Practical Algorithms for Transposition-Invariant String-Matching*

Kjell Lemström [a,1] Gonzalo Navarro [b,2] and Yoan Pinzon [c]

[a] *Dept. of Computer Science, Univ. of Helsinki, Finland.*

[b] *Center for Web Research, Dept. of Computer Science, Univ. of Chile, Chile.*

[c] *Dept. of Computer Science, King's College London, UK.*

Abstract

We consider the problems of (1) longest common subsequence (LCS) of two given strings in the case where the first may be shifted by some constant (that is, transposed) to match the second, and (2) transposition-invariant text searching using indel distance. These problems have applications in music comparison and retrieval. We introduce two novel techniques to solve these problems efficiently. The first is based on the branch and bound method, the second on bit-parallelism. Our branch and bound algorithm computes the longest common transposition-invariant subsequence (LCTS) in time $O((m^2+\log\log\sigma)\log\sigma)$ in the best case and $O((m^2+\log\sigma)\sigma)$ in the worst case, where m and σ, respectively, are the length of the strings and the size of the alphabet. On the other hand, we show that the same problem can be solved by using bit-parallelism and thus obtain a speedup of $O(w/\log m)$ over the classical algorithms, where the computer word has w bits. The advantage of this latter algorithm over the present bit-parallel ones is that it allows the use of more complex distances, including general integer weights. Since our branch and bound method is very flexible, it can be further improved by combining it with other efficient algorithms such as our novel bit-parallel algorithm. We experiment on several combination possibilities and discuss which are the best settings for each of those combinations. Our algorithms are easily extended to other musically relevant cases, such as δ-matching and polyphony (where there are several parallel texts to be considered). We also show how our bit-parallel algorithm is adapted to text searching and illustrate its effectiveness in complex cases where the only known competing method is the use of brute force.

Key words: String matching (of music), branch and bound, bit-parallelism.

* Preliminary ideas of this paper have been presented in [10,11].

[1] FDK Research Unit. Partially supported by Academy of Finland.

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

1 Introduction

Combinatorial pattern matching, with its many application domains, have been an active research field for several decades already. One of the latest such domains is comparing and retrieving symbolically encoded music. Indeed, music can be encoded as sequences of symbols, that is, as strings. At a rudimentary level, this is done by taking into account exclusively the order of the starting times of the musical events (that is, the *note on*s) together with their *pitch* information (or frequency, that is, the perceived height of the musical event). On a more complicated level, one can use several distinct attributes for each of the events (see for example [2,9]). Most of the interesting musical attributes used in such symbolic representations are directly available, for example, in the commonly used MIDI format [15].

Calculating the longest common subsequence (LCS) of two (or more) given strings is one of the fundamental problems in string matching. Let $A = a_1, \ldots, a_m$ and $B = b_1, \ldots, b_n$ be two strings over some finite alphabet. A *subsequence* of either string is obtained by deleting zero, or more characters from it. A LCS of A and B, $L = lcs(A, B)$, is such that L is a subsequence of both A and B, and its length is maximal. In the corresponding indel-distance search problem, given a pattern $P = p_1, \ldots, p_m$ and text $T = t_1, \ldots, t_n$, the task is to find whether there are substrings of T such that P can be obtained from them by performing at most k character deletions or insertions. We will define these two problems more precisely and show their intrinsic connection in Section 2. Let us now discuss the problem framework in terms of computing LCS. It should be understood, however, that the following claims are valid also when solving the corresponding search problem. In what follows, assume $m \leq n$ without loss of generality (this is also the case in text searching).

An algorithm solving LCS would be appropriate for matching music because music contains various kinds of decorations, such as grace notes or ornamentations. By comparing the length of the music strings to the length of the obtained LCS, one gets a useful measure of the essential similarity of the two strings, which happens to be more robust than alternative approaches such as edit distance (where substitutions of characters are permitted in addition to insertions and deletions). [3]

However, there are special features intrinsic to music that are not taken into account in general string matching techniques. The main feature is that western people tend to listen to music analytically by observing the intervals between the consecutive pitch values more than the actual pitch values themselves: A melody performed in two distinct pitch levels is perceived the same regard-

[3] Another possibility to this end would be to use the geometric approach [17,19].

less if its performed in a lower or higher level of pitches. This leads to the concept of *transposition invariance*. Let the alphabet be comprised of integer values: $\Sigma = \{0 \ldots \sigma\}$, and $L + c$ denote a constant adding to every character of string L, that is, $L + c = l_1 + c, l_2 + c, \ldots, l_p + c$. A longest common transposition invariant subsequence (LCTS), denoted $L = lcts(A, B)$, is such that L is a subsequence of A, $L + c$ is a subsequence of B (for some constant c, $-\sigma \leq c \leq \sigma$), and its length is maximal.

The second important feature is that music may be polyphonic, which means that there are several events occurring simultaneously. Given a set T of h strings (each representing a *musical line* or *voice*) of the form $T^g = t_1^g, \ldots, t_n^g$, $g \in \{1 \ldots h\}$, a character of P can match any t_j^g at text position j. Thirdly, in music matching it is often useful to allow some tolerance for the matching pairs. One way to this end is via the so-called δ-matching [3]: $A = a_1, \ldots, a_m$ is said to δ-match $B = b_1, \ldots, b_m$ if $a_i \in [b_i - \delta, b_i + \delta]$ for all $i = 1, \ldots, m$. A more sophisticated alternative is to introduce the possibility of substituting b_i by a_i, at a cost which is proportional to $|b_i - a_i|$. This is called a *weighted* distance because the substitution costs (or weights) are variable.

The are a few studies on LCTS in the current literature. Plain (not transposed) LCS can be computed by using dynamic programming in $O(mn)$ time. A naive way to compute LCTS is to compute LCS for all the possible $2\sigma + 1$ transpositions, in overall time $O(\sigma mn)$ [13]. In [4], Crochemore et al. introduced a bit-parallel algorithm that computes LCS in $O(mn/w)$ time, where w denotes the size of the computer word in bits. Their algorithm can be run for every transposition to obtain $O(\sigma mn/w)$ time. Mäkinen et al. [14] introduced a sparse dynamic programming algorithm for the LCTS problem that works in time $O(mn \log\log m)$, and a more practical version that works in time $O(mn \log m)$.

Polyphony and δ-matching are straightforward features to include in all these approaches. With regard to substitutions, Myers' bit-parallel algorithm [16,8] could be used instead of [4] to allow for them in $O(\sigma mn/w)$ time. If general substitution weights are to be handled, Bergeron et al.'s algorithm [1] can be used. If we give weight λ to insertions and deletions (so as to have, in comparison, smaller integer substitution costs), then this algorithm gives an $O(\sigma mn\lambda \log(\lambda)/w)$ time solution.

In this paper we introduce another bit-parallel algorithm which is specifically aimed at the LCTS problem. Unlike Crochemore et al.'s adapted algorithm, ours solves several transposition instances in a single computation. Our algorithm turns out to be more flexible: In addition to dealing with transpositions, polyphony and δ-matching, we can deal with general weights such as the aforementioned $|b_i - a_i|$. The cost of this flexibility is mild: Our time complexity is $O(\sigma mn \log(m)/w)$, which represents a speedup of $\Omega(w/\log m)$

over the naive algorithm. Although the bit-parallel algorithm [4] yields better complexity, it cannot deal with general weights. The competing algorithm for this extended case [1] has better complexity for long enough patterns, where $\log m = \Omega(\lambda \log \lambda)$. Actually, our experimental results show that our bit-parallel algorithm is the fastest existing choice for $m \leq 30$.

Moreover, we introduce another novel approach to solve LCTS, which is based on the branch and bound technique to search for the optimal transposition. In the worst case the algorithm runs in time $O((mn + \log \sigma)\sigma)$, which is not much worse than the naive solution. In the best case, however, it can be as good as $O((mn + \log \log \sigma) \log \sigma)$. Moreover, the technique can be combined with any of the aforementioned algorithms to obtain a faster solution. Our experimental results show that this algorithm is the fastest when comparing long sequences ($m \geq 120$).

The aforementioned algorithms [4,14,16,8,1] are rather easily adapted to text searching. This is also the case for our novel bit-parallel algorithm, but unfortunately not for the branch and bound technique.

The paper is organized as follows. In the next section, we introduce the appropriate basics of the string matching framework and define the problems considered . Then, in Section 3, we show how to compute LCTS by using the branch and bound technique. Section 4 introduces our novel bit-parallel algorithm and show how it can be used to speed up both the naive and the branch and bound algorithms. Sections 5 and 6 deal with the text searching problem and introduce our novel bit-parallel algorithm for this task. In Section 7 we show the results of our comprehensive experiments before concluding the paper in Section 8.

2 Preliminaries

Let us start this section with a brief introduction to string combinatorics. Let Σ be a finite set of symbols, called an *alphabet*. Then any $A = (a_1, a_2, \ldots, a_m)$ where each a_i is a symbol in Σ, is a *string* over Σ. Usually we write $A = a_1, \ldots, a_m$. The *length* of A is $|A| = m$. The string of length 0 is called the *empty string* and denoted ϵ. The set of strings of length i over Σ is denoted by Σ^i, and the set of all strings over Σ by Σ^*. If a string A is of the form $A = \beta\alpha\gamma$, where $\alpha, \beta, \gamma \in \Sigma^*$, we say that α is a *factor* (substring) of A. Furthermore, β is called a *prefix* of A, and γ a *suffix* of A. A string A' is a *subsequence* of A if it can be obtained from A by deleting zero or more symbols, that is, $A' = a_{i_1}, a_{i_2}, \ldots, a_{i_m}$, where $i_1 \ldots i_m$ is an increasing sequence of indices in A.

To define a distance between strings over Σ^*, one should first fix the set of *local*

transformations (editing operations) $\mathcal{T} \subseteq \Sigma^* \times \Sigma^*$ and a non-negative valued *cost function* W that gives for each transformation t in $\mathcal{T}$ a cost $W(t)$. Each t in $\mathcal{T}$ is a pair of strings $t = (\alpha, \beta)$. Regarding such a t as a rewriting rule suggests a notation for t, $\alpha \to \beta$ (α is replaced by β within a string containing α), which we will use below. For convenience, if $\alpha \to \beta \notin \mathcal{T}$, then we assume $W(\alpha \to \beta) = \infty$.

The definition of a distance is based on the concept of a *trace*, which gives a correspondence between two strings. Formally, a trace between two strings A and B over Σ^*, is formed by splitting A and B into equally many factors:

$$\tau = (\alpha_1, \alpha_2, \ldots, \alpha_p; \beta_1, \beta_2, \ldots, \beta_p),$$

where $A = \alpha_1, \alpha_2, \ldots, \alpha_p$, and $B = \beta_1, \beta_2, \ldots, \beta_p$, and each α_i, β_i (but not both) may be an empty string over Σ. Thus, string B can be obtained from A by steps $\alpha_1 \to \beta_1,\ \alpha_2 \to \beta_2, \ldots, \alpha_p \to \beta_p$.

The cost of the trace τ is $W(\tau) = W(\alpha_1 \to \beta_1) + \ldots + W(\alpha_p \to \beta_p)$. The distance between A and B, denoted $D_{\mathcal{T},W}(A, B)$, is defined as the minimum cost over all possible traces.

The general definition above induces, the following well-known distance measures. In *unit-cost edit distance* (or Levenshtein distance), $D_L(A, B)$, the allowed local transformations are of the forms $a \to b$ (substitution), $a \to \epsilon$ (deletion), and $\epsilon \to a$ (insertion), where $a, b \in \Sigma$. The costs are given as $W(a \to a) = 0$ for all a, $W(a \to b) = 1$ for all $a \neq b$, and $W(a \to \epsilon) = W(\epsilon \to a) = 1$ for all a. In *Hamming distance*, $D_H(A, B)$, the only allowed local transformations are of form $a \to b$ where a and b are any members of Σ, with cost $W(a \to a) = 0$ and $W(a \to b) = 1$, for $a \neq b$. Finally, the *indel distance*, $D_{ID}(A, B)$, permits only insertions and deletions. That is, the allowed transformations are $a \to a$, $a \to \epsilon$ (deletion), and $\epsilon \to a$ (insertion), where $a \in \Sigma$, with costs $W(a \to a) = 0$ and $W(a \to \epsilon) = W(\epsilon \to a) = 1$, for all a.

It is well known [6] that the straightforward computation of these distances is carried out by evaluating an appropriate recurrence relation by using *dynamic programming*, where the distances between the prefixes of A and B are tabulated. Each cell d_{ij} of a distance table (d_{ij}) stores the distance between $a_1, \ldots, a_i$ and $b_1, \ldots, b_j$ $(0 \leq i \leq m,\ 0 \leq j \leq n)$ and (d_{ij}) is evaluated by proceeding row-by-row or column-by-column using the given recurrence. For instance, the following recurrence corresponds to $D_{ID}(A, B)$:

$$\begin{aligned} d_{i,0} &= i;\ d_{0,j} = j; \\ d_{ij} &= \textbf{if}\ \ a_i = b_j\ \ \textbf{then}\ \ d_{i-1,j-1}\ \ \textbf{else}\ \ \min(d_{i-1,j} + 1, d_{i,j-1} + 1). \end{aligned}$$

Finally, $d_{m,n}$ gives the distance, in this case $D_{ID}(A, B)$. The framework is

straightforwardly adapted to the problem of searching for occurrences of P in T: The first row of the table (d_{ij}) is initialized with zero values ($d_{0,j} = 0$ for $0 \leq j \leq n$) and instead of observing just the value of the bottom-right corner $d_{m,n}$, any value $d_{m,j}$ not exceeding a given threshold k indicates an approximate occurrence of P ending at position j in T.

Naturally, we can use non unit-cost distances as well. For instance, the following recurrence uses weighted edit distance that makes a distinction according to the amount of the local distortion, as advocated in the Introduction:

$$\begin{array}{lcl} ED_{i,0} & = & i \times \lambda; \;\; ED_{0,j} \;\; = \;\; j \times \lambda \\ ED_{i,j} & = & \min(|a_i - b_j| + ED_{i-1,j-1}, \lambda + ED_{i-1,j}, \lambda + ED_{i,j-1}). \end{array} \quad (1)$$

Here λ is an application-dependent constant used to weight indel operations.

The dual case of $D_{ID}(A, B)$ is the calculation of the *longest common subsequence* of two strings A and B, or $lcs(A, B)$ for short. The length of $lcs(A, B)$, denoted by $LCS(A, B)$, is computed by the recurrence:

$$\begin{array}{lcl} LCS_{i,0} & = & 0; \;\; LCS_{0,j} \;\; = \;\; 0; \\ LCS_{i,j} & = & \textbf{if} \;\; a_i = b_j \;\; \textbf{then} \;\; 1 + LCS_{i-1,j-1} \\ & & \textbf{else} \;\; \max(LCS_{i-1,j}, LCS_{i,j-1}). \end{array} \quad (2)$$

so that $LCS(A, B) = LCS_{|A|,|B|}$.

The well-known relation between $LCS(A, B)$ and $D_{ID}(A, B)$ is as follows (see for example [5,6]): $LCS(A, B) = \frac{|A|+|B|-D_{ID}(A,B)}{2}$.

2.1 Problems under Consideration

Let us now define the required concepts within the framework given above. Given an integer alphabet $\Sigma = \{0 \ldots \sigma\}$, the following recurrence calculates $LCS(A, B)$ for any given transposition c, where $-\sigma \leq c \leq \sigma$:

$$\begin{array}{lcl} LCS^c_{i,0} & = & 0; \;\; LCS^c_{0,j} \;\; = \;\; 0; \\ LCS^c_{i,j} & = & \textbf{if} \;\; a_i + c = b_j \;\; \textbf{then} \;\; 1 + LCS^c_{i-1,j-1} \\ & & \textbf{else} \;\; \max(LCS^c_{i-1,j}, LCS^c_{i,j-1}). \end{array} \quad (3)$$

The calculation of δ-$LCS^c(A, B)$ is analogous to that of Recurrence (3), but instead of observing whether $a_i + c = b_j$ holds, one should observe the truth of the relation $b_j - \delta \leq a_i + c \leq b_j + \delta$.

Definition 1 (LCTS) *Let A and B be strings over an integer alphabet $\Sigma = \{0 \dots \sigma\}$. The* length *of the* longest common transposition invariant subsequence *of A and B, denoted $LCTS(A, B)$, is:*

$$LCTS(A, B) = \max_{c \in \{-\sigma \dots \sigma\}} LCS^c(A, B).$$

Analogously to Definition 1, one may also define the *length of the longest common transposition invariant δ-matching subsequence* as follows:

$$\delta\text{-}LCTS(A, B) = \max_{c \in \{-\sigma \dots \sigma\}} \delta\text{-}LCS^c(A, B).$$

The naive computation of $LCTS$ and its variants requires $O(\sigma|A||B|)$ time, as we have to compute the LCS^c matrix for every transposition c.

As stated above, the string matching framework can be adapted to the text searching problem. Let k be the given error threshold value and P be the pattern to be searched for in polyphonic text $T^g = t_1^g, \dots, t_n^g, \quad g \in \{1 \dots h\}$. P has a c-transposed, k-approximate indel-occurrence in T ending at position j, if in the recurrence

$$\begin{array}{rcl} M_{i,0}^c & = & i; \;\; M_{0,j}^c \;\; = \;\; 0; \\ M_{ij}^c & = & \textbf{if} \;\; p_i + c \in \{t_j^g, \; 1 \le g \le h\} \;\; \textbf{then} \;\; M_{i-1,j-1}^c \\ & & \textbf{else} \;\; \min(M_{i-1,j}^c + 1, M_{i,j-1}^c + 1), \end{array} \tag{4}$$

it holds that $M_{m,j}^c \le k$ where $1 \le j \le n$.

Matching pattern P against polyphinc text T is known as multi-track string matching [12]. The naive solution to this search problem takes $O(h\sigma mn)$ time.

Definition 2 (TIMTKI-occurrence) *Let P be a pattern string to be matched against a polyphonic (multi-track) text string T, both of which are sequences over the integer alphabet $\Sigma = \{0 \dots \sigma\}$. P is said to have a* transposition-invariant multi-track k-approximate indel-occurrence *(TIMKTI-occurrence) in T ending at j, if $M_{m,j}^c \le k$ for some c such that $-\sigma \le c \le \sigma$.*

A δ-matching TIMKTI-occurrence is defined in the obvious way.

3 A Branch and Bound Algorithm

Let X denote a subset of transpositions and $LCS^X(A, B)$ be such that a_i and b_j match whenever $b_j - a_i \in X$. It is easy to see that $LCS^X(A, B) \ge$

$\max_{c \in X} LCS^c(A, B)$: Any common subsequence between $A + c$ and B is considered in the maximum $LCS^X(A, B)$. Hence, $LCS^X(A, B)$ may not be the actual maximum $LCS^c(A, B)$ for $c \in X$, but it gives an upper bound.

Our aim is to find the maximum $LCS^c(A, B)$ value by successive approximations, restricting the subsets X where the optimum c belongs. Our algorithm is inspired in a nearest-neighbor search algorithm for spatial and metric databases [7].

3.1 Binary Hierarchy LCTS

We form a binary tree whose nodes have the form $[\tau, \tau']$ and represent the range of transpositions $X = \{\tau \dots \tau'\}$. The root is $[-\sigma, \sigma]$. The leaves have the form $[c, c]$. Every internal node $[\tau, \tau']$ has two children $[\tau, \lfloor(\tau + \tau')/2\rfloor]$ and $[\lfloor(\tau + \tau')/2\rfloor + 1, \tau']$.

The hierarchy is used to upper bound the $LCS^c(A, B)$ values. For every node $[\tau, \tau']$ of the tree, if we compute $LCS^{[\tau,\tau']}(A, B)$, the result is an upper bound to $LCS^c(A, B)$ for any $\tau \le c \le \tau'$. Moreover, $LCS^X(A, B)$ is easily computed in $O(|A||B|)$ time if $X = \{\tau \dots \tau'\}$ is a continuous range of values:

$$\begin{aligned} LCS_{i,0}^{[\tau,\tau']} &= 0; \;\; LCS_{0,j}^{[\tau,\tau']} = 0; \\ LCS_{i,j}^{[\tau,\tau']} &= \textbf{if} \;\; \tau \le b_j - a_i \le \tau' \;\; \textbf{then} \;\; 1 + LCS_{i-1,j-1}^{[\tau,\tau']} \\ &\quad\;\; \textbf{else} \;\; \max(LCS_{i-1,j}^{[\tau,\tau']}, LCS_{i,j-1}^{[\tau,\tau']}). \end{aligned} \tag{5}$$

We already know that the LCS value of the root is $\min(|A|, |B|)$, since every pair of characters match. The idea is now to compute its two children, and continue with the most promising one (higher LCS^X upper bound). For this most promising one, we compute its two children, and so on. At any moment, we have a set of subtrees to consider, each one with its own upper bound on the leaves it contains. That set of subtrees to be considered is maintained in a max-priority queue. At every step of the algorithm, we take the most promising subtree, compute its two children, and add them to the set of subtrees under consideration. If the most promising subtree turns out to be a leaf node $[c, c]$, then the upper bound value is indeed the exact LCS^c value. At this point we can stop the process, because all the upper bounds of the remaining subtrees are smaller than or equal to the actual LCS^c value we have obtained. So we are sure of having obtained the highest value.

Fig. 1 gives an example for two strings A, B of lengths $|A| = |B| = 20$ when $\sigma = 50$. With a naive algorithm we would need to compute 101 $O(mn)$ tables.

For our example, only 24 such tables are computed. Each node represents the $LCS^X(A,B)$ for some interval X. For example, the root has value 20 because $LCS^{[-50,50]} = 20$. We start with the computation of $LCS^{[-50,0]} = 14$ and $LCS^{[1,50]} = 14$. They have the same value so we pick up either of the two. We choose $LCS^{[-50,0]}$ (note that in Fig. 1 the numbers next to the node give the processing order of the algorithm) and compute $LCS^{[-50,-25]} = 6$ and $LCS^{[-24,0]} = 14$. Now we need to pick up the node with the highest value among those not already considered, in this case $LCS^{[1,50]}$. Recall that this process is implemented by using a max-priority queue. We keep repeating this procedure until we reach a leaf. In this example we stop at leaf $LCS^{[-3,-3]} = LCS^{-3} = 8$ and we can be sure that transposition -3 gives the best alignment. The correctness of our algorithm can be verified by observing that all remaining nodes (nodes without a cross) have values of 8 at most.

A	B
11	8
30	24
21	46
12	44
13	16
44	39
33	12
34	18
39	10
21	41
6	23
1	42
36	17
2	3
47	44
45	20
23	41
40	35
24	7
17	37

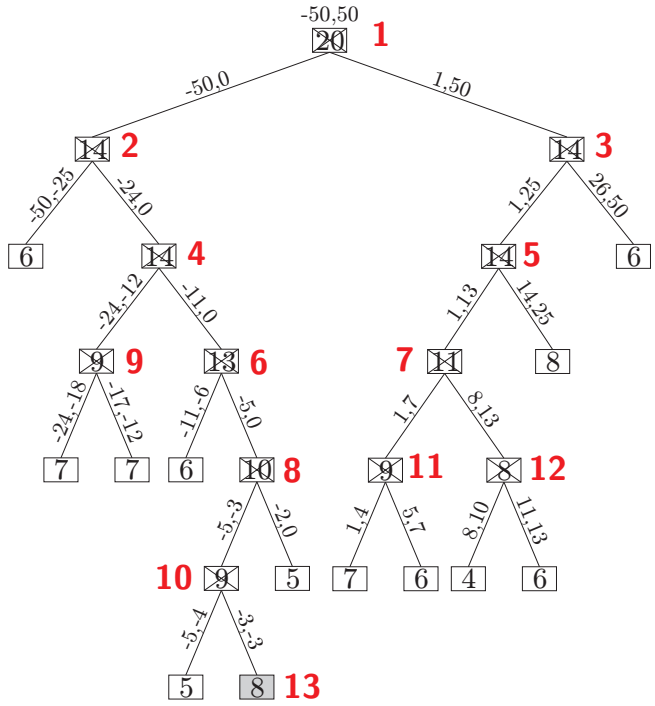


Fig. 1. Example of the *BinaryHierarchyLCTS* computation for $|A| = |B| = 20$ and $\sigma = 50$. The nodes with a cross were considered during the computation.

Fig. 2 shows the algorithm. Our priority queue stores pairs of the form $([\tau, \tau'], val)$ and permits extracting the elements in decreasing val order with *ExtractMax*. It is initialized empty and we insert new elements using *Insert*.

Analysis. We have a best case of $\log(2\sigma + 1) = O(\log \sigma)$ iterations and a worst case of $2(2\sigma + 1) - 1 = 4\sigma + 1 = O(\sigma)$ until we obtain the first leaf element. Our priority queue, which performs operations in logarithmic time, contains $O(\log \sigma)$ elements in the best case and $O(\sigma)$ in the worst case. Hence, recalling that $|A| = m$ and $|B| = n$, $m \leq n$, every iteration of the algorithm

BinaryHierarchyLCTS $(A,\ B,\ \sigma)$

1. **Init**(Q)
2. $[\tau, \tau'] \leftarrow [-\sigma, \sigma]$
3. **While** $\tau \neq \tau'$ **Do**
4. $\quad \theta \leftarrow \lfloor (\tau + \tau')/2 \rfloor$
5. $\quad$ **Insert**$(Q, ([\tau, \theta], \textbf{ComputeSetLCS}(A, B, \tau, \theta)))$
6. $\quad$ **Insert**$(Q, ([\theta + 1, \tau'], \textbf{ComputeSetLCS}(A, B, \theta + 1, \tau')))$
7. $\quad ([\tau, \tau'], lcts) \leftarrow$ **ExtractMax**(Q)
8. **Return** $lcts$

Fig. 2. Branch and bound algorithm to compute $LCTS(A, B)$. *ComputeSetLCS* (A, B, τ, τ') computes $LCS^{[\tau,\tau']}(A, B)$ as described in Recurrence (5).

takes $O(mn + \log\log\sigma)$ at best and $O(mn + \log\sigma)$ at worst. This gives an overall best case complexity of $O((mn+\log\log\sigma)\log\sigma)$ and $O((mn+\log\sigma)\sigma)$ for the worst case. The worst case is as bad as the naive algorithm (but not worse) for $mn = \Omega(\log\sigma)$, which is the case in practice.

In any case, notice that the cost of our algorithm is $O(mnf(\sigma))$. This is favorable, for large mn, compared to alternative algorithms such as the $O(mn\log m)$ one [14], which is independent of σ but whose cost grows faster than $O(mn)$. In Section 7 we show experimentally that our algorithm is better for large $m \geq 150$.

3.2 Higher Arities

Naturally, the branch and bound technique is directly applicable to higher arities as well. Instead of using a binary hierarchy tree, we use a κ-ary tree, for some integer $\kappa > 2$. In this case, every tree node works $O(\kappa mn)$ time to produce κ children that are inserted in the priority queue. On one hand, this increases the processing cost per tree node. On the other, it reduces the tree depth and it might find the right interval faster.

In Section 4.2 we will consider combining the branch and bound algorithm with a bit-parallel algorithm that can perform several LCTS computations in parallel. In that case it becomes natural to adjust the branching factor κ to how many LCTS calculations can be carried out in parallel, so that processing each internal node will cost $O(mn)$.

Analysis. We follow the binary case: The tree has depth $O(\log_\kappa \sigma)$ but processing each internal node costs $O(\kappa mn)$. Since processing an internal node produces κ children, we can consider that generating each tree node (when processing its parent) costs $O(mn)$, including the leaves.

In the best case we follow a single root to leaf path, generating $O(\kappa \log_\kappa \sigma)$ nodes that are also inserted in the priority queue. The total cost is $O((mn + \log(\kappa \log_\kappa \sigma))\kappa \log_\kappa \sigma)$. This is worse than in the binary case $\kappa = 2$. In the worst case we traverse all the $\sigma\kappa/(\kappa - 1)$ tree nodes. The total cost is thus $O((mn + \log(\sigma\kappa/(\kappa - 1)))\sigma\kappa/(\kappa - 1))$. This improves as κ grows.

Overall, it is not clear which is the best κ value, so we determine it experimentally in Section 7. We find that low κ values such as 3 and 4 are as good as $\kappa = 2$, but never significantly better.

4 Speeding Up with Bit-Parallelism

In this section we show how *bit-parallelism* can be used to speed up the computation of the LCTS between strings A and B. Bit-parallelism is a technique to pack several values in a single computer word and to manage to update them all simultaneously, hence speeding up the computations of an algorithm. We will first apply bit-parallelism to speed up the naive LCTS computation, and later to the branch and bound technique.

We will use the following notation to describe bit-parallel algorithms. The number of bits in the computer word will be denoted by w (typically $w = 32$ or 64). In general we will manipulate *bit masks*, which are sequences of bits of length up to w. The bitwise *and* operation between bit masks M_1 and M_2 will be denoted "M_1 & M_2", the bitwise *or* as "$M_1 \mid M_2$", and the bit complementation as "$\sim M_1$". By "$M_1 << i$" we denote the operation of shifting all bits of M_1 to the left by i positions, where the bits that fall outside the bit mask are discarded and the new bits that enter are zero. Similarly, "$M_1 >> i$" shifts the bits to the right. We can perform arithmetic operations, such as addition, subtraction and multiplication, over the bit masks, thus treating them as numbers. We can also compare their numerical values. When carrying out those operations, remind that the most significant bit is at the left. We use exponentiation to denote repetition of bits, such as $0^3 1 = 0001$. Also, we write $[x]_\ell$ to denote the integer x represented in ℓ bits (with $x < 2^\ell$).

4.1 Speeding Up the Brute-Force Algorithm

The simplest technique to compute $LCTS(A, B)$ is to compute $LCS^c(A, B)$ for all c, and choose the maximum. This requires a triple iteration to compute $LCS^c_{i,j}$ for every $i \in 0 \ldots |A|$, $j \in 0 \ldots |B|$, and $c \in -\sigma \ldots \sigma$, which takes $O(\sigma mn)$ time. Our idea is to compute $LCS^c(A, B)$ for several c values simultaneously, in principle iterating only over i and j. Fig. 3 illustrates.

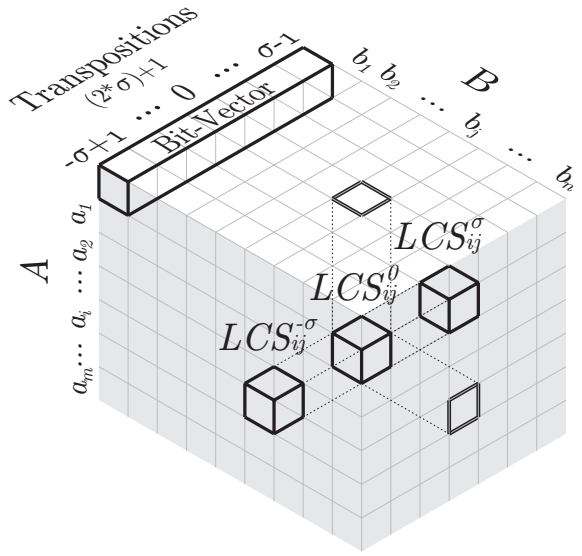


Fig. 3. General scheme to compute $LCTS(A, B)$ using bit-parallelism. We work individually on every (a_i, b_j) pair, but solve several transpositions simultaneously.

The first question is how many c values can we compute in parallel, that is, how many $LCS_{i,j}^c$ numbers can we store in a computer word of w bits. Since all $LCS_{i,j}^c$ values are in the range $\{0 \ldots \min(|A|, |B|)\}$, we need $\ell = \lceil \log(\min(|A|, |B|) + 1) \rceil$ bits to store each value. For reasons that will be made clear soon, we will in fact need $\ell + 1$ bits per value, and hence we will be able to store $\kappa = \lfloor w/(\ell + 1) \rfloor$ values in a single computer word. All our bit masks will be of length $\kappa(\ell + 1) \leq w$. Our bit masks will also be seen as sequences of κ *fields*. The r-th field is formed by the bits $(r-1)(\ell+1)+1 \ldots r(\ell+1)$.

This means that we can compute for κ values of c simultaneously. Therefore, we divide the process of computing $LCS^c(A, B)$ for every $c \in -\sigma \ldots \sigma$ into $\lceil (2\sigma+1)/\kappa \rceil$ separate bit-parallel computations, each for κ contiguous c values. From now on, let us focus on the bit-parallel computation of $LCS^c(A, B)$ for one such contiguous range, $c \in \{\tau \ldots \tau + \kappa - 1\}$ for some τ. We will compute bit masks $LCTS_{i,j}$, for $0 \leq i \leq |A|$ and $0 \leq j \leq |B|$, holding all the $LCS_{i,j}^c$ values in the current c range. That is,

$$LCTS_{i,j} \quad = \quad 0[LCS_{i,j}^{\tau+\kappa-1}]_\ell \; 0[LCS_{i,j}^{\tau+\kappa-2}]_\ell \; \ldots \; 0[LCS_{i,j}^{\tau+1}]_\ell \; 0[LCS_{i,j}^{\tau}]_\ell \; .$$

Eq. (3) has an if-then-else structure. For a given c, if $a_i + c = b_j$, then we have to use value $1 + LCS_{i-1,j-1}^c$, otherwise we have to use $\max(LCS_{i,j-1}^c, LCS_{i-1,j}^c)$. Note that the distinguished value $c = b_j - a_i$ may or may not be in our range $\tau \ldots \tau + \kappa - 1$. To simulate this if-then-else in the bit-parallel computation of $LCTS_{i,j}$, we build a bit mask E holding κ fields of $\ell+1$ bits each, corresponding to the c values in the current range. Those fields in E usually contain $0^{\ell+1}$, except for the one corresponding to $c = b_j - a_i$ (if present), which contains $1^{\ell+1}$. The definition of E follows. The way to use it will be made clear soon.

$$E \;=\; \textbf{if } \tau \le b_j - a_i < \tau + \kappa \qquad (6)$$
$$\textbf{then } 0^{(\tau+\kappa-1-(b_j-a_i))(\ell+1)}\ 1^{(\ell+1)}\ 0^{(b_j-a_i-\tau)(\ell+1)}$$
$$\textbf{else } 0^{\kappa(\ell+1)}\ .$$

Now, we need to build two bit masks corresponding to the two choices to assign value to $LCS^c_{i,j}$. The first corresponds to $1 + LCS^c_{i,j}$. Its bit-parallel version is easy to compute:

$$LCTS_{i,j} + (0^\ell 1)^\kappa \;=\; 0[LCS^{\tau+\kappa-1}_{i,j} + 1]_\ell\ \ldots\ 0[LCS^{\tau}_{i,j} + 1]_\ell\ .$$

The second choice corresponds to value $\max(LCS^c_{i,j-1}, LCS^c_{i-1,j})$. Its bit-parallel version is $Max(LCTS_{i,j-1}, LCTS_{i-1,j})$, where $Max()$ is defined as:

$$Max(0[x^\kappa]_\ell \ldots 0[x^1]_\ell, 0[y^\kappa]_\ell \ldots 0[y^1]_\ell)$$
$$= \; 0[\max(x^\kappa, y^\kappa)]_\ell\ \ldots\ 0[\max(x^1, y^1)]_\ell\ ,$$

that is, $Max()$ takes the field-wise maxima of the two bit masks. Given function $Max()$, value $LCTS_{i,j}$ is computed as

$$LCTS_{i,j} \;=\; (E\ \&\ (LCTS_{i-1,j-1} + (0^\ell 1)^\kappa))$$
$$\mid\ (\sim E\ \&\ Max(LCTS_{i,j-1}, LCTS_{i-1,j}))\ .$$

To see that the above formula is correct, consider its r-th field, corresponding to $LCS^c_{i,j}$ for $c = \tau + r - 1$. If $c = b_j - a_i$, then $r = c - \tau + 1 = (b_j - a_i) - \tau + 1$, and the r-th field in E is 1^ℓ (see Eq. (6)). Thus, the above formula correctly assigns value $LCS^c_{i-1,j-1} + 1$ to $LCS^c_{i,j}$. If, on the other hand, $c \neq b_j - a_i$, then the r-th field in E is $0^{\ell+1}$. Thus, the above formula correctly assigns value $\max(LCS^c_{i,j-1}, LCS^c_{i-1,j})$ to $LCS^c_{i,j}$.

The only missing piece is the computation of $Max(X, Y)$. This is where we need the $(\ell+1)$-th (highest) bit of the fields, always in zero. This solution is from [18] and we repeat it here for completeness. A bit mask $J = (10^\ell)^\kappa$ will be precomputed. Then, to compute $Max(X, Y)$, start with $F \leftarrow ((X \mid J) - Y)\ \&\ J$. Note that the r-th field of $X \mid J$ is $1[x^r]_\ell$, which is always larger than $0[y^r]_\ell$, so the subtraction never overflows from one field to the next. Now, if $x^r \ge y^r$, then $1[x^r]_\ell - 0[y^r]_\ell = 1[x^r - y^r]_\ell$, otherwise $1[x^r]_\ell - 0[y^r]_\ell = 0[2^\ell + x^r - y^r]_\ell$. If we *and* the result of $(X \mid J) - Y$ with J, only the highest bits of the fields survive. That is, the r-th field of F is 10^ℓ if $x^r \ge y^r$, otherwise it is 00^ℓ. We now compute $F \leftarrow F - (F >> \ell)$, so that the r-th bit of F will be $10^\ell - 0^\ell 1 = 01^\ell$ if $x^r \ge y^r$, and $00^\ell - 00^\ell = 00^\ell$ otherwise. At this point, F plays the role of our E mask for the condition "$x^r \ge y^r$". Therefore, it is clear that $Max(X, Y) = (F\ \&\ X) \mid (\sim F\ \&\ Y)$. Also, we easily obtain $Min(X, Y) = (F\ \&\ Y) \mid (\sim F\ \&\ X)$ in the same manner. Fig. 4 gives the code, and Fig. 5 an example.

Max $(X,\ Y,\ \ell,\ \kappa)$
1. $J \leftarrow (10^\ell)^\kappa$
2. $F \leftarrow ((X \mid J) - Y)\ \&\ J$
3. $F \leftarrow F - (F >> \ell)$
4. **Return** $(F\ \&\ X) \mid (\sim F\ \&\ Y)$

Min $(X,\ Y,\ \ell,\ \kappa)$
1. $J \leftarrow (10^\ell)^\kappa$
2. $F \leftarrow ((X \mid J) - Y)\ \&\ J$
3. $F \leftarrow F - (F >> \ell)$
4. **Return** $(F\ \&\ Y) \mid (\sim F\ \&\ X)$

Fig. 4. Bit-parallel computation of field-wise maximum and minimum. J is actually precomputed.

$$
\begin{array}{rcccc}
X & = & 0\ 000 & 0\ 001 & 0\ 010 \\
Y & = & 0\ 001 & 0\ 001 & 0\ 001 \\
J & = & 1\ 000 & 1\ 000 & 1\ 000 \\
X \mid J & = & 1\ 000 & 1\ 001 & 1\ 010 \\
(X \mid J) - Y & = & 0\ 111 & 1\ 000 & 1\ 001 \\
((X \mid J) - Y)\ \&\ J & = & 0\ 000 & 1\ 000 & 1\ 000 \ \rightarrow\ F \\
F >> 3 & = & 0\ 000 & 0\ 001 & 0\ 001 \\
F - (F >> 3) & = & 0\ 000 & 0\ 111 & 0\ 111 \ \rightarrow\ F \\
\sim F & = & 1\ 111 & 1\ 000 & 1\ 000 \\
F\ \&\ X & = & 0\ 000 & 0\ 001 & 0\ 010 \\
\sim F\ \&\ Y & = & 0\ 001 & 0\ 000 & 0\ 000 \\
(F\ \&\ X) \mid (\sim F\ \&\ Y) & = & 0\ 001 & 0\ 001 & 0\ 010 \ \rightarrow\ Max()
\end{array}
$$

Fig. 5. Example of the computation of $Max([0][1][2],[1][1][1]) = [1][1][2]$ with $\ell = 3$.

Fig. 6 shows *RangeLCTS*, the LCTS algorithm for a range of counters $\tau \ldots \tau + \kappa - 1$. We have done some optimizations to the conceptual formulas exposed above.

Using *RangeLCTS*, algorithm *BitParallelLCTS* (also in Fig. 6) traverses all the $c \in -\sigma \ldots \sigma$ transpositions and computes $LCTS(A,B)$ as the maximum $LCTS^c(A,B)$. For this last maximization, the resulting LCTS is stored in a bit mask V, whose fields are examined one by one to find the maximum LCS^c.

Fig. 7 shows a partial example of the computation of $LCTS(2\ 3,\ 2\ 1\ 2\ 3)$, considering transpositions $\tau \ldots \tau + \kappa - 1 = -1 \ldots 1$, with $\ell = 3$.

It is possible to adapt this algorithm to compute δ-$LCTS(A,B)$, where we assume that two characters match if their difference does not exceed δ. This is arranged at no extra cost by considering that there is a match whenever $b_j - a_i - \delta \le c \le b_j - a_i + \delta$. The only change needed in our algorithm is in lines 6–7 of *RangeLCTS*, which should become:

$low \leftarrow \max(\tau, b_j - a_i - \delta)$
$high \leftarrow \min(\tau + \kappa - 1, b_j - a_i + \delta)$
If $low \le high$ **Then**
$E \leftarrow 0^{(\tau+\kappa-1-high)(\ell+1)}\ 1^{(high-low+1)(\ell\ +1)} 0^{(low-\tau)(\ell+1)}$

RangeLCTS $(A,\ B,\ \tau,\ \kappa,\ \ell)$

1. **For** $i \in 0 \ldots |A|$ **Do**
2. **For** $j \in 0 \ldots |B|$ **Do**
3. **If** $i = 0\ \vee\ j = 0$ **Then** $LCTS_{i,j} \leftarrow 0^{\kappa(\ell+1)}$
4. **Else**
5. $M \leftarrow \mathbf{Max}(LCTS_{i-1,j}, LCTS_{i,j-1}, \ell, \kappa)$
6. **If** $\tau \leq b_j - a_i < \tau + \kappa$ **Then**
7. $E \leftarrow 0^{(\tau+\kappa-1-(b_j-a_i))(\ell+1)}\ 1^{(\ell+1)}\ 0^{(b_j-a_i-\tau)(\ell+1)}$
8. $LCTS_{i,j} \leftarrow (E\ \&\ (LCS_{i-1,j-1} + (0^\ell 1)^\kappa))\ |\ (\sim E\ \&\ M)$
9. **Else** $LCTS_{i,j} \leftarrow M$
10. **Return** $LCTS_{|A|,|B|}$

BitParallelLCTS $(A,\ B,\ \sigma)$

1. $\ell \leftarrow \lceil \log(\min(|A|, |B|) + 1) \rceil$
2. $\kappa \leftarrow \lfloor w/(\ell+1) \rfloor$
3. $\tau \leftarrow -\sigma$
4. $lcts \leftarrow 0$
5. **While** $\tau \leq \sigma$ **Do**
6. $V \leftarrow \mathbf{RangeLCTS}(A, B, \tau, \kappa, \ell)$
7. **For** $c \in \tau \ldots \min(\tau + \kappa - 1, \sigma - 1)$ **Do**
8. $lcts \leftarrow \max(lcts, V\ \&\ 0^{(\kappa-1)(\ell+1)}01^\ell)$
9. $V \leftarrow V >> (\ell+1)$
10. $\tau \leftarrow \tau + \kappa$
11. **Return** $lcts$

Fig. 6. Computing $LCTS(A, B)$ using bit-parallelism. All constant bit masks are precomputed.

Analysis. Let us now analyze the algorithm and compare against other alternatives. *BitParallelLCTS* performs $\lceil (2\sigma+1)/\kappa \rceil$ invocations of *RangeLCTS* plus a minimization over $2\sigma + 1$ values. In turn, *RangeLCTS* takes $O(|A||B|)$ time. Since $\kappa = \Theta(w/\log\min(|A|, |B|))$, the time complexity of the algorithm is $O(\sigma|A||B|\log(\min(|A|, |B|))/w)$. If $|A| = m$, $|B| = n$, $m \leq n$, the algorithm is $O(\sigma mn \log(m)/w)$ time, which represents a speedup of $\Theta(w/\log m)$ over the naive $O(\sigma mn)$ time algorithm.

Our complexity is worse than $O(mn \log\log m)$, obtained in [14]. However, in practice, an $O(mn \log m)$ variant presented in the same paper works better for moderate m values. Compared to that variant, we pay $O(\sigma/w)$ more time, so our algorithm should be better with longer computer words and smaller alphabets. Hence the comparison depends on the machine (w) and the application (σ), as well as on the implementation-dependent constants of each algorithm. In Section 7 we compare both algorithms for the MIDI application with $\sigma + 1 = 128$ pitch values, showing that our algorithm wins for small

		b_1	b_2	b_3	b_4
		2	1	2	3
a_1	2	[0][1][0]	[0][1][1]	[0][1][1]	[1][1][1]
a_2	3	[0][1][1]	[0][1][1]	[0][1][2]	[1][2][2]

$\tau = -1,\ \kappa = 3,\ \ell = 3,\ A = 2\ 3,\ B = 2\ 1\ 2\ 3$

$$
\begin{array}{rcllll}
E & = & 0\ 000 & 1\ 111 & 0\ 000 & (c = b_4 - a_2 = 0,\ \tau \le c < \tau + \kappa) \\
LCTS_{1,3} & = & 0\ 000 & 0\ 001 & 0\ 001 & (=\ [0][1][1]) \\
 & + & 0\ 001 & 0\ 001 & 0\ 001 & (+(0^\ell 1)^\kappa) \\
 & = & 0\ 001 & 0\ 010 & 0\ 010 & (=\ [1][2][2]) \\
\&E & = & 0\ 000 & 0\ 010 & 0\ 000 & (v_1) \\
M & = & 0\ 001 & 0\ 001 & 0\ 010 & (Max(LCTS_{2,3}, LCTS_{1,4}) = [1][1][2] \\
 & & & & & \quad = Max([0][1][2], [1][1][1]),\ \text{Fig. 5}) \\
\& \sim E & = & 0\ 001 & 0\ 000 & 0\ 010 & (v_2) \\
LCTS_{2,4} & = & 0\ 001 & 0\ 010 & 0\ 010 & (=\ v_1 \mid v_2\ =\ [1][2][2])
\end{array}
$$

Fig. 7. Example of the computation of $LCTS(2\ 3,\ 2\ 1\ 2\ 3)$ considering transpositions $-1,\ 0,\ 1$ with $\ell = 3$. Bit masks are written $[x][y][z]$ for simplicity, where z corresponds to transposition -1, y to 0 and x for 1. We focus on the computation of the last cell.

$m \le 30$. The algorithm [14] can also be extended to compute δ-$LCTS(A, B)$, but its cost raises to $O(\delta mn \log\log m)$, while ours stays the same.

On the other hand, the $O(mn/w)$ bit-parallel algorithm of [4] can be run $2\sigma+1$ times, for each transposition, to compute $LCTS(A, B)$ in $O(\sigma mn/w)$ time. In this case, our complexity is worse by an $O(\log m)$ factor, and our algorithm is indeed slower for large $m \ge 85$, as seen in Section 7. For smaller m, however, our algorithm is better because the algorithm of [4] performs actually $m\lceil n/w \rceil$ steps, which is larger than mn/w. The algorithm [4] can easily be extended to compute δ-$LCTS(A, B)$ at the same cost.

4.2 Speeding Up the Branch and Bound Algorithm

In Section 3 we have shown how the transposition c yielding the longest $LCS^c(A, B) = LCTS(A, B)$ can be searched for better than by brute force. We considered mostly a binary partition of the space of possible transpositions, where for each transposition range $\tau \ldots \tau'$ we computed an upper bound $LCS^{[\tau,\tau']}(A, B) \ge LCS^c(A, B)$ for any $\tau \le c \le \tau'$. We also considered the possibility of a higher arity tree, with the tradeoff of finer-grained ranges but higher cost to generate them.

By using bit-parallelism, we can generate a κ-ary tree at the cost of a *single* (bit-parallel) LCTS computation per internal tree node processed. Recurrence (5) can be converted into a bit-parallel LCTS computation for all the κ partitions of the current range. That is, if the current tree node corresponds to range $\tau \ldots \tau'$, then we can compute $LCS^{[\tau,\tau+\theta-1]}$, $LCS^{[\tau+\theta,\tau+2\theta-1]}$, $\ldots$, $LCS^{[\tau+(\kappa-1)\theta,\tau']}$, where $\theta = \lceil (\tau' - \tau + 1)/\kappa \rceil$, all in one shot. That is,

$$LCTS_{i,j}^{[\tau,\tau']} \quad = \quad 0[LCS_{i,j}^{[\tau,\tau+\theta-1]}]_\ell \; 0[LCS_{i,j}^{[\tau+\theta,\tau+2\theta-1]}]_\ell \; \ldots \; 0[LCS_{i,j}^{[\tau+(\kappa-1)\theta,\tau']}]_\ell \; .$$

Fig. 8 depicts the computation for our previous binary hierarchy example (Fig. 1). We assume $w = 32$ and $m = 20$, so $\ell = \kappa = 5$. That means that we use a 5-ary tree and that the children of a given node can be computed by using mn operations instead of $5mn$, which represents a speedup of 5 over *BinaryHierarchyLCTS* algorithm. This is why, for this instance, we only perform 6 $O(mn)$ table computations instead of 24 with *BinaryHierarchyLCTS*, 26 with *BitParallelLCTS*, and 101 with the naive algorithm. The algorithm stops at leaf $[-3,-3]$ with $LCS^{-3}(A,B) = 8$. Note that all remaining nodes are valued 8 at most.

A	B
11	8
30	24
21	46
12	44
13	16
44	39
33	12
34	18
39	10
21	41
6	23
1	42
36	17
2	3
47	44
45	20
23	41
40	35
24	7
17	37

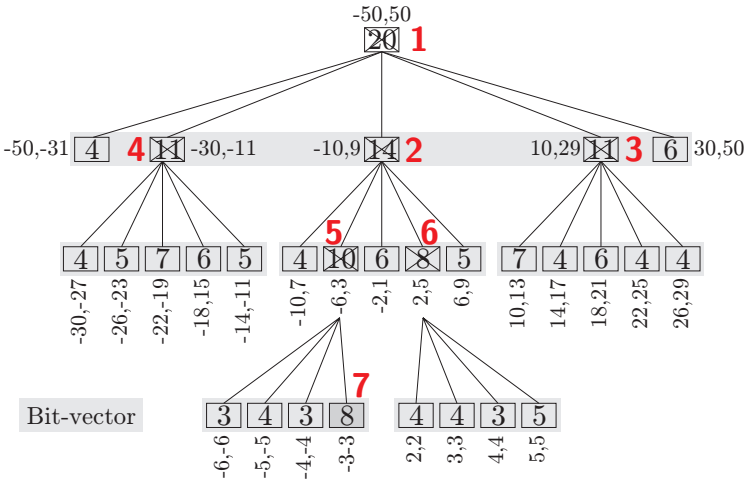


Fig. 8. Example of κ-ary bit-parallel LCTS computation for $|A| = |B| = 20$, $\sigma = 50$, $w = 32$, $\ell = 5$ and $\kappa = 5$. The nodes with a cross were considered during the computation.

For the bit-parallel computation of $LCTS_{i,j}^{[\tau,\tau']}$, we only need to modify the definition of E (Eq. (6)) so that it considers to which interval $b_j - a_i$ belongs. That is, if $\tau \leq b_j - a_i < \tau + \theta$, then it belongs to the first interval; if $\tau + \theta \leq b_j - a_i \leq \tau + 2\theta$, then it belongs to the second interval; and so on. Hence, we have to put 1^ℓ in E at the r-th field, where $r = 1 + \lfloor (b_j - a_i - \tau)/\theta \rfloor$. This is

$$E \quad = \quad \textbf{if } \tau \leq b_j - a_i \leq \tau'$$

then $0^{(\kappa-1-\lfloor(b_j-a_i-\tau)/\theta\rfloor)(\ell+1)}\ 1^{(\ell+1)}\ 0^{\lfloor(b_j-a_i-\tau)/\theta\rfloor(\ell+1)}$
else $0^{\kappa(\ell+1)}$.

Fig. 9 gives the pseudocode of the algorithm. *T-aryNode* computes all the children of an internal tree node in one shot, and returns them in an LCTS bit mask. *T-aryHierarchyLCTS* manages the priority queue of tree nodes and finishes as soon as the first leaf is extracted.

T-aryNode $(A,\ B,\ \tau,\ \tau',\ \kappa,\ \ell,\ \theta)$
For $i \in 0 \ldots |A|$ **Do**
    For $j \in 0 \ldots |B|$ **Do**
        If $i = 0\ \vee\ j = 0$ **Then** $LCTS_{i,j} \leftarrow 0^{\kappa(\ell+1)}$
        Else
            If $\tau \leq b_j - a_i \leq \tau'$ **Then**
                $E \leftarrow 0^{(\kappa-1-\lfloor(b_j-a_i-\tau)/\theta\rfloor)(\ell+1)}\ 1^{(\ell+1)}\ 0^{\lfloor(b_j-a_i-\tau)/\theta\rfloor(\ell+1)}$
            Else $E \leftarrow 0^{\kappa(\ell+1)}$
            $LCTS_{i,j} \leftarrow (E\ \&\ (LCTS_{i-1,j-1} + (0^\ell 1)^\kappa))$
                $\mid\ (\sim E\ \&\ \textbf{Max}(LCTS_{i-1,j}, LCTS_{i,j-1}, \ell, \kappa)$
Return $LCTS_{|A|,|B|}$

T-aryHierarchyLCTS $(A,\ B,\ \sigma)$
Init(Q)
$\ell \leftarrow \lceil \log(\min(|A|,|B|)+1) \rceil$
$\kappa \leftarrow \lfloor w/(\ell+1) \rfloor$
$[\tau,\tau'] \leftarrow [-\sigma,\sigma]$
While $\tau \neq \tau'$ **Do**
    $\theta \leftarrow \lceil (\tau'-\tau+1)/\kappa \rceil$
    $V \leftarrow$ **T-aryNode**$(A,B,\tau,\tau',\kappa,\ell,\theta)$
    For $r \in 0 \ldots \kappa-1$ **Do**
        $v \leftarrow V\ \&\ 0^{(\kappa-1)(\ell+1)}01^\ell$
        If $r = \kappa-1$ **Then** $t \leftarrow \tau'$ **Else** $t \leftarrow \tau + (r+1)\theta - 1$
        Insert$(Q, ([\tau + r\theta, t], v))$
        $V \leftarrow V >> (\ell+1)$
    $([\tau,\tau'], lcts) \leftarrow$ **ExtractMax**(Q)
Return $lcts$

Fig. 9. Bit-Parallel branch and bound algorithm to compute $LCTS(A, B)$.

Analysis. The analysis of the algorithm closely follows that of Section 3.2. The tree is κ-ary but we pay $O(mn)$ instead of $O(\kappa mn)$ to process each node. Our best case turns out to be $O((mn + \kappa\log(\kappa\log_\kappa\sigma))\log_\kappa\sigma)$ and our worst case $O((mn + \kappa\log\sigma)\sigma/(\kappa-1))$, where $\kappa = \Theta(w/\log m)$.

This is obviously better than the result of Section 3.2. As shown in Section 7,

this version is better than the non-bit-parallel approach for small m. However, at that point, the plain bit-parallel algorithm of Section 4.1 is equally good.

5 Text Searching

Up to now we have considered the computation of the longest common subsequence (or its dual, the indel distance) between two sequences. This permits comparing them as a whole and is useful for some applications. In this section we focus on the search problem, where we need to point out the substrings of a long string $T_{1\ldots n}$ (the text) with small distance (at most k) to a short string $P_{1\ldots m}$ (the pattern). Moreover, the long string has in general h tracks $T^1 \ldots T^h$, and of course we aim at transposition invariant matching. The exact formulation of the search problem was given in Section 2.

Let us express Recurrence (4) more operationally. Instead of filling a matrix we will compute one column of it at a time. In order to compute column j we only need column $j-1$. Therefore, our first column is $D^c_{0\ldots m} = M^c_{0\ldots m,0}$ and then, at the j-th step of the algorithm, we compute the new column $D^{c\,\prime}_{0\ldots m} = M^c_{0\ldots m,j}$ by using the current column $D^c_{0\ldots m} = M^c_{0\ldots m,j-1}$. Then $D^{c\,\prime}$ becomes D^c, and if $D^c_m \leq k$ for some c, we report an approximate occurrence of P ending at text position j. Under this notation, Recurrence (4) becomes

$$D^{c\,\prime}_i \;=\; \textbf{if}\;\; P_i + c \in \{T^1_j \ldots T^h_j\}\;\; \textbf{then}\;\; D^c_{i-1}\;\; \textbf{else}\;\; 1 + \min(D^{c\,\prime}_{i-1}, D^c_i), \quad (7)$$

where always $D^c_0 = D^{c\,\prime}_0 = 0$, and we report every text position j such that $\min_{c \in -\sigma \ldots \sigma} D^{c\,\prime}_m \leq k$.

We note that the branch and bound mechanisms developed in Sections 3 and 4.2 cannot be efficiently applied to this scenario. The reason is that Recurrence (4) manages to compute the smallest indel distance between P and $T_{j'\ldots j}$, simultaneously for every j' (in this paragraph we consider a single text for simplicity). That is, $M^c_{i,j} = \min_{j' \leq j} D^c_{ID}(P_{1\ldots i}, T_{j'\ldots j})$. If we are interested in error threshold k, then the relevant j' values are in the range $j-m-k+1 \ldots j-m+k+1$. The branch and bound mechanism, which can only compute indel distance between two strings, would need to perform $2k+1$ computations per text position, namely $D^c_{ID}(P, T_{j'\ldots j})$ for $j-m-k+1 \leq j' \leq j-m+k+1$, in order to find $\min_{c \in -\sigma \ldots \sigma} M^c_{m,j}$. This would render it not competitive. Note that the computations performed for position $j-1$ do not necessarily serve to compute for position j, since the backtracking can go by different branches and need different c ranges than those computed for $j-1$.

On the other hand, the bit-parallel technique we developed in Section 4.1 can be efficiently extended to text searching. We analyze in the sequel the

necessary changes to bit-parallelize Eq. (7) instead of Eq. (3).

First, polyphony, that is, the fact that there are h text tracks $T^1 \dots T^h$, is dealt with by extending the definition of mask E (Eq. (6)) such that it contains $1^{\ell+1}$ in every field corresponding to any transposition in the set $\{T_j^g - P_i,\ 1 \leq g \leq h\}$. To be precise, let us call $E(b_j - a_i)$ the definition of Eq. (6). Then, our E mask is defined as

$$E \quad = \quad E(T_j^1 - P_i) \mid E(T_j^2 - P_i) \mid \ \dots \ \mid E(T_j^h - P_i) \ .$$

Second, we observe that min() is used instead of the max() of Eq. (3), and that the "+1" is at a different place. Both changes are easily addressed.

Additionally, we note that, when a D_i^c value is larger than k, all we need to know is that it is larger than k, so we store $k+1$ for those values in order to represent smaller numbers. Once we achieve this, the number of bits needed by a D_i^c cell is reduced to $\ell = \lceil \log(k+2) \rceil$ and our bit-parallel speedup will increase.

Enforcing the $k+1$ limit is only necessary when we add 1 in the "**else**" clause of Recurrence (7). Since $D_{i-1}^{c\,\prime}$ or D_i^c can already by $k+1$, adding 1 to them overflows to $k+2$. A simple solution is to rewrite the minimization of the "**else**" clause to $1 + \min(D_{i-1}^{c\,\prime}, D_i^c, k)$, which ensures that overflows cannot occur.

To summarize, if we let DT_i be the bit-parallel version of D_i^c, that is

$$DT_i \quad = \quad 0[D_i^{\tau+\kappa-1}]_\ell \ 0[D_i^{\tau+\kappa-2}] \ \dots \ 0[D_i^{\tau+1}]_\ell \ 0[D_i^{\tau}]_\ell \ ,$$

then the recurrence for DT_i is as follows:

$$\begin{aligned} DT_i' \quad &= \quad (E \ \& \ DT_{i-1}) \\ &\quad \mid \ (\sim E \ \& \ ((0^\ell 1)^\kappa + Min(Min(DT_{i-1}', DT_i), (0[k]_\ell)^\kappa))) \ , \end{aligned}$$

and we report the current text position whenever $DT_m' \neq (0[k+1]_\ell)^\kappa$, that is, when some cell at row m is not $k+1$ (hence it is smaller than $k+1$).

Fig. 10 shows *RangeIDSearch*, which searches for a range of transpositions that fit in a computer word. The general algorithm, *IDSearch*, simply applies the former procedure to successive ranges. Note that we do not use two arrays DT and DT', but rather overwrite a single array DT, managing to maintain the previous DT_{i-1} value in $oldD$ and the new DT_i value in $newD$. Observe also the initialization of DT, where we set $D_i^c = i$ for $0 \leq i \leq k$ and $D_i^c = k+1$ otherwise.

Analysis. The algorithm is $O(h\sigma mn \log(k)/w)$ time, which represents a speedup of $O(w/\log k)$ over the classical solution. Note that we could use $\ell = \lceil \log(m+$

RangeIDSearch $(P,\ T^1 \dots T^h,\ k,\ \tau,\ \kappa,\ \ell)$

1. $K \leftarrow (0[k]_\ell)^\kappa$
2. **For** $i \in 0 \dots k$ **Do** $DT_i \leftarrow (0[i]_\ell)^\kappa$
3. **For** $i \in k+1 \dots |P|$ **Do** $DT_i \leftarrow (0[k+1]_\ell)^\kappa$
4. **For** $j \in 1 \dots |T|$ **Do**
5. $\quad oldD \leftarrow 0^{\kappa(\ell+1)}$
6. $\quad$ **For** $i \in 1 \dots |P|$ **Do**
7. $\quad\quad E \leftarrow 0^{\kappa(\ell+1)}$
8. $\quad\quad$ **For** $g \in 1 \dots h$ **Do**
9. $\quad\quad\quad$ **If** $\tau \le T_j^g - P_i < \tau + \kappa$ **Then**
10. $\quad\quad\quad\quad E \leftarrow E \mid 0^{(\tau+\kappa-1-(T_j^g-P_i))(\ell+1)}\ 1^{(\ell+1)}\ 0^{((T_j^g-P_i)-\tau)(\ell+1)}$
11. $\quad\quad newD \leftarrow (E \ \&\ oldD)$
$\quad\quad\quad\quad \mid (\sim E \ \&\ ((0^\ell 1)^\kappa + Min(Min(DT_{i-1}, DT_i, \ell, \kappa), K, \ell, \kappa)))$
12. $\quad\quad oldD \leftarrow DT_i,\ DT_i \leftarrow newD$
13. $\quad$ **If** $newD \neq (0[k+1]_\ell)^\kappa$ **Then** Report an occurrence ending at j

IDSearch $(P,\ T^1 \dots T^h,\ k,\ \sigma)$

1. $\ell \leftarrow \lceil \log(k+2) \rceil$
2. $\kappa \leftarrow \lfloor w/(\ell+1) \rfloor$
3. $\tau \leftarrow -\sigma$
4. **While** $\tau \le \sigma$ **Do**
5. $\quad$ **RangeIDSearch**$(P, T^1 \dots T^h, k, \tau, \kappa, \ell)$
6. $\quad \tau \leftarrow \tau + \kappa$

Fig. 10. Searching polyphonic text with indel distance allowing any transposition. Constant bit masks are precomputed.

$1)\rceil$, thus removing the possibility of overflows in cell values. This reduces the degree of parallelization in exchange of removing one application of Min in line 11 of *RangeIDSearch*.

On the other hand, the algorithms of [14] and [4] can be easily extended to deal with polyphonic text searching, at $O(hmn \log\log m)$ and $O(h\sigma mn/w)$ cost, respectively. Therefore, the complexity comparisons done at the end of Section 4.1, both in theory and in practice, hold for text searching too.

6 More General Distance Functions

The distinguishing feature of our approach, compared to other bit-parallel algorithms used for transposition invariant string matching, is that they apply bit-parallelism along a different dimension of the cube in Fig. 3. Ours is

the only algorithm that packs different transpositions in the bit masks and computes the cells one by one. This gives us extra flexibility, because we can handle complex recurrences among cells as long as we can do several similar operations in parallel, without any dependence between the values computed in the same computer word.

As explained in the Introduction, a weighted edit distance where the cost to convert a note into another is proportional to the absolute difference among the notes is of interest in music retrieval. In this section we demonstrate the flexibility of our approach by addressing the computation of the weighted edit distance detailed in Eq. (1). The only alternative algorithm for this task [1] yields $O(mn\lambda \log(\lambda)/w)$ time.

What follows is the search version for a given transposition c in polyphonic text, bounded by $k+1$ as we did in Section 5.

$$D_i^{c\,\prime} \;=\; \min(\min_{g\in 1\ldots h} |P_i + c - T_j^g| + D_{i-1}^c, \lambda + D_{i-1}^{c\,\prime}, \lambda + D_i^c, k+1). \qquad (8)$$

The main challenge is to compute $|P_i + c - T_j^g|$ in bit-parallel for a set of consecutive c values. For a sequence of transpositions $c = -\sigma,\ -\sigma+1, \ldots$, the values $|P_i+c-T_j^g|$ form a decreasing sequence that reaches zero at $c = T_j^g - P_i$. Thereafter the values start to increase (see Fig. 11). When the transposition range $-\sigma \ldots \sigma$ is split into consecutive ranges $\tau \ldots \tau+\kappa-1$, depending on the range, the values $|P_i + c - T_j^g|$ form a decreasing, increasing, or decreasing-then-increasing sequence.

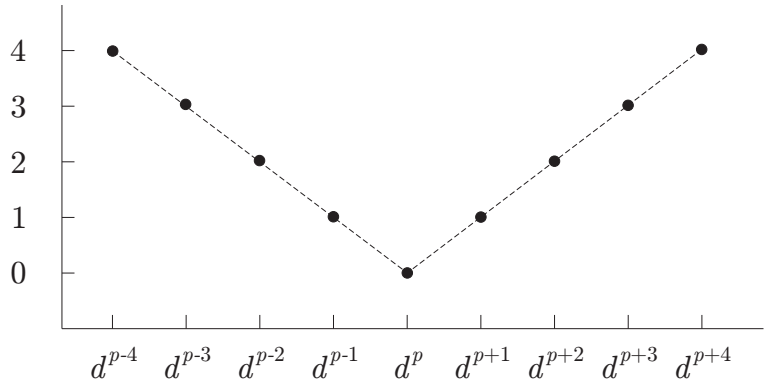


Fig. 11. The values for successive transpositions first decrease (until $c = p = T_j^g - P_i$), then increase.

For brevity, we will use $[x]$ to denote $0[x]_\ell$ in this discussion. An increasing sequence of the form $I_t = [t+\kappa-1]\ldots[t+1]\ [t]$, $t \geq 0$, is obtained simply as $I_t = (0^\ell 1)^{\kappa-1}[t] \times (0^\ell 1)^\kappa$. To see this, view the multiplication as operating two numbers written in base $2^{\ell+1}$, hence each field is a digit. If $Z = X \times Y$, where $X = [x_\kappa]\ldots[x_1]$, $Y = [y_\kappa]\ldots[y_1]$, and $Z = [z_\kappa]\ldots[z_1]$, then $z_r = \sum_{s=1}^{r} x_s \cdot y_{r-s+1}$. In our case, $x_1 = t$, and all others x_i and y_j are 1. Thus

$z_r = t + r - 1$ as desired. We could even accommodate substitution costs of the form $|a_i - b_j|/q$ for integer q by multiplying by $(0^{q(\ell+1)-1}1)^\kappa$ instead of $(0^\ell 1)^\kappa$.

A decreasing sequence $D_t = [t-\kappa+1]\ldots[t-1]\ [t]$ is obtained simply as $D_t = [t]^\kappa - I_0$, as its r-th field will have value $t - r + 1$. Finally, a decreasing-then-increasing sequence $DI_t = [\kappa-t-1]\ [\kappa-t-2]\ldots[2]\ [1]\ [0]\ [1]\ldots[t-1]\ [t]$ is obtained as $DI_t = (I_0 << t(\ell+1)) \mid (D_\kappa >> (\kappa-t)(\ell+1))$, by concatenating an increasing and a decreasing sequence.

A secondary challenge we face is to ensure that we never surpass $k+1$ in the D_i values. This is more difficult than in Section 5 since the increments are not only by 1. We choose to compute the full values and then take the minimum with $k+1$, as suggested by Recurrence (8). However the intermediate values can be larger. We obviously may assume that $\lambda \leq k$, thus the terms corresponding to insertion and deletion are bounded by $2k+1$. We will also keep the first term of Recurrence (8) below $2k+2$. Thus we will need $\lceil \log(2k+3) \rceil$ bits for our counters.

For the latter purpose, we need to ensure that our increasing and/or decreasing sequences are bounded by $k+1$. A version of I_t where all the values are bounded by r is obtained as $I_t^r = (I_t \ \&\ 0^{(\kappa-(r-t))(\ell+1)}1^{(r-t)(\ell+1)}) \mid ([r]^{\kappa-(r-t)}0^{(r-t)(\ell+1)})$. Decreasing sequences are similarly bounded to D_t^r. The bounded version of DI is obtained by using I^r and D^r instead of I and D. Fig. 12 gives the code to build these sequences.

Fig. 13 shows the search algorithm using this general distance function. Most of the comments made for the algorithm of Fig. 10 apply. The main change, apart of course of the initialization in lines 3–4 and the recurrence in line 16, is the form to compute E. Each character T_j^g produces a sequence (increasing, decreasing, or decreasing-then-increasing). In E we take the pointwise minima over those sequences.

Analysis. It is clear that the algorithm runs in $O(h\sigma mn \log(k)/w)$ time. The competing algorithm [1] can be adapted to run in $O(h\sigma mn\lambda \log(\lambda)/w)$ time for this problem. Our complexity is better whenever $\log k = O(\lambda \log \lambda)$, which is the case in practice for short strings.

7 Experiments

We concentrate our experimental study on all the known LCTS variants. Four sets of experiments were carried out. The first experiment aims at determining the best branching factor for our hierarchical algorithm of Section 3. Once this

I $(t,\ r,\ \kappa,\ \ell)$

1. **If** $r \leq t$ **Then Return** $(0[r]_\ell)^\kappa$
2. $I_t \leftarrow ((0^\ell 1)^{\kappa-1}\ 0[t]_\ell) \times (0^\ell 1)^\kappa$
3. **Return** $(I_t\ \&\ 0^{(\kappa-(r-t))(\ell+1)}1^{(r-t)(\ell+1)}) \mid (0[r]_{\ell+1})^{\kappa-(r-t)}0^{(r-t)(\ell+1)}$

D $(t,\ r,\ \kappa,\ \ell)$

1. **If** $r \geq t$ **Then** $r \leftarrow t$
2. $D_r \leftarrow (0[r]_\ell)^\kappa - (0^\ell 1)^{\kappa-1}0^{\ell+1} \times (0^\ell 1)^\kappa$
3. **Return** $(D_r << (t-r)(\ell+1)) \mid 0^{(\kappa-(t-r))(\ell+1)}(0[r]_\ell)^{(t-r)(\ell+1)}$

DI $(t,\ r,\ \kappa,\ \ell)$

1. $I \leftarrow \mathbf{I}(0, r, \kappa-1, \ell)$
2. $D \leftarrow \mathbf{D}(\kappa, r, \kappa, \ell)$
3. **Return** $(I << t(\ell+1)) \mid (D >> (\kappa-t)(\ell+1))$

Fig. 12. Bit-parallel codes for increasing, decreasing, and decreasing-increasing sequences. Constant masks are precomputed. We also precompute all masks that depend on r, since in the main algorithm $r = k+1$ always holds.

is determined, our second experiment compares classical algorithms, that is, those algorithms that do not make use of bit-parallelism. The third experiment shows how the different bit-parallel algorithms perform in practice. In the final experiment we seek to determine the best algorithm overall, both for LCTS computation and for text searching.

The alphabet used was ASCII of size 128, to emulate the MIDI format. Strings of length 20–2500 were randomly generated (we assume $n = m$ for all cases), to account for different cases of interest in music retrieval. Each experiment was repeated 100 times and the median is reported in order to reduce estimator variance. All experiments were conducted on a 900 MHz Pentium machine with 256MB of RAM and $w = 32$ bits. All codes were compiled at the highest optimization level.

We compare eight different algorithms, which are summarized in Table 1. The SDP and YBP code were obtained directly from the authors, while all other codes are our own implementations.

7.1 Classical Algorithms

The first experiment was to try different arities for TBB (see Table 1). As it can be seen in Fig. 14, arities 3 and 4 usually give the best performance,

RangeEDSearch $(P,\ T^1 \ldots T^h,\ k,\ \tau,\ \kappa,\ \ell)$

$Kp1 \leftarrow (0[k+1]_\ell)^\kappa$
$L \leftarrow (0[\lambda]_\ell)^\kappa$
For $i \in 0 \ldots \lfloor k/\lambda \rfloor$ **Do** $D_i \leftarrow (0[i \cdot \lambda]_\ell)^\kappa$
For $i \in \lfloor k/\lambda \rfloor + 1 \ldots |P|$ **Do** $D_i \leftarrow Kp1$
For $j \in 1 \ldots |T|$ **Do**
    $oldD \leftarrow 0$
    For $i \in 1 \ldots |P|$ **Do**
        $E \leftarrow Kp1$
        For $g \in 1 \ldots h$ **Do**
            If $T_j^g - P_i \leq \tau$ **Then** // Increasing sequence
                $E' \leftarrow \mathbf{I}(\tau - (T_j^g - P_i), k+1, \kappa, \ell)$
            Else If $T_j^g - P_i \geq \tau + \kappa$ **Then** // Decreasing sequence
                $E' \leftarrow \mathbf{D}((T_j^g - P_i) - \tau, k+1, \kappa, \ell)$
            Else $E' \leftarrow \mathbf{DI}((T_j^g - P_i) - \tau, k+1, \kappa, \ell)$
            $E \leftarrow Min(E, E', \ell, \kappa)$
        $newD \leftarrow Min(Min(E + oldD, Min(D_{i-1}, D_i, \ell, \kappa) + L, \ell, \kappa),$
                $Kp1, \ell, \kappa)$
        $oldD \leftarrow D_i,\ D_i \leftarrow newD$
    If $newD \neq Kp1$ **Then** Report an occurrence ending at j

EDSearch $(P,\ T^1 \ldots T^h,\ k,\ \sigma)$

$\ell \leftarrow \lceil \log(2k+3) \rceil$
$\kappa \leftarrow \lfloor w/(\ell+1) \rfloor$
$c \leftarrow -\sigma$
While $c \leq \sigma$ **Do**
    RangeEDSearch$(P, T^1 \ldots T^h, k, c, \kappa, \ell)$
    $c \leftarrow c + \kappa$

Fig. 13. Searching polyphonic text with weighted edit distance allowing any transposition. Constant masks are precomputed.

arity 3 being never much worse than the optimum. So for the rest of the experiments we use 3BB as the representative of the TBB family. We note that this conclusion probably depends on the type of text considered, so it could change for non-random text.

We observed that, in general, performance measures for hierarchical algorithms suffer from considerable variance. To get reliable results we took the measurement for 100 different inputs and used the median rather than the mean.

The experiment comparing different classical algorithms to compute LCTS was run for codes 1–4 (see Table 1). The results are shown in Fig. 15-A. For small sizes the best algorithm is SPD, but for larger sizes (more than 150) our

No.	Algorithm	Category	Search?	ShortName	Source
1	Classical Dynamic Programming	Classical	Yes	CDP	[13]
2	Sparse Dynamic Programming	Classical	Yes	SDP	[14]
3	Binary Branch and Bound	Classical	No	BBB	Sec 3.1
4	T-ary Branch and Bound	Classical	No	TBB	Sec 3.2
5	Bit-Parallel in Y-Dimension	Bit-Parallel	Yes	YBP	[4]
6	Bit-Parallel in Z-Dimension	Bit-Parallel	Yes	ZBP	Sec 4.1
7	3+5	Bit-Parallel	No	BBBYBP	Sec 3+[4]
8	4+6	Bit-Parallel	No	TBBZBP	Sec 4.2

Table 1
Summary of the codes used for the experiments.

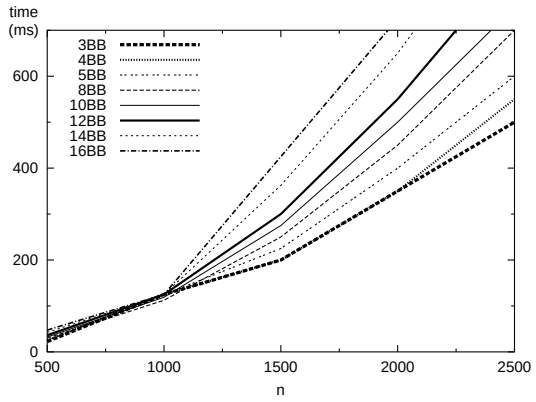
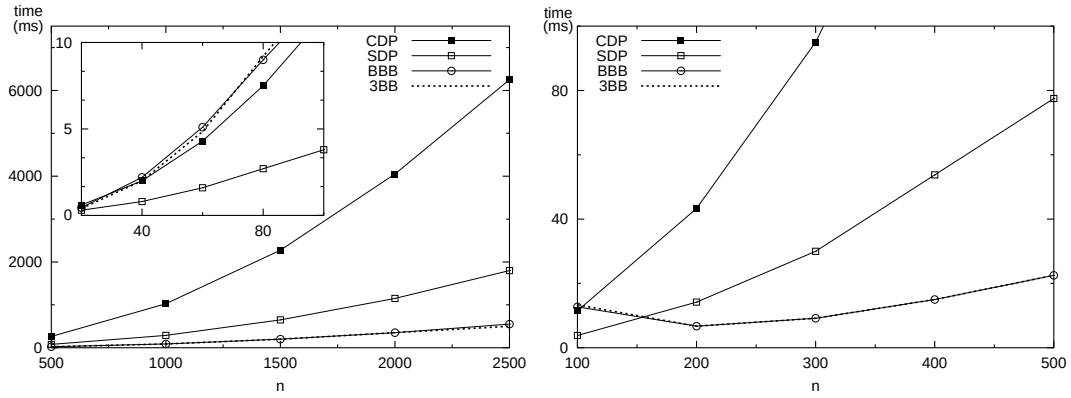


Fig. 14. Finding best arity for TBB.

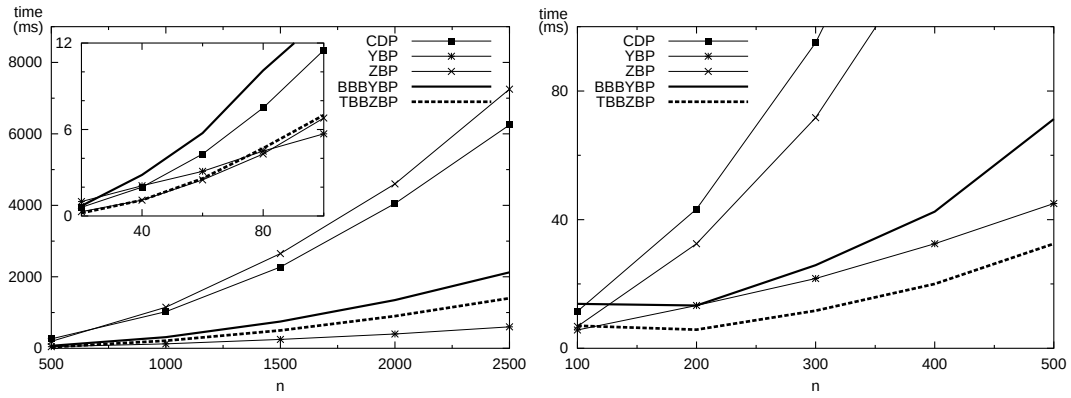
hierarchical algorithms BBB (binary hierarchy) and 3BB (ternary hierarchy) were faster.

This coincides with the algorithm complexities. SPD is $O(mn \log m)$ time, while BBB and TBB are $O(mnf(\sigma))$. Therefore SPD suffers more than our algorithms from an increase in m. On the other hand, for larger alphabets, SPD should beat BBB/TBB for larger m values, and vice versa for smaller alphabets. In these experiments we have considered only the MIDI application where $\sigma = 128$.

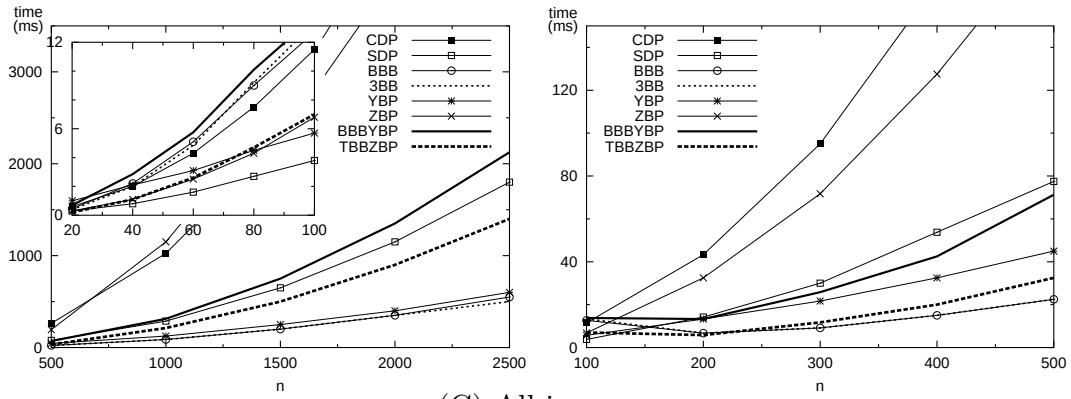
Note also that the cost of all branch and bound algorithms actually decreases up to length 200 and then starts to grow again (actually BBB/3BB is worse than CDP up to length 100). This is because the number of tree nodes processed decreases as the string lengths grow. The reason is that the LCS of longer strings gives finer grained information and hence permits finding the right transposition faster. For long enough strings, of course, the $O(mn)$ cost to compute each tree node takes over.



(A) Classical Algorithms



(B) Bit-Parallel Algorithms



(C) All in one

Fig. 15. Comparison of LCTS algorithms. Left hand side graphs are for small (20–100, plot inside the graph) and large (500–2500) lengths. Right hand size graphs are for intermediate (100–500) lengths.

7.2 Bit-Parallel Algorithms

Our next experiment studies the effects that bit-parallelism has over classical algorithms. We consider two types of bit-parallel algorithms: without hierarchy (codes 5 and 6 in Table 1) and with hierarchy (codes 7 and 8 in Table 1).

For the former, we compared YBP and ZBP. YBP uses the bit-parallel algorithm by Crochemore et al. [4] to compute each possible LCS^c value in isolation. ZBP uses bit-parallelism to compute several LCS^c values together. Fig. 15-B shows that ZBP is faster for string sizes smaller than 80, but slower for longer strings. This is because, the bigger the input, the more bits are needed to store each cell value, and therefore more words are required by ZBP. In complexity terms, The YBP approach is $O(\sigma mn/w)$ time while ZBP is $O(\sigma mn \log(m)/w)$, hence it worsens faster with m.

The other two bit-parallel algorithms we implemented were BBBYBP and TBBZBP. In BBBYBP(TBBZBP) we use YBP(ZBP) instead of CDP to compute each node in BBB(TBB). As can be seen in Fig. 15-B, BBBYBP is never relevant. This is probably because we cannot precompute the match table for $LCS^X(A, B)$ as efficiently as that for $LCS^c(A, B)$, so BBBYBP is never better than YBP. TBBZBP, on the other hand, is much better than ZBP for long strings, and it is clearly the fastest choice up to length 600 or so. At that point its $O(\log m)$ extra cost compared to YBP becomes noticeable and YBP wins.

7.3 Overall Comparison

Fig. 15-C shows how all algorithms compare to each other. It can be seen that ZBP is the fastest for short sequences (length up to 30), where its bit-parallelism is highest. From moderate length strings (length from 30 to 120), SDP is clearly the best choice. For longer strings (length from 120 to 230), TBBZBP dominates. Finally, for large strings (longer than 230), BBB/3BB is the fastest, closely followed by YBP. All those length ranges turn out to be relevant for different problems in music-related applications.

These results also permit figuring out what text search costs would be. If we exclude hierarchical schemes, we have that ZBP is the best to search for short patterns ($m \leq 30$), SDP for medium-length patterns ($m \leq 160$), and YBP for long patterns ($m > 160$). For small k, ZBP can be adapted as in Section 5 to have a speedup of $O(w/\log k)$ instead of $O(w/\log m)$. However, in practice, the resulting code is more complex, so it is not clear how advantageous that would be. Algorithms SDP and YBP, on the other hand, do not benefit at all from a lower k value.

Other factors that would affect the performance are the length w of the computer word and the alphabet size σ. On a $w = 64$ bit machine, ZBP, TBBZBP and YBP performances would double, although for YBP this would be noticeable only for $m > 32$. On the other hand, an increase in σ (for a different application) proportionally affects ZBP and YBP performance, while BBB/3BB costs are expected to grow slower, and SDP remains essentially unaffected.

8 Conclusions

In this paper we have focused on string matching problems that have applications in music comparison and retrieval. Three specific features typical to music retrieval, not taken into account in conventional string pattern matching, are (a) approximate searching permitting missing, extra, and distorted notes, (b) transposition invariance to allow matching a sequence that appears in a different scale, and (c) handling polyphonic music.

We have introduced two classes of algorithms to cope with this problem. The first one uses branch and bound over the set of possible transpositions in order to find the optimal one without trying them all. The second family uses bit-parallelism to compare strings under several different transpositions in one shot. The ideas can also be combined to obtain other new algorithms.

We have shown experimentally that our algorithms are competitive with the best existing choices. In particular, our bit-parallel algorithm turns out to be the fastest to handle short strings (of length up to 30), which covers many interesting cases of music comparison, and especially, searching for music passages over long music files. Our branch and bound algorithms, on the other hand, turn out to be the best to compare long strings (longer than 120), which covers other cases of music comparison, especially those related to global comparison of musical pieces.

References

[1] A. Bergeron and S. Hamel. Vector algorithms for approximate string matching. *International Journal of Foundations of Computer Science*, 13(1):53–65, 2002.

[2] E. Cambouropoulos. A general pitch interval representation: Theory and applications. *Journal of New Music Research*, 25:231–251, 1996.

[3] M. Crochemore, C. S. Iliopoulos, G. Navarro, and Y. Pinzon. A bit-parallel suffix automaton approach for (δ, γ)-matching in music retrieval. In *Proc. 10th International Symposium on String Processing and Information Retrieval (SPIRE'03)*, volume 2857 of *Lecture Notes in Computer Sience*, pages 211–223. Springer-Verlag, 2003.

[4] M. Crochemore, C.S. Iliopoulos, Y.J. Pinzon, and J.F. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. *Information Processing Letters*, 80(6):279–285, 2001.

[5] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

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

[7] G. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, 1999.

[8] H. Hyyrö and G. Navarro. Faster bit-parallel approximate string matching. In *Proc. 13th Annual Symposium on Combinatorial Pattern Matching (CPM'02)*, volume 2373 of *Lecture Notes in Computer Science*, pages 203–224. Springer-Verlag, 2002.

[9] K. Lemström and P. Laine. Musical information retrieval using musical parameters. In *Proc. 1998 International Computer Music Conference*, pages 341–348, Ann Arbor, MI, 1998.

[10] K. Lemström and G. Navarro. Flexible and efficient bit-parallel techniques for transposition invariant approximate matching in music retrieval. In *Proc. 10th Int'l Symp. on String Processing and Information Retrieval (SPIRE'03)*, volume 2857 of *Lecture Notes in Computer Science*, pages 224–237. Springer-Verlag, 2003.

[11] K. Lemström, G. Navarro, and Y. Pinzon. Bit-parallel branch and bound algorithm for transposition invariant LCS. Submitted to conference.

[12] K. Lemström and J. Tarhio. Transposition invariant pattern matching for multi-track strings. *Nordic Journal of Computing*, 10(3):185–205, 2003.

[13] K. Lemström and E. Ukkonen. Including interval encoding into edit distance based music comparison and retrieval. In *Proc. AISB'00 Symposium on Creative & Cultural Aspects and Applications of AI & Cognitive Science*, pages 53–60, Birmingham, April 2000.

[14] V. Mäkinen, G. Navarro, and E. Ukkonen. Algorithms for transposition invariant string matching. In *Proc. 20th Int'l Symp. on Theoretical Aspects of Computer Science (STACS'03)*, volume 2607 of *Lecture Notes in Computer Science*, pages 191–202, 2003.

[15] MIDI Manufacturers Association, Los Angeles, California. *The Complete Detailed MIDI 1.0 Specification*, 1996.

[16] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999.

[17] E. Ukkonen, K. Lemström, and V. Mäkinen. Sweepline the music! In *Computer Science in Perspective — Essays Dedicated to Thomas Ottmann*, volume 2598 of *Lecture Notes in Computer Science*, pages 330–342. Springer-Verlag, 2003.

[18] Paul W. and Simon J. Decision trees and random access machines. In *Proc. Int'l. Symp. on Logic and Algorithmic*, pages 331–340, Zurich, 1980.

[19] G.A. Wiggins, K. Lemström, and D. Meredith. SIA(M): A family of efficient algorithms for translation-invariant pattern matching in multidimensional datasets. Submitted.