, was a more brute force approach since all transpositions were considered. However, since most of the transpositions produce sparse instances for the edit distance matrix, specialized algorithms could be used to solve these sparse instances efficiently. These kind of sparse dynamic programming algorithms already existed in the literature; we gave new sparse dynamic algorithms for episode matching and for matching with α–limited gaps in the LCS and in the unit cost Levenshtein distance. The problem of matching with α–limited gaps demonstrated the connection between sparse dynamic programming and the problem of semi-static range searching for minima.

An interesting remaining question is whether the log factors could be avoided to achieve $O(mn)$ for transposition invariant edit distance. For episode matching we achieved the $O(mn)$ bound, except that the preprocessing can (in very uncommon situations on real alphabets) take $O(mn \log n)$ 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 [30] can be generalized to give bounds like $O(|\mathbb{T}|dn)$, where $\mathbb{T}$ is the set of transpositions and $d = d_*^{\mathrm{t}}(A, B)$. This algorithm can be combined with the sparse evaluation to get an algorithm that is fast both in practice and in the worst case.

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] B. S. Baker and R. Giancarlo. Sparse dynamic programming for longest common subsequence from fragments. *J. of Algorithms*, 42:231–254, 2002.

[4] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. 4th Latin American Symposium on Theoretical Informatics (LATIN 2000)*, pp. 88-94, 2000.

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

[6] J. L. Bentley. Multidimensional divide-and-conquer. *Comm. ACM*, 23:214–229, 1980.

[7] 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, Winter 2001.

[8] 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.

[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. 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.

[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, Spring 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'2002)*, pp. 679–688. ACM-SIAM, 2002.

[15] 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.

[16] 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, October 2001.

[17] D. Eppstein, Z. Galil, R. Giancarlo, and G. F. Italiano. Sparse dynamic programming I: linear cost functions. *J. of the ACM* 39(3):519–545, July 1992.

[18] K. Fredriksson. *Rotation Invarinat Template Matching.* PhD Thesis, Department of Computer Science, University of Helsinki, 139 pages, 2001.

[19] 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.

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

[21] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13:338–355, 1984.

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

[23] 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, April 12-14, 2000.

[24] 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, April 17-20, 2000.

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

[26] C. Iliopoulos, M. Crochemore, G. Navarro, and Y. Pinzón. A bit-parallel suffix automaton approach for (δ, γ)–matching in music retrieval. Submitted for publication, 2002.

[27] 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.

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

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

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

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

[32] 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.

[33] J. Vuillemin. A unifying look at data structures. *Comm. ACM*, 23(4):229–239, 1980.

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

[35] 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.

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