Flexible and Efficient Bit-Parallel Techniques for Transposition Invariant Approximate Matching in Music Retrieval

Kjell Lemström[1] and Gonzalo Navarro[⋆2]

[1] Department of Computer Science, University of Helsinki, Finland
[2] Department of Computer Science, University of Chile
`klemstro@cs.helsinki.fi, gnavarro@dcc.uchile.cl`

Abstract. Recent research in music retrieval has shown that a combinatorial approach to the problem could be fruitful. Three distinguishing requirements of this particular problem 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. These combined requirements make up a complex combinatorial problem that is currently under research. On the other hand, bit-parallelism has proved a powerful practical tool for combinatorial pattern matching, both flexible and efficient. In this paper we use bit-parallelism to search for several transpositions at the same time, and obtain speedups of $O(w/\log k)$ over the classical algorithms, where the computer word has w bits and k is the error threshold allowed in the match. Although not the best solution for the easier approximation measures, we show that our technique can be adapted to complex cases where no competing method exists, and that are the most interesting in terms of music retrieval.

1 Introduction

Combinatorial pattern matching with its many application domains have been an active research field already for several decades. One of the latest such domains is *music retrieval*. Indeed, music can be encoded as sequences of symbols, i.e. as strings. At a rudimentary level this is done by taking into account exclusively the order of the starting times of the musical events (i.e., the *note on*s) together with their *pitch* information (i.e. the frequency, the perceived height of the musical event). On a more complicated level one can use several distinct attributes for each of the events (see e.g. [1, 9]). Most of the interesting musical attributes used in such symbolic representations are directly available in MIDI format [13] which is a commonly used compact symbolic representation.

A straightforward application of general string matching techniques on symbolic music representation, however, does not suffice for musically pertinent matching queries; music has special features that have not been considered in

⋆ Partially supported by Fondecyt Grant 1-020831.

general string matching techniques. Firstly, music is often *polyphonic*, i.e., there are several events occurring simultaneously (in a case where there exists no simultaneous events the music is said to be *monophonic*). These simultaneous events may have a collective meaning and, therefore, the polyphony has to be preserved and taken into account in the matching process. For instance, a typical music retrieval, or searching problem, is the *distributed string matching problem*: given a set t (called a *text* or a *target*) of h strings (each representing a *voice*) $t^i = t^i_1, \ldots, t^i_n, \;\; i \in \{1 \ldots h\}$, for some constant h and a *pattern* $p = p_1, \ldots, p_m$, we say that p occurs at position j of t if $p_1 = t^{i_1}_j, p_2 = t^{i_2}_{j+1}, \ldots, p_m = t^{i_m}_{j+m-1}$ for some $\{i_1, \ldots, i_m\} \in \{1 \ldots h\}$. The problem has been studied in [7, 10].

Secondly, western people tend to listen 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 and recognized as the same regardless of the performed pitch level. This leads to the concept of *transposition invariance*. Formally, the *transposition invariant distributed string matching problem* is as follows. Given a monophonic pattern p and a polyphonic target t of h voices, $t^i = t^i_1 \cdots t^i_n, i \in \{1, \ldots, h\}$, the task is to find all the js such that $p_1 = t^{i_1}_j + c, p_2 = t^{i_2}_{j+1} + c, \ldots, p_m = t^{i_m}_{j+m-1} + c$ holds, for some constant c and $\{i_1, \ldots, i_m\} \in \{1, \ldots, h\}$ [10].

Thirdly, real music is often decorated, i.e., it may contain grace notes or ornamentations, for instance. The conventional procedure to overcome this problem is to allow gaps between the consecutive matching elements in found occurrences [2, 6, 16]. The choices are either to use parametrized gapping (as in [2, 6]) or arbitrary gapping (as in [16]). As we aim at a matching method that finds all the occurrences (although it may also find spurious ones), we will use the latter approach. Instead of using the geometric approach of Wiggins et al. [16], we will use the string matching framework and apply the indel distance (the dual of LCS-matching) [5]. We claim that it is a more fruitful approach not to drop any occurrences although in some situations it may lead to a large number of spurious occurrences. The set of found occurrences may then be post-processed by musically motivated filters, for instance by those discussed in [12].

Fourthly, in a typical transposition invariant distributed string matching application the query pattern is given by humming. This kind of an application is sometimes referred as "WYHIWYG" (What You Hum Is What You Get) or "query by humming". In such a case we may expect that all the events in the hummed query pattern are relevant, but its (absolute) pitch values may be somewhat distorted. This distortion has the form of Gaussian distribution with the mean value of the correct (desired) pitch and with a relatively small variance. Therefore, in a WYHIWYG application, we would like to enable some tolerance for such errors. Here we consider two solutions for this problem, the first of which is the so-called δ-matching [3]. The pattern $p = p_1 \cdots p_m$ is said to have a δ-match in $t_1 \cdots t_n$ if $p_i \in [t_{j+i-1} - \delta, t_{j+i-1} + \delta]$ for all $i = 1, \ldots, m$. Although this approach works reasonably well in practice, it is musically more appropriate to penalize an error according to how much the pitch differs from the desired one than to allow any distortion as long it is within the allowed tolerance. Therefore, we will use a more general distance function which implements the claim above.

Although all the problems given above have been studied, no current solution can solve them all. Most relevantly, the bit-parallel algorithm by Crochemore et al. [4] can compute the LCS in $O(m^2/w)$ time, where w denotes the size of the computer word in bits. Moreover, as we discuss in Section 5, the algorithm can be extended straightforwardly to deal with polyphony, transpositions and δ matching in $O(h\sigma m^2/w)$ time (here σ denotes the number of possible transpositions). Furthermore, the same complexity is obtainable with the unit-cost edit distance by using other bit-parallel algorithms [14, 8].

Our solution is also based on bit-parallelism, which is well-known for its flexibility. Our transposition invariant δ-matching algorithm for distributed string matching runs in time $O(\sigma m^2 \log(m)/w)$. Noteworthy, it is capable of applying more general and musically pertinent distance functions than the previous related solutions, e.g. those that are not based on unit costs.

2 Preliminaries

Let us start this section by a brief introduction to string combinatorics. Let Σ be a finite set of symbols, called an *alphabet*, and $\sigma = |\Sigma|$. 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 $A = a_1 \cdots a_m$. The *length* of A is $|A| = m$. The string of length 0 is called the *empty string* and denoted λ. 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 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, i.e., $A' = a_{i_1} a_{i_2} \cdots a_{i_m}$, 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) $T \subseteq \Sigma^* \times \Sigma^*$ and a non-negative valued *cost function* W that gives for each transformation t in T a cost $W(t)$. Each t in T is a pair of strings $t = (\alpha, \beta)$. Observing such a t as a rewriting rule, suggests a notation for t, $\alpha \rightarrow \beta$ (α is replaced by β within a string containing α), which we will use below. For convenience, if $\alpha \rightarrow \beta \notin T$, then $W(\alpha \rightarrow \beta) = \infty$.

The definition of the distance is based on the concept of *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 $A = \alpha_1\alpha_2 \cdots \alpha_p$, and $B = \beta_1\beta_2 \cdots \beta_p$, 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 $\alpha_1 \rightarrow \beta_1,\ \alpha_2 \rightarrow \beta_2, \ldots, \alpha_p \rightarrow \beta_p$.

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

The general definition above induces, for instance, 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 $a \rightarrow b$ (substitution), $a \rightarrow \lambda$ (deletion), and $\lambda \rightarrow a$ (insertion), where $a, b \in \Sigma$. The costs are given as $W(a \rightarrow a) = 0$ for all a, $W(a \rightarrow b) = 1$ for all $a \neq b$, and $W(a \rightarrow \lambda) = W(\lambda \rightarrow a) = 1$ for all a. In *Hamming distance*, $D_H(A,B)$, the only allowed local transformations are of form $a \rightarrow b$ where a and b are any members of Σ, with cost $W(a \rightarrow a) = 0$ and $W(a \rightarrow b) = 1$, for $a \neq b$. Finally, the *indel distance*, $D_{LCS}(A,B)$, is defined as Levenshtein distance without the possibility to use substitutions.

It is well-known that the straightforward computation of these distances is by using recurrences like the following used for $D_{LCS}(A,B)$:

$$d_{i,0} = i; \;\; d_{0,j} = j;$$
$$d_{ij} = \min \begin{cases} d_{i-1,j} + 1 \\ d_{i,j-1} + 1 \\ d_{i-1,j-1}, \text{if } a_i = b_j. \end{cases}$$

The evaluation of such a recurrence is done by *dynamic programming*, where the distances between the prefixes of A and B are tabulated. Each cell d_{ij} of the distance table (d_{ij}) stores the distance between $a_1 \cdots a_i$ and $b_1 \cdots b_j$ $(0 \leq i \leq m$, $0 \leq j \leq n)$ and (d_{ij}) is evaluated by proceeding row-by-row or column-by-column using the recurrence. Finally, $d_{m,n}$ gives the distance, in this case $D_{LCS}(A,B)$.

The dual case of $D_{LCS}(A,B)$ 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:

$$\begin{aligned} LCS_{i,0} &\leftarrow 0; \;\; LCS_{0,j} \leftarrow 0; \\ LCS_{i+1,j+1} &\leftarrow \text{if } \;\; a_{i+1} = b_{j+1} \quad \text{then } \; 1 + LCS_{i,j} \\ &\qquad\qquad\qquad\qquad\quad \text{else } \; \max(LCS_{i,j+1}, LCS_{i+1,j}). \end{aligned} \tag{1}$$

Now it is rather clear that $LCS(A,B) = \frac{|A|+|B|-D_{LCS}(A,B)}{2}$.

If we want to calculate the length of the *longest common transposition invariant subsequence*, $LCTS(A,B)$, it may be done by calculating $LCS^c(A,B)$ by all the possible $2\sigma + 1$ transpositions, and select the transposition c which gives the maximum [11]. $LCS^c(A,B)$ is defined just like $LCS(A,B)$ except that there is a match when $a_{i+1} + c = b_{j+1}$. Our idea is to simulate the computation of the (d^c_{ij}) tables, for $c = [-\sigma, \sigma]$, so that the aligned d_{ij} values are computed simultaneously in a bit vector, as long as they fit in the used computed word of w bits (see Fig. 1). Typical sizes of alphabet are, e.g., 88 (the number of keys in piano) and 127 (the number of MIDI pitch values), and 32 or 64 for the size of the current computer word. In practice, we need 3–8 bit-vectors for each d_{ij}.

Finally, the weighted edit distance that we use to make a distinction according to the amount of the local distortion is as follows:

$$\begin{aligned} ED_{i,0} &\leftarrow i \times ID; \;\; ED_{0,j} \leftarrow j \times ID; \\ ED_{i+1,j+1} &\leftarrow \min(|a_{i+1} - b_{j+1}| + ED_{i,j}, ID + ED_{i,j+1}, ID + ED_{i+1,j}), \end{aligned} \tag{2}$$

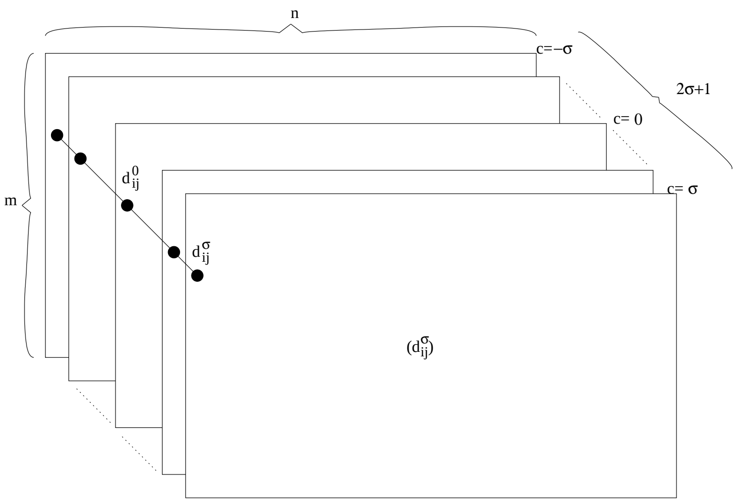

Fig. 1. We calculate in parallel $2\sigma + 1$ (d_{ij}) tables. The idea is to present the aligned nodes $d_{ij}^{-\sigma} \ldots d_{ij}^{\sigma}$ with a single bit-vector (as long as they fit in a computer word).

where ID is a constant used for indel operations.

3 A Bit-Parallel Algorithm

We present a speedup technique for the computation of the $2\sigma+1$ LCS matrices. We resort to bit-parallelism, that is, to storing several values inside the same computer word. For this sake we will denote the bitwise *and* operation as "&", the *or* as "|", and the bit complementation as "$\sim$". Shifting i positions to the left (right) is represented as "$<< i$" ("$>> i$"), where the bits that fall are discarded and the new bits that enter are zero. We can also perform arithmetic operations over the computer words. We use exponentiation to denote bit repetition, e.g. $0^3 1 = 0001$, and write the most significant bit as the leftmost bit. When we write $[x]_\ell$ we mean the integer x represented in ℓ bits.

Since the values of the LCS matrix are in the range $\{0 \ldots \min(|a|, |b|)\}$, we need $\ell = \lceil \log_2(\min(|a|, |b|) + 1) \rceil$ bits to store them. This means that in a computer word of w bits we can store $\lfloor w/\ell \rfloor$ counters. For reasons that will be made clear soon, we will in fact need $\ell + 1$ bits per counter, where the highest bit will always be zero, and hence we will be able to store $A = \lfloor w/(\ell + 1) \rfloor$ counters.

We will divide the process of computing $LCS^c(a, b)$ for every $c \in \Sigma$ into $\lceil (2\sigma + 1)/A \rceil$ separate bit-parallel computations, each for A contiguous c values. From now on, let us consider that we are computing in parallel $LCS^c(a, b)$ for $c \in \{C \ldots C + A - 1\}$.

The first problem to bit-parallelize Eq. (1) is its if-then-else structure. For a given c, if $a_{i+1} + c = b_{j+1}$ we have to use the value $1 + LCS^c_{i,j}$, otherwise we have to use $\max(LCS^c_{i+1,j}, LCS^c_{i,j+1})$. We solve this by using a bit-mask B of length $A(\ell+1)$, which should have all 1's in the c values for which $a_{i+1} + c = b_{j+1}$, and zeros elsewhere. This means that we have 1's only for the value $c = b_{j+1} - a_{i+1}$. It is possible that this c value is outside our current range $\{C \ldots C + A - 1\}$. So the computation of B is as follows:

$$B \leftarrow \text{ if } C \leq b_{j+1} - a_{i+1} < C + A \quad \begin{cases} \text{then } \; 0^{(A+C-1-(b_{j+1}-a_{i+1}))(\ell+1)} \; 1^{(\ell+1)} \; 0^{(b_{j+1}-a_{i+1}-C)(\ell+1)} \\ \text{else } \; 0^{A(\ell+1)} \end{cases}$$

Once we have computed B, we want to take the value $1 + LCS^c_{i,j}$ for the c values where B has 1's and the value $\max(LCS^c_{i+1,j}, LCS^c_{i,j+1})$ elsewhere. For the former we need to add 1 to all the counters at the same time, which is easily achieved by adding $(0^\ell 1)^A$. For the latter we need to compute max() in bit-parallel. Let us call Max this function. Hence the value we want is

$$LCS_{i+1,j+1} \;\; \leftarrow \;\; (B \;\&\; (LCS_{i,j} + (0^\ell 1)^A)) \mid (\sim B \;\&\; Max(LCS_{i+1,j}, LCS_{i,j+1}))$$

To compute $Max(X, Y)$, where X and Y contain several counters properly aligned, we need the aforementioned extra highest bit per counter, always zero. We precompute the bit mask $J = (10^\ell)^A$ and perform the operation $F \leftarrow ((X \mid J) - Y) \;\&\; J$. The result is that, in F, each highest bit is set iff the counter of X is larger than that of Y. We now compute $F \leftarrow F - (F >> \ell)$, so that the counters where X is larger than Y have all their bits set in F, and the others have all the bits in zero. Finally, we choose the maxima as $Max(X,Y) \leftarrow (F \;\&\; X) \mid (\sim F \;\&\; Y)$. Also, we easily obtain $Min(X,Y) \leftarrow (F \;\&\; Y) \mid (\sim F \;\&\; X)$. Fig. 2 gives the code. These methods are due to [15].

Fig. 3 shows **RangeLCTS**, the bit-parallel algorithm for a range of counters $C \ldots C + A - 1$. Using this algorithm we traverse all the $c \in \Sigma$ values and compute $LCTS(a,b) = \max_{c \in -\sigma \ldots \sigma} LCS^c(a,b)$. This is done by **LCTS**.

Let us now analyze the algorithm. **LCTS** runs $(2\sigma + 1)/A$ iterations of **RangeLCTS** plus a minimization over $2\sigma + 1$ values. In turn, **RangeLCTS** takes $O(|a||b|)$ time. Since $A = w / \log_2 \min(|a|,|b|)(1 + o(1))$, the algorithm is $O(\sigma |a||b| \log(\min(|a|,|b|))/w)$ time. If $|a| = |b| = m$, the algorithm is $O(\sigma m^2 \log(m)/w)$ time, which represents a speedup of $\Omega(w/\log m)$ over the naive $O(\sigma m^2)$ time algorithm.

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 $b_{j+1} - a_{i+1} - \delta \leq c \leq b_{j+1} - a_{i+1} + \delta$. The only change needed in our algorithm is in lines 5–7 of **RangeLCTS**, which should become:

```
$low \leftarrow \max(C, b_j - a_i - \delta)$
$high \leftarrow \min(C + A - 1, b_j - a_i + \delta)$
**If** $low \leq high$ **Then**
```

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

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

Fig. 2. Bit-parallel computation of maximum and minimum between two sets of counters aligned in a computer word. In practice J is precomputed.

```
    $B \leftarrow 0^{(A+C-1-high)(\ell+1)}\ 1^{(high-low+1)(\ell\ +1)} 0^{(low-C)(\ell+1)}$
**Else** $B \leftarrow 0^{A(\ell+1)}$
```

4 Text Searching

The above procedure can be adapted to search for a pattern P of length m in a text T of length n under the indel distance, permitting transposition invariance. The goal is, given a threshold value k, report all text positions j such that $d(P, T_{j' \ldots j}) \leq k$ for some j', where d is the indel distance (number of character insertions and deletions needed to make two strings equal).

Additionally, we can search polyphonic text, where there are actually h parallel texts $T^1 \ldots T^h$, and text position j matches any character in the set $\{T^1_j \ldots T^h_j\}$.

Let us consider a new recurrence for searching. We start with a column $D_i = i$ and update D to D' for every new text position j. For every j where $D_m \leq k$ we report text position j as the end position of a recurrence. The formula for searching with indel distance using transposition c is as follows:

$$D'^{c}_{0} \leftarrow 0$$
$$D'^{c}_{i+1} \leftarrow \text{if} \quad P_{i+1} + c \in \{T^1_j \ldots T^h_j\} \quad \text{then} \quad D^c_i \quad \text{else} \quad 1 + \min(D'^{c}_{i}, D^c_{i+1})$$

where we note that we have suppressed column number j, as we will speak about the *current* column (D' or $newD$) built using the *previous* column (D or $oldD$).

```
**RangeLCTS** $(a,\ b,\ C,\ A,\ \ell)$
 1.   **For** $i \in 0 \ldots |a|$ **Do**
 2.       **For** $j \in 0 \ldots |b|$ **Do**
 3.           **If** $i = 0\ \vee\ j = 0$ **Then** $LCS_{i,j} \leftarrow 0^{A(\ell+1)}$
 4.           **Else**
 5.               **If** $C \le b_j - a_i < C + A$ **Then**
 6.                   $B \leftarrow 0^{(A+C-1-(b_j-a_i))(\ell+1)}\ 1^{(\ell+1)}\ 0^{(b_j-a_i-C)(\ell+1)}$
 7.               **Else** $B \leftarrow 0^{A(\ell+1)}$
 8.               $LCS_{i,j} \leftarrow (B\ \&\ (LCS_{i-1,j-1} + (0^\ell 1)^A))$
                          $|\ (\sim B\ \&\ Max(LCS_{i-1,j}, LCS_{i,j-1})$
 10.  **Return** $LCS_{|a|,|b|}$
```

```
**LCTS** $(a,\ b,\ \sigma)$
 1.   $\ell \leftarrow \lceil \log_2(\min(|a|,|b|)+1) \rceil$
 2.   $A \leftarrow \lfloor w/(\ell+1) \rfloor$
 3.   $c \leftarrow -\sigma$
 4.   $Max \leftarrow 0$
 5.   **While** $c \le \sigma$ **Do**
 6.       $V \leftarrow$ **RangeLCTS**$(a,b,c,A,\ell)$
 7.       **For** $t \in c \ldots c+A-1$ **Do**
 8.           $Max \leftarrow \max(Max, (V >> (t-c)(\ell+1))\ \&\ 0^{(A-1)(\ell+1)}01^\ell)$
 9.       $c \leftarrow c + A$
 10.  **Return** $Max$
```

Fig. 3. Computing $LCTS(a,b)$ using bit-parallelism. **RangeLCTS** computes $LCS^c(a,b)$ for every $c \in C \ldots C{+}A{-}1$ in bit-parallel, and returns a bit mask containing $LCS^c(a,b)$ for all those c values.

Additionally, we note that, when a 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. Hence the number of bits needed by a counter is $\ell = \lceil \log_2(k+2) \rceil$.

The new recurrence requires the same tools we have already developed for the LCS computation, except for the polyphony issue and for the $k+1$ limit. Polyphony can be accommodated by *or*-ing the B masks corresponding to the different text characters at position j. The $k+1$ limit has to be taken care of only when we add 1 in the "else" clause of the recurrence.

The typical way to solve this requires one extra bit for the counters. We prefer instead to reuse our result for bit-parallel minimum. The recurrence can be rewritten as follows, which guarantees that any value larger than k stays at $k+1$.

$$D'^{\,c}_{0} \leftarrow 0$$
$$D'^{\,c}_{i+1} \leftarrow \text{if}\ \ P_{i+1} + c \in \{T_j^1 \ldots T_j^h\}\ \ \text{then}\ \ D_i^c\ \ \text{else}\ \ 1 + \min(D'^{\,c}_{i}, D^c_{i+1}, k)$$

Finally, we have to report every text position where $D_m \leq k$. In our setting, this means that any counter different from $k+1$ makes the current text position to be reported.

Fig. 4 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. The algorithm is $O(h\sigma mn \log(k)/w)$ time, which represents a speedup of $O(w/\log k)$ over the classical solution.

```
RangeIDSearch ($P,\ T^1 \ldots T^h,\ k,\ C,\ A,\ \ell$)
 1.    $K \leftarrow [k]_{\ell+1} \times (0^\ell 1)^A$
 2.    $Kp1 \leftarrow K + (0^\ell 1)^A$
 3.    For $i \in 0 \ldots k$ Do $D_i \leftarrow [i]_{\ell+1} \times (0^\ell 1)^A$
 4.    For $i \in k+1 \ldots |P|$ Do $D_i \leftarrow Kp1$
 5.    For $j \in 1 \ldots |T|$ Do
 6.        $oldD \leftarrow 0$
 7.        For $i \in 1 \ldots |P|$ Do
 8.            $B \leftarrow 0^{A(\ell+1)}$
 9.            For $g \in 1 \ldots h$ Do
10.                If $C \leq T_j^g - P_i < C + A$ Then
11.                    $B \leftarrow B \mid 0^{(A+C-1-(T_j^g-P_i))(\ell+1)}\ 1^{(\ell+1)}\ 0^{((T_j^g-P_i)-C)(\ell+1)}$
12.            $newD \leftarrow (B \,\&\, oldD) \mid (\sim B \,\&\, (Min(Min(D_{i-1}, D_i), K) + (0^\ell 1)^A))$
13.            $oldD \leftarrow D_i,\ D_i \leftarrow newD$
14.        If $newD \neq Kp1$ Then Report an occurrence ending at $j$
```

```
IDSearch ($P,\ T^1 \ldots T^h,\ k,\ \sigma$)
 1.    $\ell \leftarrow \lceil \log_2(k+1) \rceil$
 2.    $A \leftarrow \lfloor w/(\ell+1) \rfloor$
 3.    $c \leftarrow -\sigma$
 4.    While $c \leq \sigma$ Do
 5.        RangeIDSearch($P, T^1 \ldots T^h, k, c, A, \ell$)
 6.        $c \leftarrow c + A$
```

Fig. 4. Searching polyphonic text with indel distance permitting any transposition.

5 A More General Distance Function

Although we have obtained important speedups with respect to classical algorithms, it turns out that there exist bit-parallel techniques that can compute the LCS in $O(m^2/w)$ time [4]. Extending these algorithms naively to deal with polyphony, transpositions and δ matching yields $O(h\sigma m^2/w)$ time. Although it has not been done, we believe that it is not hard to convert these algorithms into search algorithms for indel distance at $O(h\sigma mn/w)$ cost, which is better than ours by an $O(\log k)$ factor. The same times can be obtained if we use edit distance instead of indel distance [14, 8].

The strength of our approach resides in that we are using bit-parallelism in a different dimension: rather than computing several cells of a matrix in parallel, we compute several transpositions in parallel, while the cells are computed 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. Parallelizing the work inside the matrix is more complex, and has been achieved only for unit-cost distances. As explained before, 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. We demonstrate the flexibility of our approach by addressing the computation of the weighted edit distance detailed in Eq. (2). Which follows is the search version for a given transposition c in polyphonic text, bounded by $k+1$.

$$D'^{\,c}_{0} \leftarrow 0$$
$$D'^{\,c}_{i+1} \leftarrow \min(\min_{g \in 1 \ldots h} |P_{i+1} + c - T^g_j| + D^c_{i-1}, ID + D'^{\,c}_{i}, ID + D^c_{i+1}, k+1)$$

There are two challenges to bit-parallelize this recurrence. The first is that ensuring that we never surpass $k+1$ is more difficult, because the increments are not only by 1. We choose to compute the full values and then take minimum with $k+1$, as suggested by the recurrence. However, the intermediate values can be larger. Since $ID \leq k$ (otherwise the problem is totally different), the latter terms are bounded by $2k+1$. We will manage to keep also the first term of the minimization below $2k+2$. This means that we need $\lceil \log_2(2k+3) \rceil$ bits for our counters.

The second challenge is how to compute $|P_{i+1} + c - T^g_j|$ in bit-parallel for a set of consecutive c values, with the added trouble of not exceeding $k+1$ in any counter. Depending on the range $C \ldots C+A-1$ of transpositions we are considering, these values form an increasing, decreasing, or decreasing-then-increasing sequence. For shortness, we will use $[x]$ to denote $[x]_{\ell+1}$ in this discussion. An increasing sequence of the form $I_t = [t+A-1] \ldots [t+1]\ [t]$, $t \geq 0$, is obtained simply as $I_t \leftarrow (0^\ell 1)^{A-1}[t] \times (0^\ell 1)^A$. A version bounded by $r \geq t$ is obtained as $I^r_t \leftarrow (I_t \ \&\ 0^{(A-(r-t))(\ell+1)} 1^{(r-t)(\ell+1)}) \mid ([r]^{A-(r-t)} 0^{(r-t)(\ell+1)})$. Similarly, a decreasing sequence $D_t = [t-A+1] \ldots [t-1]\ [t]$ is obtained as $D_t \leftarrow [t]^A - I_0$. The bounded version is obtained similarly as for increasing sequences. Finally, a decreasing-then-increasing sequence $DI_t = [A-t-1]\ [A-t-2] \ldots [2]\ [1]\ [0]\ [1] \ldots [t-1]\ [t]$ is obtained as $DI_t = (I_0 << t(\ell+1)) \mid (D_A >> (A-t)(\ell+1))$. The bounded version DI^r is obtained similarly, using I^r and D^r instead. We could even accommodate substitution costs of the form $|a_i - b_j|/q$ for integer q by multiplying by $(0^{q(\ell+1)-1}1)^A$ instead of by $(0^\ell 1)^A$. Fig. 5 gives the code to build these sequences.

It becomes clear that we can perform approximate searching using this general distance function, permitting transposition invariance and polyphony, in $O(h\sigma mn \log(k)/w)$. This cannot be done with previous approaches and illustrates the strength of our method. Fig. 6 gives the details.

```
I $(t,\ r,\ A,\ \ell)$
1.    If $r \le t$ Then Return $[r]_{\ell+1} \times (0^\ell 1)^A$
2.    $I_t \leftarrow ((0^\ell 1)^{A-1} 0^{\ell+1} \mid [t]_{\ell+1}) \times (0^\ell 1)^A$
3.    Return $(I_t \ \& \ 0^{(A-(r-t))(\ell+1)} 1^{(r-t)(\ell+1)}) \mid ([r]_{\ell+1} \times (0^\ell 1)^{A-(r-t)} 0^{(r-t)(\ell+1)})$
```

```
D $(t,\ r,\ A,\ \ell)$
1.    If $r \ge t$ Then $r \leftarrow t$
2.    $D_r \leftarrow [r]_{\ell+1} \times (0^\ell 1)^A - (0^\ell 1)^{A-1} 0^{\ell+1} \times (0^\ell 1)^A$
3.    Return $(D_r << (t-r)(\ell+1)) \mid ([r]_{\ell+1} \times 0^{(A-(t-r))(\ell+1)} (0^\ell 1)^{(t-r)(\ell+1)})$
```

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

Fig. 5. Bit-parallel code to obtain increasing, decreasing, and decreasing-then-increasing sequences.

SUN	n =10	100	1,000	10,000
LCTS	0.00049	0.07334	14.6688	1,466.88
CDP	0.00124	0.124173	12.4173	1,241.73

Table 1. Execution times (in sec) for our LCTS and CDP when running on Sun.

6 Experiments

We conducted a brief experiment on comparing the efficiency between our LCTS method and an algorithm based on classical dynamic programming (CDB). The experiment was run on two distinct computers, the first of which was a Sun UltraSparc-1 running SunOS 5.8 with 167 MHZ and 64 Mb RAM, and the second was a Pentium IV running Linux 2.4.10-4GB with 2 GHZ and 512 MB RAM.

Both the codes for our **RangeLCTS** algorithm and for the classical dynamic programming were highly optimized. In the experiment we used LCS matrixes of size $10,000 \times 10,000$ (the content was pitch values of musical data). We measured the CPU times spent by 1 iteration of the **RangeLCTS** (for the whole LCTS query we then calculated the required total time for a given w) and by CDB. Moreover, since both of the algorithms scale up well with n^2, we were able to estimate the running times for distinct ranges of n.

Table 1 gives results when running the two observed algorithms when executed on Sun. Note that we are better for large $n < 1,000$ (more precisely, up to $n = 510$). The reason is that our counters have to maintain the current LCS values in $w = 32$ bits, which can be as large as n.

```
RangeEDSearch ($P,\ T^1 \dots T^h,\ k,\ C,\ A,\ \ell$)
1.   $Kp1 \leftarrow [k+1]_{\ell+1} \times (0^\ell 1)^A$
2.   $IDmask \leftarrow [ID]_{\ell+1} \times (0^\ell 1)^A$
3.   For $i \in 0 \dots \lfloor k/ID \rfloor$ Do $D_i \leftarrow [i \cdot ID]_{\ell+1} \times (0^\ell 1)^A$
4.   For $i \in k+1 \dots |P|$ Do $D_i \leftarrow Kp1$
5.   For $j \in 1 \dots |T|$ Do
6.       $oldD \leftarrow 0$
7.       For $i \in 1 \dots |P|$ Do
8.           $B \leftarrow Kp1$
9.           For $g \in 1 \dots h$ Do
10.              If $T_j^g - P_i \le C$ Then // Increasing sequence
11.                  $B' \leftarrow \mathbf{I}(C - (T_j^g - P_i), k+1, A, \ell)$
12.              Else If $T_j^g - P_i \ge C + A$ Then // Decreasing sequence
13.                  $B' \leftarrow \mathbf{D}((T_j^g - P_i) - C, k+1, A, \ell)$
14.              Else $B' \leftarrow \mathbf{DI}((T_j^g - P_i) - C, k+1, A, \ell)$
15.              $B \leftarrow Min(B, B')$
16.          $newD \leftarrow Min(Min(B + oldD, Min(D_{i-1}, D_i) + IDmask), Kp1)$
17.          $oldD \leftarrow D_i,\ D_i \leftarrow newD$
18.      If $newD \neq Kp1$ Then Report an occurrence ending at $j$
```

```
EDSearch ($P,\ T^1 \dots T^h,\ k,\ \sigma$)
1.   $\ell \leftarrow \lceil \log_2(2k+3) \rceil$
2.   $A \leftarrow \lfloor w/(\ell+1) \rfloor$
3.   $c \leftarrow -\sigma$
4.   While $c \le \sigma$ Do
5.       RangeEDSearch($P, T^1 \dots T^h, k, c, A, \ell$)
6.       $c \leftarrow c + A$
```

Fig. 6. Searching polyphonic text with weighted edit distance permitting any transposition.

We wanted also experiment on a Pentium IV due to its very optimized pipelining, which should compromize the overhead in number of register operations due to the bit-parallelism with the fact that we have broken the if-then-else nature of the original recurrences.

Table 2 shows that in the same setting ($w = 32$) with Pentium IV. This time we are faster even for $n = 10,000$, more precisely up to $n = 65,534$. This covers virtually all cases of interest.

7 Conclusions

In this paper we have focused on music retrieval. Since we wanted to apply the general string matching framework to this particular application domain, we introduced problems that are typical to music retrieval but that are not

Pentium IV	n =10	100	1,000	10,000
LCTS	0.00004644	0.006912	1.3797	137.97
CDP	0.000199	0.019929	1.9929	199.29

Table 2. Execution times for our LCTS and CDP when running on Pentium IV.

taken into account in the combinatorial pattern matching algorithms. The three distinguishing requirements 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 introduced a flexible and efficient bit-parallel algorithm that takes into account all the requirements above, and obtains a speedup of $O(w/\log k)$ over the classical algorithms, where the computer word has w bits and k is the error threshold allowed in the match. Even though it is not the best solution when unit cost distances are applied, it performs at a comparative level. Our algorithm, however, can be adapted to complex cases where no competing method exists. Furthermore, these cases are the most interesting in terms of music retrieval.

References

1. E. Cambouropoulos. A general pitch interval representation: Theory and applications. *Journal of New Music Research*, 25:231–251, 1996.
2. M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and W. Rytter. Finding motifs with gaps. In *First International Symposium on Music Information Retrieval (ISMIR'2000)*, Plymouth, MA, 2000.
3. M. Crochemore, C.S. Iliopoulos, G. Navarro, and Y. Pinzon. A bit-parallel suffix automaton approach for (δ, γ)-matching in music retrieval. To appear in *Proc. SPIRE 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. M.J. Dovey. A technique for "regular expression" style searching in polyphonic music. In *the 2nd Annual International Symposium on Music Information Retrieval (ISMIR'2001)*, pages 179–185, Bloomington, IND, October 2001.
7. J. Holub, C.S. Iliopoulos, and L. Mouchard. Distributed string matching using finite automata. *Journal of Automata, Languages and Combinatorics*, 6(2):191–204, 2001.
8. H. Hyyrö and G. Navarro. Faster bit-parallel approximate string matching. In *Proc. 13th Annual Symposium on Combinatorial Pattern Matching (CPM 2002)*, pages 203–224, 2002. LNCS 2373.
9. K. Lemström and P. Laine. Musical information retrieval using musical parameters. In *Proceedings of the 1998 International Computer Music Conference*, pages 341–348, Ann Arbor, MI, 1998.
10. K. Lemström and J. Tarhio. Transposition invariant pattern matching for multi-track strings. *Nordic Journal of Computing*, 2003. (to appear).

11. K. Lemström and E. Ukkonen. Including interval encoding into edit distance based music comparison and retrieval. In *Proceedings of the AISB'2000 Symposium on Creative & Cultural Aspects and Applications of AI & Cognitive Science*, pages 53–60, Birmingham, April 2000.
12. D. Meredith, K. Lemström, and G.A. Wiggins. Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music. *Journal of New Music Research*, 31(4):321–345, 2002.
13. MIDI Manufacturers Association, Los Angeles, California. *The Complete Detailed MIDI 1.0 Specification*, 1996.
14. G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999. Earlier version in *Proc. CPM'98*, LNCS 1448.
15. W. Paul and J. Simon. Decision trees and random access machines. In *Proc. Int'l. Symp. on Logic and Algorithmic*, pages 331–340, Zurich, 1980.
16. G.A. Wiggins, K. Lemström, and D. Meredith. SIA(M): A family of efficient algorithms for translation-invariant pattern matching in multidimensional datasets. (submitted).