Algorithms for Transposition Invariant String Matching (Extended Abstract)

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

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

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

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

Topic: Algorithms and data structures.

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 arised in the field of music information retrieval (MIR) [4, 16, 17]. In music analysis and retrieval, one often wants to compare two music pieces to test how similar they are. A reasonable way of modeling music is to consider the pitches and durations of the notes. Often the durations are omitted, too, since it is usually possible to recognize the melody from a sequence of pitches. In this paper, we study distance measures for pitch sequences (of monophonic music) and their computation.

In general, edit distance measures can be used for matching two pitch sequences. There are, however, couple of properties related to music that should be taken into account. Transposition invariance is one of those; the same melody is perceived even if the pitch sequence is shifted from one key to another. Another property is the continuity of the alignment; the size of the gaps between matches should be limited, since one long gap in a central part can make a crucial difference in perception. Small gaps should be accepted, e.g. for removing "decorative" notes.

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

The full version of this paper appears as a technical report [21]. There we also study "easier" (non-gapped) measures of similarity under transposition invariance (like Hamming distance and (δ,γ)–matching [5, 19]), that have other applications besides MIR.

* Supported by the Academy of Finland under grant 22584.

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

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 $A = a_1a_2 \dots a_m$ and $B = b_1b_2 \dots b_n$ 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 consider are symmetric. String A' is a *substring* of A if $A' = A_{i \dots j} = a_i \dots a_j$ for some $1 \leq i \leq j \leq m$. String A'' is a *subsequence* of A, denoted by $A'' \sqsubseteq A$, if $A'' = a_{i_1}a_{i_2} \dots a_{i_{|A''|}}$ for some indexes $1 \leq i_1 < i_2 < \cdots < i_{|A''|} \leq m$.

The following measures can be defined. The length of the *longest common subsequence (LCS)* of A and B is $\mathrm{lcs}(A, B) = \max\{|S| \mid S \sqsubseteq A, S \sqsubseteq B\}$. The *edit distance* [18, 27, 23] 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 $* \in \{\mathrm{L}, \mathrm{ID}, \mathrm{D}\}$ can now be stated as $d^{\mathrm{t}}_*(A, B) = \min_{t \in \mathbb{U}} d_*(A + t, B)$.

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

The *approximate string matching problem*, based on the above distance functions, is to find the minimum distance between A and any substring of B. In this case we call A the *pattern* and denote it $P_{1 \dots m} = p_1p_2 \cdots p_m$, and call B the *text* and denote it $T_{1 \dots n} = t_1t_2 \cdots t_n$, 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 $d(P, T_{j_l \dots j_r}) \leq k$ 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* [20, 8]. It can also be stated as follows: Find the shortest substring of the text that contains the pattern as a subsequence.

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

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 [17]. They proposed (among other measures) transposition invariant longest common subsequence (LCTS) as a measure of similarity between two monophonic music (pitch) sequences. They gave a descriptive nick name for the measure: "Longest Common Hidden Melody". As the alphabet of pitches is some limited integer alphabet $\Sigma \subset \mathbb{Z}$, the transpositions that have to be considered are $\mathbb{T} = \{b - a \mid a, b \in \Sigma\}$. This gives a brute force algorithm for computing the length of the LCTS [17]: 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.

We will also use a brute force approach as described above, but since most transpositions produce sparse instances of the dynamic programming problem, we can use specialized sparse dynamic programming algorithms to get good worst case bounds. Moreover, we show a connection between the resulting sparse dynamic programming problems and semi-static range minimum queries. We obtain simple yet efficient algorithms for the distances $d_{\mathrm{L}}^{\mathrm{t},\alpha}$, $d_{\mathrm{ID}}^{\mathrm{t},\alpha}$, and $d_{\mathrm{D}}^{\mathrm{t},\alpha}$.

For LCS (and thus for d_{ID}) there already exists Hunt-Szymanski [15] 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 [1, 10] 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. [10] have proposed sparse dynamic programming algorithms for more complex distance computations such as Wilbur-Lipman fragment alignment problem [28, 29]. Also the unit cost Levenshtein distance can be solved using these techniques [13]. 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 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^{α}_{ID} it is also easy to improve the bound below to $O(mn)$ using min-deques [12] in each row and column of the DP matrix.

distance	distance evaluation	searching
exact	$O(m)$	$O(m+n)$
$d_{\mathrm{ID}}, d_{\mathrm{L}}$	$O(mn/\log m)$	$O(mn/\log m)$ [7]
d_{D}	$O(m+n)$ (greedy)	$O(mn/\log m)$ [8]
$d^{\alpha}_{\mathrm{ID}}, d^{\alpha}_{\mathrm{L}}$	$O(\alpha mn)$	$O(\alpha mn)$ [22] (Chapter 10, Sect. 4.2)
d^{α}_{D}	$O(m+n)$ (greedy)	$O(mn)$ [9, 6]

Table 2. Upper bounds for transposition invariant string matching. We have not added, for clarity, the size of the output in the (thresholded) search complexity, nor the preprocessing time in Lemma 2. 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	$O(m)$	$O(m+n)$
$d^{\mathrm{t}}_{\mathrm{ID}}$	$O(mn\log\log m)$	$O(mn\log\log m)$
$d^{\mathrm{t},\alpha}_{\mathrm{ID}}$	$O(mn\log n)$	$O(mn\log m)$
$d^{\mathrm{t}}_{\mathrm{L}}$	$O(mn\log\log n)$	$O(mn\log\log m)$
$d^{\mathrm{t}\alpha}_{\mathrm{L}}$	$O(mn\log^2 n\log\log m)$	$O(mn\log^2 m\log\log m)$
$d^{\mathrm{t},\alpha}_{\mathrm{D}}$	$O(mn)$	$O(mn)$

4 Computation of Transposition Invariant Edit Distances

Let us first review how the edit distances can be computed using dynamic programming [18, 27, 23]. Let $A = a_1a_2\cdots a_m$ and $B = b_1b_2\cdots b_n$. For d_{ID}, evaluate an $(m+1)\times(n+1)$ matrix (d_{ij}), $0 \le i \le m$, $0 \le j \le n$, using the recurrence

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

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

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

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

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

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

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

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

with initialization $d_{i,0} = \infty$ for $0 \le i \le m$ and $d_{0,j} = j$ for $1 \le j \le n$. Note that $d_{\mathrm{D}}(A,B)$ can also be computed using a greedy algorithm; the recurrence is only given because it can be generalized to the search problem, too.

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 $\min_{0 \le j \le n} d_{m,j}$. For thresholded searching, we report the endpositions of the occurrences, i.e., those j where $d_{m,j} \le k$.

To solve our tranposition invariant problems, we could try to prove that only some transpositions need to be checked, as is the case with some easier distance measures [21]. 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. $M = M(A,B) = \{(i,j) \mid a_i = b_j, 1 \le i \le m, 1 \le j \le n\}$. 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 $\mathbb{T} = \{b_j - a_i \mid 1 \le i \le m, 1 \le j \le n\}$. 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 1 *If an algorithm computes a distance $d(A,B)$ in $O(g(r(A,B))f(m,n))$ time, where g is a concave function, then the transposition invariant distance $d^{\mathrm{t}}(A,B) = \min_{t \in \mathbb{T}} d(A+t,B)$ can be computed in $O(g(mn)f(m,n))$ time.*

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

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

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

4.1 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 $i' > i$).[1] The match set corresponding to a transposition t will be called $M_t = \{(i,j) \mid a_i + t = b_j\}$.

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

Lemma 2 *The* match sets $M_t = \{(i,j) \mid a_i + t = b_j\}$*, each sorted in column-by-column order, for all transpositions* $t \in \mathbb{T}$*, can be constructed 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.*

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.

The case of real alphabets with $O(mn)$ memory is solved as follows. 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. $\square$

The preprocessing can be made more space-efficient with some penalty in the time requirement; the details can be found in [21].

4.2 Computing the Longest Common Subsequence

For LCS (and thus for d_{ID}) there exist algorithms that depend on r. The classical Hunt-Szymanski [15] algorithm has running time $O(r \log n)$ if the set of matches M is already given in the proper order. Using Lemma 1 we can conclude that there is an algorithm for transposition invariant LCS that has time complexity $O(mn \log n)$. There are even faster algorithms for LCS [1, 10]; Eppstein et. al. [10] improved an algorithm of Apostolico and Guerra [1] 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., [1] for a definition). Using this algorithm, we have the bound $O(mn \log \log n)$ for the transposition invariant case.

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 Appendix A, 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 a partial order defined as $i' < i$ and $j' < j$, and let $(i', j') \leq^p (i,j)$ denote another partial order defined as $i' \leq i$, $j' \leq j$, and $(i', j') \neq (i,j)$.

Lemma 3 *The recurrence (1) can be replaced by*

$$d_{i,j} = \begin{cases} \min\{d(i',j') + i - i' + j - j' \mid a_{i'} = b_{j'}, (i',j') \leq^p (i,j)\} & \textit{when } a_i \neq b_j \\ \min\{d(i',j') + i - i' + j - j' - 2 \mid a_{i'} = b_{j'}, (i',j') <^p (i,j)\} & \textit{when } a_i = b_j \end{cases}, \tag{4}$$

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 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 $(i-1,j-1)$) 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 $(i'',j'') \leq^p (i,j)$, $a_{i''} = b_{j''}$. 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 $d_{i',j'}$. Finally, if $a_i = b_j$, then any path to (i,j) that takes diagonal steps and does not traverse through $(i-1,j-1)$ can be replaced by equal or smaller cost path that traverses through $(i-1,j-1)$. □

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

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

We are ready to give the algorithm. Initialize a balanced binary tree $\mathcal{T}$ by adding the value of $d_{0,0} - i - j = 0$ with key $i = 0$ ($\mathcal{T}.Insert(0,0)$). Proceed with the match set 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, $d_{\mathrm{ID}}(A,B) = \mathcal{T}.Minimum(m+1,-) + m + n$.

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.

Clearly, the time complexity is $O(r \log r)$.

The algorithm also generalizes easily to the search problem; the 0 values in the first row can be added implicitly by using $d \leftarrow \min(i, d+i+j-2)$ in step (1) above. Also, every value $d_{i,j} = d$ computed in step (2) above induces a value $d_{m,j+s} \le d + (m-i) + s$ in the last row, which can be used either to keep the minimum $d_{m,j}$ value, or to report all values $d_{m,j} \le k$ in thresholded searching (each $d_{i,j}$ induces a range at the last row where values are $\le k$; 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 $v \ge -j+2$. 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 Appendix A 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. The only change we need in our algorithms is to make sure that, in order to compute $d_{i,j}$, we only take into account the matches that are in the range $(i',j') \in [i-\alpha-1, i) \times [j-\alpha-1, j)$. What we need to do is to change the range $[-\infty, i)$ into $[i-\alpha-1, i)$ in $\mathcal{T}$, as well as to delete elements in column $\le j-\alpha-1$ 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.

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

Theorem 5 *The transposition invariant distance $d^{\mathrm{t}}_{\mathrm{ID}}(A,B)$ can be computed in $O(mn\log\log m)$ time and $O(mn)$ space. The corresponding search problem can be solved in $O(mn\log\log m)$ time and in $O(m^2)$ space. For the case of α–limited gaps, $d^{\mathrm{t},\alpha}_{\mathrm{ID}}(A,B)$, 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 2 need to be added to these bounds.*

Note that to achieve space complexity $O(m^2)$ we need to slide a window of length m over the text, and run preprocessing and computation in parallel so that all transpositions are evaluated in each window. All occurrences will still be found since values in column j can not affect the values in column $j+m$.

4.3 Computing the Levenshtein Distance

For the Levenshtein distance, there exists a $O(r\log\log\min(r,mn/r))$ sparse dynamic programming algorithm [10, 13]. 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 3. The following lemma can be proven using similar arguments as in the proof of Lemma 3.

Lemma 6 *The recurrence (2) can be replaced by*

$$d_{i,j} = \begin{cases} \min\left\{\begin{array}{l}\{d(i',j')+j-j' \mid a_{i'}=b_{j'}, i'\le i, j'-i'\le j-i\}\ \cup \\ \{d(i',j')+i-i' \mid a_{i'}=b_{j'}, j'\le j, j'-i'> j-i\}\end{array}\right. & \textit{when } a_i\neq b_j \\ \min\left\{\begin{array}{l}\{d(i',j')+j-j'-1 \mid a_{i'}=b_{j'}, i'< i, j'-i'\le j-i\}\ \cup \\ \{d(i',j')+i-i'-1 \mid a_{i'}=b_{j'}, j'< j, j'-i'> j-i\}\end{array}\right. & \textit{when } a_i= b_j \end{cases}, \qquad (5)$$

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

This relation is however much more complex than the one for d_{ID}. In the case of d_{ID} we could store values $d_{i',j'}$ 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 $j-j'-1$.

Let us call the two sets in the lower minimization formula of 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 $d_{i',j'}$ 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 problem with the upper region. No traversal order will allow us to limit to one-dimensional range searches in both regions simultaneously; we will need two-dimensional range searches in one of them. Let us consider the two-dimensional range search for the lower region. We would need a query that retrieves the minimum of values $d_{i',j'} - j'$ whose coordinates satisfty $i' < i, j' - i' \leq j - i$. 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 7 specify an abstract data structure for $\mathcal{L}$ that supports this operation. Such structure is given in [11]; we will review this structure in Appendix A.

Lemma 7 *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. Initialize a balanced binary tree $\mathcal{U}$ for the upper region by adding the value of $d_{0,0} - i = 0$ with key $i = 0$ ($\mathcal{U}.Insert(0, 0)$). Initialize a data structure $\mathcal{L}$ for the lower region ($\mathcal{R}$ of Lemma 7) with the triples (i, j, ∞) such that $(i, j) \in M \cup \{(0, 0)\}$. 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. 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 7 is actually $\log\log u$, where $1 \ldots u$ is the the range of values added to the (secondary) structure (see Appendix A). We can implement the structure in Lemma 7 so that $u = m$.

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 $2m$ 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 $j+m$. 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 [10] (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 Appendix A, the data structure of Lemma 7 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 $i' \geq i-\alpha-1$ 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 $O(r \log^2 m \log\log m)$.

Using Lemma 1 with the above algorithms, we obtain the following result for the transposition invariant case.

Theorem 8 *Transposition invariant Levenshtein distance $d^{\mathrm{t}}_{\mathrm{L}}(A,B)$ 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, $d^{\mathrm{t},\alpha}_{\mathrm{L}}(A,B)$, 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 2 need to be added to these bounds.*

4.4 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 9 *The recurrence (3) can be replaced by*

$$d_{i,j} = d(i-1, j') + j - j' - 1, \tag{6}$$

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.

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 10 *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 2 need to be added to the bounds for the transposition invariant cases.*

5 Conclusions and Future Work

We studied a brute force approach for transposition invariant edit distance computation. However, as we noticed, most of the tranpositions produce sparse instances of the edit distance problem, and 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.

References

1. A. Apostolico and C. Guerra. The longest common subsequence problems revisited. *Algorithmica* 2:315–336, 1987.
2. M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. LATIN 2000)*, pp. 88-94, 2000.
3. J. L. Bentley. Multidimensional divide-and-conquer. *Comm. ACM*, 23:214–229, 1980.
4. T. Crawford, C.S. Iliopoulos, and R. Raman. String matching techniques for musical similarity and melodic recognition. *Computing in Musicology* 11:71–100, 1998.
5. M. Crochemore, C.S. Iliopoulos, T. Lecroq, and Y.J. Pinzón. Approximate string matching in musical sequences. In *Proc. PSC 2001*, pp. 26–36, 2001.
6. 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.
7. M. Crochemore, G. Landau, and M. Ziv-Ukelson. A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. In *Proc. SODA'2002*, pp. 679–688. ACM-SIAM, 2002.
8. G. Das, R. Fleischer, L. Gasieniec, D. Gunopulos, and J. Kärkkäinen. Episode matching. In *Proc. CPM'97*, LNCS 1264, Springer, pp. 12–27, 1997.
9. M.J. Dovey. A technique for "regular expression" style searching in polyphonic music. In *Proc. ISMIR 2001*, pp. 179–185, October 2001.
10. 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.
11. H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. *Proc. STOC'84*, pp. 135–143, 1984.
12. H. Gajewska and R. Tarjan. Deques with heap order. *Information Processing Letters* 12(4):197–200, 1986.
13. Z. Galil and K. Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science* 92:49–76, 1992.
14. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13:338–355, 1984.
15. J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, May 1977.
16. K. Lemström and J. Tarhio. Searching monophonic patterns within polyphonic sources. In *Proc. RIAO 2000* , pp. 1261–1279 (vol 2), 2000.
17. K. Lemström and E. Ukkonen. Including interval encoding into edit distance based music comparison and retrieval. In *Proc. AISB 2000*, pp. 53–60, 2000.
18. V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady* 6:707–710, 1966.
19. 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.
20. H. Mannila and H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *Proc. KDD'95*, AAAI Press, pp. 210–215, 1995.
21. V. Mäkinen, G. Navarro, and E. Ukkonen. Algorithms for Transposition Invariant String Matching. *Technical Report TR/DCC-2002-5*, Department of Computer Science, University of Chile, July 2002, "ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/ti_matching.ps.gz"
22. D. Sankoff and J. B. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley Publishing Company, 1983.
23. P. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *J. of Algorithms*, 1(4):359–373, 1980.
24. P. van Emde Boas, R. Kaas, E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Systems Theory*, 10:99–127, 1977.
25. 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.
26. J. Vuillemin. A unifying look at data structures. *Comm. ACM*, 23(4):229–239, 1980.
27. R. Wagner and M. Fisher. The string-to-string correction problem. *J. of the ACM* 21(1):168–173, 1974.
28. 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.
29. W. J. Wilbur and D. J. Lipman. The contect-dependent comparison of biological sequence. *SIAM J. Appl. Math.* 44(3):557-567, 1984.

A Range Searching for Minima

We will now describe the data structure $\mathcal{R}$ of Lemma 7. 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 [11]. 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 [11]. Sort S in increasing order and construct an array $A[1 \ldots n]$ of the labels in that order. Then construct a Cartesian tree [26] 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 [14]. See [2] 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 [3]). 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*; 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 4 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 [11]). Consider the one-dimensional case $[-\infty, l)$. We will show (following [11]) that it is enough to use a 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. When inserting a new point s_i, we can test whether its label is smaller than the label of the point $s_{i'}$, where $i' = Q.predecessor(i)$ 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 $Q.successor(i), Q.successor(Q.successor(i)), \ldots$ until we find a point $s_{i''}$ whose label is smaller than the label of s_i. This guarantees that a range query $[-\infty, l)$ can be answered by $\ell(s_{Q.predecessor(i_l)})$, where i_l is the rank of l if inserted to S.

It takes $\log |S|$ time to find the rank of l (using binary search in sorted S), so as such, this improvement is not useful in the one-dimensional case. However, since we use this structure multiple times in the nodes of the primary tree for two-dimensional queries, the binary search is only needed once. Also, the structure can be used in the one-dimensional case when points are integers smaller than n; we can store the points in the queue, not their indices, and avoid the binary search in the query.

The above mentioned 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 [24, 25]. 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 4 with this priority queue, we have proven Lemma 7.

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.