Restricted Transposition Invariant Approximate String Matching Under Edit Distance

Heikki Hyyrö

Department of Computer Sciences, University of Tampere, Finland.
heikki.hyyro@gmail.com

Abstract. Let A and B be strings with lengths m and n, respectively, over a finite integer alphabet. Two classic string mathing problems are computing the edit distance between A and B, and searching for approximate occurrences of A inside B. We consider the classic Levenshtein distance, but the discussion is applicable also to indel distance. A relatively new variant [8] of string matching, motivated initially by the nature of string matching in music, is to allow transposition invariance for A. This means allowing A to be "shifted" by adding some fixed integer t to the values of all its characters: the underlying string matching task must then consider all possible values of t. Mäkinen et al. [12, 13] have recently proposed $O(mn \log\log m)$ and $O(dn \log\log m)$ algorithms for transposition invariant edit distance computation, where d is the transposition invariant distance between A and B, and an $O(mn \log\log m)$ algorithm for transposition invariant approximate string matching. In this paper we first propose a scheme to construct transposition invariant algorithms that depend on d or k. Then we proceed to give an $O(n + d^3)$ algorithm for transposition invariant edit distance, and an $O(k^2 n)$ algorithm for transposition invariant approximate string matching.

1 Introduction

Let Σ be a finite integer alphabet of size σ so that each character in Σ has a value in the range $0, \ldots, \sigma - 1$. We assume that strings are composed of a finite (possibly length-zero) sequence of characters from Σ. The length of a string A is denoted by $|A|$. When $1 \leq i \leq |A|$, A_i denotes the ith character of A. The notation $A_{i..h}$, where $i \leq h$, denotes the substring of A that begins at character A_i and ends at character A_h. Hence $A = A_{1..|A|}$. String A is a subsequence of string B if B can be transformed into A by deleting zero or more characters from it.

Let $ed(A, B)$ denote the edit distance between strings A and B. For conveniency, the length of A is m and the length of B is n throughout the paper, and we also assume that $m \leq n$. In general, $ed(A, B)$ is defined as the minimum number of edit operations that are needed in transforming A into B, or vice versa. In this paper we concentrate specifically on Levenshtein distance (denoted by $ed_L(A, B)$), which allows a single edit operation to insert, delete or substitute a single character. But the methods are applicable also to indel distance (denoted by $ed_{id}(A, B)$), which differs only in that it does not allow substitutions.

Indel distance is interesting for example because it is related to $llcs(A, B)$, the length of the longest common subsequence between A and B, by the formula $2 \times llcs(A, B) = m + n - ed_{id}(A, B)$.

If A takes the role of a pattern string and B the role of a text string, approximate string matching is defined as searching for those locations j in B where $ed(A, B_{h..j}) \leq k$ for some $h \leq j$. Here k is a predetermined error threshold.

An interesting variation of string comparison/matching, allowing *transposition invariance*, was proposed recently by Lemström and Ukkonen in [8] in the context of music comparison and retrieval. If musical pieces are stored as sequences of note pitches and we want to find a melody pattern p (ie. a string whose characters are note pitch values) from a music database, it may be natural not to care about the overall pitch at which p is found. This leads to the basic idea of transposition invariant matching: to allow A to match at any pitch. Transposition invariance has also other possible uses. Mäkinen et al. [13] mention several, such as time series comparison.

A transposition is represented as an integer t. Characters A_i and B_j match under transposition t if $A_i + t = B_j$. With the finite integer alphabet Σ, the set of all possible transpositions is $\mathbb{T} = \{b - a \mid a, b \in \Sigma\}$. Let $A + t$ denote string A that has been shifted by transposition t. That is, the ith character of $A + t$ is $A_i + t$. The notation $ed^{\mathrm{t}}(A, B)$ denotes the transposition invariant edit distance between A and B. It requires us to find the minimum distance over all transpositions, ie. $ed^{\mathrm{t}}(A, B) = \min\{ed(A + t, B) \mid t \in \mathbb{T}\}$. Then the task of transposition invariant approximate matching is to find locations j in B where $ed^{\mathrm{t}}(A, B_{h..j}) \leq k$ for some $h \leq j$.

At the moment there is no algorithm that is able to compute $ed^{\mathrm{t}}(A, B)$ more efficiently in the worse case than the approach of simply computing the distances $ed(A + t, B)$ separately for each possible transposition t. Hence the current non-trivial solutions, including the ones we propose, are concentrated on how to compute each distance $ed(A + t, B)$ efficiently [12, 13], and on building heuristics on how to quickly discard possible transpositions from further consideration [9].

We will assume for the remaining part if the paper that the notion of edit distance refers to the Levenshtein distance (e.g. $ed(A, B) = ed_L(A, B)$).

2 Dynamic programming

The classic $O(mn)$ solution [19] for computing $ed(A, B)$ is to fill an $(m + 1) \times (n + 1)$ dynamic programming table D, where the cell $D[i, j]$ will eventually hold the value $ed(A_{1..i}, B_{1..j})$. Under Levenshtein distance it works as follows. First the boundary values of D are initialized by setting $D[i, 0] = i$ and $D[0, j] = j$ for $i \in 0, \ldots, m$ and $j \in 1, \ldots, n$. Then the remaining cells are filled recursively so that $D[i, j] = D[i - 1, j - 1]$ if $A_i = B_j$, and otherwise $D[i, j] = 1 + \min\{D[i - 1, j - 1], D[i - 1, j], D[i, j - 1]$. The recurrence for indel distance is very similar.

The dynamic programming method can be modified to conduct approximate string matching by changing the boundary initialization rule $D[0, j] = j$ into $D[0, j] = 0$ [16].

2.1 Greedy filling order

Let diagonal q be the up-left to low-right diagonal in D whose cells $D[i, j]$ satisfy $j - i = q$. Ukkonen [17] proposed a greedy algorithm for computing edit distance. The cells in D are filled in the order of increasing distance values $0, 1, \ldots$. The algorithm is based on the well-known facts that the values along a diagonal q are non-decreasing, and that moving from one diagonal to another costs one operation. Let $L[r, q]$ denote the lowest row on diagonal q of D that has a value less or equal to r, and define $L[r, q] = -1$ if such row does not exist. Thus $L[r, q] = \max\{i \mid D[i, i + q] \leq r \vee i = -1\}$. The greedy method can be used as follows under Levenshtein distance. First the values $L[r, q]$ are initialized with the value -1. Once the values $L[r - 1, q]$ are known, the values $L[r, q]$ can be computed using the rule $L[r, q] = \min\{m, s + lcp(s + 1, q + s + 1)\}$, where $s = \max\{L[r - 1, q - 1], L[r - 1, q] + 1, L[r - 1, q + 1] + 1\}$, and $lcp(i, j)$ is the length of the longest common prefix between $A_{i..m}$ and $B_{j..n}$. Fig. 1 shows an example. The value $lcp(i, j)$ can be computed in constant time by using the method of Chang and Lawler [1], which requires $O(n)$ time preprocessing. Hence the greedy algorithm is able to compute $d = ed(A, B)$ in $O(n + d^2)$ time, as at most $O(d)$ diagonals are processed and a single diagonal involves $O(d)$ computations.

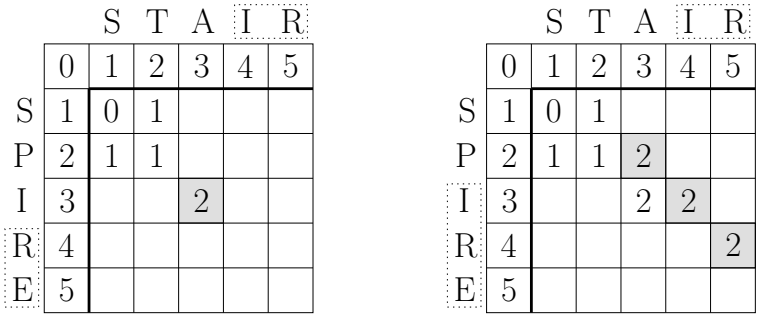

Fig. 1. Assume the values $L[1, -1] = 2$, $L[1, 0] = 2$, and $L[1, 1] = 1$ are already known. (*Left*) When computing the value $L[2, 0]$, we have $q = 0$, $s = \max\{L[1, -1], L[1, 0] + 1, L[1, 1] + 1\} = 3$, and $lcp(s + 1, q + s + 1) = lcp(4, 4) = 0$. Hence $L[2, 0] = 3$.
(*Right*) When computing the value $L[2, 1]$, we have $q = 1$, $s = \max\{L[1, 0], L[1, 1] + 1, L[1, 2] + 1\} = 2$, and $lcp(s, q + s) = lcp(3, 4) = 2$. Hence $L[2, 1] = 4$.

3 Sparse dynamic programming

Sparse dynamic programming concentrates only on the matching points $D[i, j]$ where $A_i = B_j$. Let $\mathbb{M}(t) = \{(i, j) \mid A_i + t = B_j\}$ be the set of matching points under transposition t. A single set $\mathbb{M}(t)$ can be represented in linear space and generated in $O(n \log n)$ time [5]. By following [13], the sets $\mathbb{M}(t)$ can be computed

for all relevant transpositions in $O(\sigma + mn)$ time. There are $|\mathbb{M}(t)| = O(mn)$ matching points under a given transposition t. The overall number of matches under all relevant[1] transpositions is of this same complexity: $\sum_{t \in \mathbb{T}} |\mathbb{M}(t)| = mn$, as each point is a match point for exactly one value of t. This observation was abstracted in [13] into the following Lemma, which reduces the solution of transposition invariant matching into finding an efficient sparse dynamic programming algorithm for the basic case without transposition invariance.

Lemma 1 ([13]). *If distance $ed(A, B)$ can be computed in $O(g(|\mathbb{M}(t)|)f(m, n))$ time, where g is a convex (concave up) increasing function, then the distance $ed^t(A, B)$ can be computed in $O(g(mn)f(m, n))$ time.*

For edit distance, a sparse recurrence can be derived in a straightforward manner from the corresponding dynamic programming recurrence. Let the notation $(i', j') \prec (i, j)$ mean that $i' < i$ and $j' < j$, and we also say that in this case (i', j') *precedes* (i, j). The following sparse scheme for Levenshtein distance is adapted from Galil and Park [4]. First we initialize $D[0, 0] = 0$. Then each value $D[i, j]$ where $A_i = B_j$ is computed recursively by setting $D[i, j] = \min\{D[i', j'] + \max\{i - i', j - j'\} \mid (i', j') \in \mathbb{M}(t) \cup (0, 0) \wedge (i', j') \prec (i, j)\} - 1$. Fig. 2 shows an example. After computing the values at all matching points, the possibly still uncomputed values $D[m, j]$ can be computed with $O(n)$ extra cost. Approximate string matching is achieved by adding the value i as one possible choice in the minimum clause. The decisive factor for efficiency of sparse dynamic programming is how the preceding points (i', j') that give minimal distances at $D[i, j]$ are found.

		S	T	R	I	P	E
	0	1	2	3	4	5	6
S	1	0	1	2	3	4	5
P	2	1	1	2	3	3	4
I	3	2	2	2	2	3	4
R	4	3	3	2	3	3	4
E	5	4	4	3	3	4	3

		S	T	R	I	P	E
	0	1	2	3	4	5	6
S	1	0					
P	2					3	
I	3				2		
R	4			2			
E	5						3

Fig. 2. (*Left*) A completely filled dynamic programming matrix for computing ed("SPIRE","STRIPE"). (*Right*) The same computation using sparse dynamic programming. Only the cell $(0, 0)$ and the match points are considered (shown in grey). When the cell $(i, j) = (5, 6)$ is computed, the preceding cells (i', j') are (0,0), (1,1), (2,5), (3,4), and (4,3). The corresponding values $D[i', j'] + \max\{i - i', j - j'\}$ are 6, 5, 3, 2, and 3, respectively. The minimum value 2 corresponds to $(i', j') = (3, 4)$. This leads into setting $D[5, 6] = D[3, 4] + 2 - 1 = 3$.

[1] A transposition is relevant if it leads into at least one match between A and B.

Galil and Park [4], by following the framework of Eppstein et al. [2], discussed a scheme that is able to compute the distances $D[i, j]$ for all $(i, j) \in \mathbb{M}(t)$ in overall time of $O(|\mathbb{M}(t)| \log\log(\min\{|\mathbb{M}(t)|, mn/|\mathbb{M}(t)|\}))$. They process the points in row-wise manner for increasing i, and within each row for increasing j, and maintain an *owner list*. The point (i', j') is the owner of (i, j) if it results in the minimum distance at (i, j) in the sparse dynamic programming recurrence. On row i, the owner list contains the column indices of the owners of the points (i, j) on row i. Hence its size is $O(n)$, and all its key values are integers in the range $1, \ldots, n$. The list of owners at each point (i, j) can be updated by doing an amortized constant number of insert, delete and lookup operations on a priority queue. This results in the overall cost $O(|\mathbb{M}(t)| \log\log(\min\{|\mathbb{M}(t)|, mn/|\mathbb{M}(t)|\}))$ if the priority queue is implemented using Johnson's data structure [6]. When the values stored in the priority queue are integers in the range $0, \ldots, z$, this data structure facilitates a homogenous sequence of $r \leq z$ insertions/deletions/lookups (all of the same type) in $O(r \log\log(z/r))$ time.

Mäkinen et al. [13] proposed to use two-dimensional range minimum queries in finding the minimum point (i', j'). They achieved $O(|\mathbb{M}(t)| \log\log m)$ time for indel distance and $O(|\mathbb{M}(t)| \log m \log\log m)$ time for Levenshtein distance by using the data structure of Gabow et al. [3]. With the exception of computing indel distance, Mäkinen et al. resorted to processing B in segments of $O(m)$ match points within the matrix D (distance), or $O(m)$ characters (approximate matching), in order to achieve the preceding time boundaries. They achieved also $O(|\mathbb{M}(t)| \log\log m)$ in all cases by applying the segmenting techniques to the approach of Eppstein et al. [2].

We note here that the above-mentioned segmenting technique is not necessary for achieving $O(|\mathbb{M}(t)| \log\log m)$: the method of Eppstein et al. does not internally rely on the typical assumption that $m \leq n$. If we switch the roles of the string pair so that A becomes B and vice versa, then that method leads directly into the run time $O(|\mathbb{M}(t)| \log\log m)$: now the owner list contains integers in the range $1, \ldots, m$, in which case each operation on Johnson's data structure takes $O(\log\log m)$ time. Taking this into account changes their time bound into $O(|\mathbb{M}(t)| \log\log(\min\{m, |\mathbb{M}(t)|, mn/|\mathbb{M}(t)|\}))$.

4 Restricting the computation

Now we are ready to present our technique for restricting the computation with transposition invariant edit distance. The first building block is the following Lemma that is essentially similar to the idea of so-called counting filter [7, 14].

Lemma 2. *Let A and B be two strings and D be a corresponding dynamic programming table that has been filled as described in section 2. The condition $D[i, j] \leq k$ can hold only if the substring $B_{j-h+1..j}$, where $h = \min\{i, j\}$, matches at least $i - k$ characters of $A_{1..i}$.*

Using the preceding lemma, we get the main rule for restricting the computation.

Lemma 3. *Let $c > 1$ be a constant, and ck and j be positive integers that fulfill the conditions $k < ck \leq m$ and $1 \leq j \leq n$. There exists at most $O(k)$ different transpositions t for which $D^t[ck, j] \leq k$.*

Proof. By Lemma 2, the length-ck prefix $A_{1..ck}$ can match a substring ending at B_j with at most k errors only if the substring $B_{j-h+1..j}$, where $h = \min\{ck, j\}$, matches at least $ck - k$ characters of $A_{1..i}$. The corresponding $ck \times h$ submatrix of D, spanning rows $1, \ldots, ck$ and columns $j-h+1, \ldots, j$, contains at most c^2k^2 character-pairs (A_i, B_j). Since $A_i + t = B_j$ for only one transposition t, there can be at most $c^2k^2/(ck - k) = k(c^2/(c-1)) = O(k)$ different transpositions that have at least $ck - k$ matches. □

The value $c^2/(c-1)$ in Lemma 3 gets its minimum value, 4, when we choose $c = 2$. In the following section we propose how to use this Lemma.

5 The main algorithm

Assume first that we wish to compute the accurate value $ed^{\mathrm{t}}(A, B)$ only if it is at most k. We can use Lemma 3 in solving this problem efficiently. The first step is to conduct sparse dynamic programming on a restricted part of D.

Step 1: partial sparse dynamic programming Consider a given relevant transposition t. In the first step we compute $ed(A_{1..2k} + t, B)$ by using sparse dynamic programming. During (or after) this process we compute the values $D[2k, j]$ for $j = 1, \ldots, n$ and record the values that are $\leq k$. This contains several elements. The first is an ordered list *matchPos* that contains the pair $(j, D[2k, j])$ if $D[2k, j] \leq k$. The *matchPos* list is in ascending order according to j. The positions j are also recorded into $k + 1$ ordered lists $matchPos(p)$ for $p = 0, \ldots, k$. The list $matchPos(p)$ contains in ascending order all positions j where $D[2k, j] = p$.

At this point we have all relevant information of D until row $2k$. Fig. 3 illustrates.

Step 2: match extension Let *consPos* be an ordered list of the positions and lengths of maximal groups of consecutive values $D[2k, j] \leq k$. The pair (x, y) appears in *consPos* if and only if $D[2k, j] \leq k$ for $j = x, \ldots, x + y - 1$ and it is not true that $D[2k, x - 1] \leq k$ or $D[2k, x + y] \leq k$. We do not actually store this complete list, as it can be recovered during a single linear time traversal of the list *matchPos*. We start the traversal from the beginning of *matchPos*, and every time an item (x, y) is recovered, the traversal is suspended until the following checking phase is completed for that (x, y). Fig. 4a illustrates. Then we recover the next item (x, y), and so on until the *matchPos* has been completely traversed.

The final checking step applies a variation of a greedy edit distance algorithm [17] (Section 2.1) over each group of consecutive potentially match-seeding positions. This stage checks which, if any, of the cells $D[2k, j]$ recorded in *matchPos*

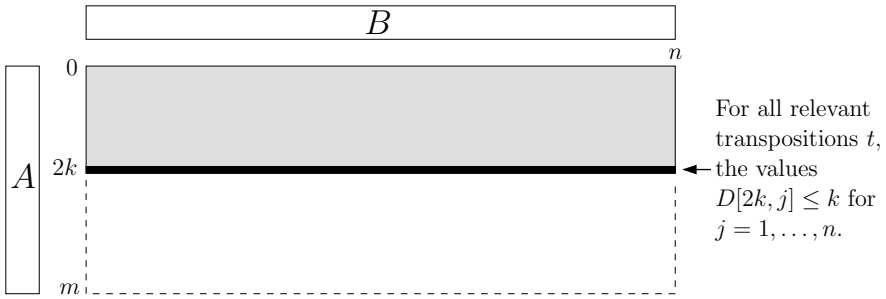

Fig. 3. For each relevant transposition, the values $D[2k, j] \leq k$ are recorded for $j = 1, \ldots, n$ by using sparse dynamic programming within rows $0, \ldots, 2k$ of the dynamic programming matrix (shaded area).

array can be extended to a full match between A and a prefix of B within edit distance k. This is done separately for each item (x, y) in $consPos$. The consecutive cells $D[2k, x], \ldots, D[2k, x + y - 1]$ that correspond to (x, y) lie on the diagonals $x - 2k, \ldots, x - 2k + y - 1$ in D. It is enough to consider these diagonals using the greedy approach. Other type of diagonals are already known to contain a value larger than k, and values along a diagonal never decrease. The basic greedy algorithm needs to be modified in order to handle the fact that values on row $2k$ do not originally follow the greedy computing order.

For simplicity, we consider a $(m - 2k + 1) \times y$ submatrix D' of D in which $D'[i, j]$ is equal to $D[2k + i, x + j]$. We use the prime symbol to mean that a previously used construct addresses via D' instead of D. For example the value $L'[r, q]$ equals $L[r, x - 2k + q] - 2k$ (see Section 2.1). We also use the notation $m' = m - 2k$.

We use two auxiliary arrays of size $y = O(n)$ in our greedy algorithm. The first array is called $initDist$, and it initially contains the values $initDist[j] = D'[0, j] = D[2k, x + j]$ for $j = 0, \ldots, y - 1$. These values are set by re-traversing the items of $matchPos$ that were included in recovering the currently processed item (x, y) of $consPos$. The second array, called $sortDist$, contains the indices $0, \ldots, y - 1$ sorted according to the value of the corresponding entry in $initDist$. Fig. 4b illustrates. This way $initDist[sortDist[z]] \leq initDist[sortDist[z + 1]]$ for $z = 0..y - 2$. Also $sortDist$ can be initialized in linear time by extracting the values from the $matchPos(p)$ lists.

We also use values $readyCount$ and $currDist$. The value $readyCount$ tells how many diagonals have been completely processed, and it is first initialized to 0. The value $currDist$ tells the distance values that will be expanded/added next in $L'[r, q]$. That is, we will next update values of form $L'[currDist, q]$ among the diagonals that are still not completely processed. Initially $currDist$ is equal to the minimum value in $initDist + 1$, and this can be first computed as $initDist[sortDist[0]] + 1$.

First all values $L'[r, q]$ are initialized by setting $L'[r, q] = -1$ for $q = 0, \ldots, y - 1$ and $r = 0, \ldots, k$. Then the list $initDist$ is traversed for $j = 0, \ldots, y - 1$ and

at each j we set $L'[initDist[j], j] = lcp(2k+1, x+j+1)$. This lets the current value $D'[0, j] = D[2k, x+j]$ propagate along the diagonal j of D' as long as there are consecutive matching characters, if any. This in part makes the values $L'[r, q]$ more compatible with the basic greedy algorithm.

Before beginning a new round of iterations, the greedy algorithm starts from position $z = readyCount$ of $sortDist$. The positions $readyCount, \ldots, y-1$ of $sortDist$ always refer to the still unprocessed diagonals in ascending order of their last computed value (highest r for which $L'[r, q]$ is computed along the corresponding diagonal q). Fig. 4c illustrates. At the beginning of each iteration, at position z, the algorithm checks if $z < y$ (did we already process all diagonals?) and $initDist[sortDist[z]] < currDist$ (did we already process all diagonals that may get new values $currDist$?). If the check is successful, then the diagonal that corresponds to $sortDist[z]$ is processed, as described soon, and the value of z is incremented. If the check fails, the algorithm begins a new iteration if $readyCount < y$. The value $currDist$ is incremented before the next iteration.

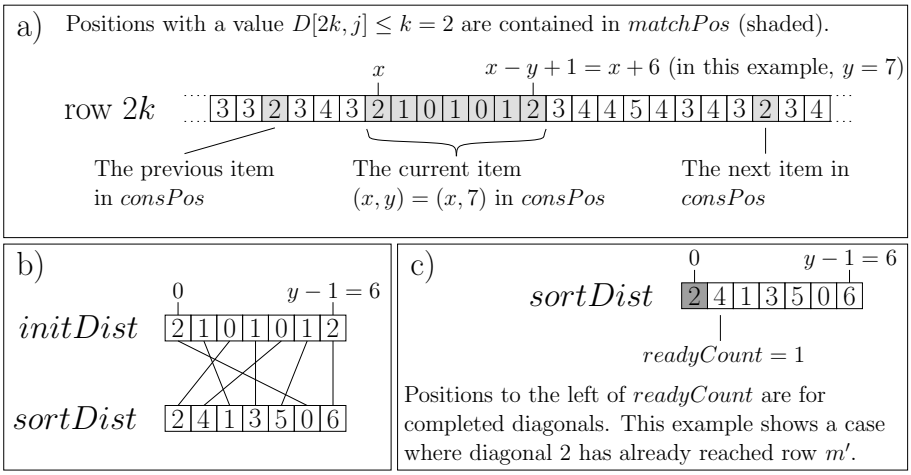

Fig. 4. Examples of the used variables and arrays.

Let us use the shorthand $q = sortDist[z]$ in the following. Processing diagonal q begins by checking whether the value $L'[r-1, q]$ has been properly propagated to the two neighboring diagonals. If $r > 0$, $q > 0$ and $L'[r, q-1] < L'[r-1, q]+1$, we set $L'[r, q-1] = \min\{m', u' + lcp(2k+u'+1, x+q+u'+1)\}$, where $u' = L'[r-1, q]+1$. In similar way, if $r > 0$, $q < y-1$ and $L'[r, q+1] < L'[r-1, q]$, we set $L'[r, q+1] = \min\{m', u'+lcp(2k+u'+1, x+q+u'+1)\}$, where $u' = L'[r-1, q]$. This removes possible anomalies present due to the initial setting that is incompatible with the basic greedy algorithm. The cost of this extra processing is $O(1)$ per processed diagonal, but there could be more efficient ways in practice if one for example records which diagonals no longer need to be checked like this. We also

note that this processing can lead into finding a match. But we let the match be discovered and handled sometime later by the following step (the same is true for the initial extension of the values from row $2k$).

Finally we update the value $L[r,q]$ itself. Because we do know that all diagonals have already been processed at least up to the value $currDist-1$, we may use the formula from section 2.1 as such, and set $L'[r,q] = \min\{m', s'+lcp(2k+s'+1, x+q+s'+1)\}$, where $s' = \max\{L'[r-1,q-1], L'[r-1,q]+1, L'[r-1,q+1]+1\}$. If the new value is $L'[r,q] = m'$, we record the match, interchange the values $sortDist[z]$ and $sortDist[readyCount]$ in $sortDist$, and increment $readyCount$. The matches can be recorded into a size-y array so that the occurrences can be reported in the end in $O(y)$ time. After recording value $L'[r,q]$, we also check whether the value affects the neighboring diagonals.

As a final note, we would like to note how to compute the lcp-values efficiently. A naive way would require us to do separate preprocessing for $A+t$ for each transposition t. But this can be avoided by doing the preprocessing for $\overline{A}$ and $\overline{B}$, where $|\overline{A}| = m-1$, $|\overline{B}| = n-1$, $\overline{A}_i = A_{i+1} - A_i$ for $i = 1,\ldots,m-1$, and $\overline{B}_j = B_{j+1} - B_j$ for $j = 1,\ldots,n-1$. Let $lcpT(i,j,t)$ be the value of $lcp(i,j)$ under transposition t. Now $lcpT(i,j) = 0$ if $A_i + t \neq B_j$, and otherwise $lcpT(i,j) = 1 + \overline{lcp}(i,j)$, where $\overline{lcp}(i,j)$ is as $lcp(i,j)$ but for $\overline{A}$ and $\overline{B}$ instead of A and B.

6 Analysis

The time complexity of the algorithm described in the preceding section is as follows. The first stage of sparse computation takes a total time of $O(|\mathbb{M}(t)| \log\log k)$ over all relevant transpositions. These computations produce a total of $O(kn)$ match points for further checking. The initialization phase before the greedy algorithm takes linear time in the number of checked matches, that is, $O(1)$ per diagonal. The greedy algorithm spends $O(k)$ time per diagonal (ie. match point), which makes the checking cost $O(k^2 n)$ time. This complexity dominates the overall time.

In order to conduct approximate string matching, we only need to change the stage of sparse dynamic programming. This does not change the cost, and hence we have $O(k^2 n)$ time for approximate string matching.

The $O(k^2 n)$ procedure for thresholded edit distance computation can be made $O(n + k^3)$ by considering only an $O(k)$ diagonal band in D. And this then enables us to obtain $O(n + d^3)$ time for computing $d = ed^{\mathrm{t}}(A,B)$. We first do the computation with a limit $k = 1$, and then double k until the resulting computation manages to find a value $ed^{\mathrm{t}}(A,B) \leq k$. The last round spends $O((2d)^3) = O(d^3)$ time, and the total time is bounded by $O(d^3) \times \sum_{h=0}^{\infty}(1/2^h)^3 = O(d^3) \times O(1) = O(d^3)$. The term $O(n)$ comes from preprocessing (for example, reading the input strings).

The space requirements for sparse dynamic programming are typically dominated by the sets $\mathbb{M}(t)$, but a lower limit is the size of the input strings. When we restrict the matching sets to contain matches only within the area of D that

participates in sparse dynamic programming, we use $O(n + k^2)$ in thresholded and $O(n+d^2)$ space in edit distance computation. The space required in approximate string matching is $O(nk)$, but by imitating [13], this can be diminished to $O(n+k^2)$ by processing B in overlapping segments of length-$O(k)$. The space requirements for the second phase of extending the matches are of the same order: the dominating factor is the table $L'[r, q]$, whose size in each scenario happens to have the same asymptotic limit as the corresponding total size of the match sets (minus the basis $O(n)$).

7 Conclusion

Transposition invariant string matching is a relatively new and interesting problem proposed by Lemström and Ukkonen [8]. Previously Mäkinen et al. [12, 13] have given $O(mn \log\log m)$ and $O(dn \log\log m)$ algorithms for transposition invariant edit distance computation, where d is the transposition invariant distance between A and B. The same authors also gave an $O(mn \log\log m)$ algorithm for transposition invariant approximate string matching. In the same work, Mäkinen et al. also stated the challenge to develop error-dependent algorithms for transposition invariant string matching. In this paper we have given an initial answer to that challenge by presenting a basic scheme for constructing transposition invariant algorithms that depend on d or k. We then introduced an $O(n + d^3)$ algorithm for transposition invariant edit distance and an $O(k^2 n)$ algorithm for transposition invariant approximate string matching. To the best of our knowledge, these are the first such error-dependent algorithms for this problem.

References

1. Chang, W. I., and Lawler, E. L. Sublinear approximate string matching and biological applications. *Algorithmica*, 12:327–344, 1994.
2. Eppstein, D., Galil, Z., Giancarlo, R., and Italiano, G. F. Sparse dynamic programming I: linear cost functions. *Journal of ACM*, 39(3):519–545, 1992.
3. Gabow, H. N., Bentley, J. L., and Tarjan, R. E. Scaling and related techniques for geometry problems In *Proc. 16th ACM Symposium on Theory of Computing (STOC'84)*, 135–143, 1984.
4. Galil, Z., and Park, K. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science*, 92:49–76, 1992.
5. Hirschberg, D. S. Algorithms for the longest common subsequence problem *Journal of ACM*, 24:664–675, 1977.
6. Johnson, D. B. A priority queue in which initialization and queue operations take $O(\text{loglog } D)$ time. *Mathematical Systems Theory*, 15:295–309, 1982.
7. Jokinen, P., Tarhio, J., and Ukkonen, E. A comparison of approximate string matching algorithms. *Software Practice & Experience*, 26(12):1439–1458, 1996.
8. Lemström, K., and Ukkonen, E. 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)*, 53–60, 2000.

9. Lemström, K., Navarro, G., and Pinzon, Y. Practical algorithms for transposition-invariant string-matching To appear in *Journal of Discrete Algorithms*
10. Landau, G. M., and Vishkin, U. Fast parallel and serial approximate string matching *Journal of Algorithms*, 10:157–169, 1989.
11. Levenshtein, V. I. Binary codes capable of correcting spurious insertions and deletions of ones (original in Russian). *Russian Problemy Peredachi Informatsii 1*, 12–25, 1965.
12. Mäkinen, V, Navarro, G., and Ukkonen, E. Algorithms for transposition invariant string matching. In *Proc. 20th International Symposium on Theoretical Aspects of Computer Science (STACS'03)"*, LNCS 2607, 191–202, 2003.
13. Mäkinen, V, Navarro, G., and Ukkonen, E. Transposition invariant string matching. To appear in *Journal of Algorithms*.
14. Navarro, G. Multiple approximate string matching by counting. In *Proc. 4th South American Workshop on String Processing (WSP'97)*, 125–139, 1997.
15. Navarro, G. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
16. Sellers, P. The theory and computation of evolutionary distances: pattern recognition. *Journal of Algorithms*, 1:359–373, 1980.
17. Ukkonen, E. Algorithms for approximate string matching *Information and Control*, 64:100–118, 1985.
18. van Emde Boas, P. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6:80–82, 1977.
19. Wagner, R., and Fisher, M. The string-to-string correction problem. *Journal of ACM*, 21(1):168–173, 1974.