Flexible Music Retrieval in Sublinear Time

Kimmo Fredriksson[1*], Veli Mäkinen[2†], Gonzalo Navarro[3‡]

[1] Dept. of Computer Science, University of Joensuu, Finland
e-mail: kfredrik@cs.joensuu.fi

[2] Technische Fakultät, Bielefeld Universität, Germany
e-mail: veli@cebitec.uni-bielefeld.de

[3] Dept. of Computer Science, University of Chile, Chile
e-mail: gnavarro@dcc.uchile.cl

Abstract. Music sequences can be treated as texts in order to perform music retrieval tasks on them. However, the text search problems that result from this modeling are unique to music retrieval. Up to date, several approaches derived from classical string matching have been proposed to cope with the new search problems, yet each problem had its own algorithms. In this paper we show that a technique recently developed for multipattern approximate string matching is flexible enough to be successfully extended to solve many different music retrieval problems, as well as combinations thereof not addressed before. We show that the resulting algorithms are close to optimal and much better than existing approaches in many practical cases.

Keywords: Music retrieval, approximate string matching, (δ, γ)-matching, transposition invariance.

1 Introduction

In this paper we are interested in music retrieval, and in particular, in a recent approach to it where musical scores are regarded as strings and string matching techniques can be used to solve music retrieval problems. In order to map the problem to string matching, the alphabet of the string could simply be the set of notes in the chromatic or diatonic notation, or the set of intervals that appear between notes (for example, pitches may be represented as MIDI numbers and pitch intervals as number of semitones). In both cases, we deal with *numeric* strings. Then, music retrieval problems can be converted into string matching problems, that is, find the occurrences of a short string (called the *pattern*) in a longer string (called the *text*). This is usually not enough to fully solve all music retrieval problems, but it provides a useful and efficient filter to leave the most promising candidates for a more profound and costly evaluation.

*Funded by the Academy of Finland, grant 202281.

†Funded by the Deutsche Forschungsgemeinschaft (BO 1910/1-3) within the Computer Science Action Program.

‡Partially funded by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile.

Exact string matching cannot be used to find occurrences of a particular melody, because a number of irrelevant distortions could exist between the melody sought and its version stored in the music database. To perform meaningful music retrieval one must resort to diverse forms of *approximate* matching, where a limited amount of *differences* of diverse kinds are permitted between the search pattern and its occurrence in the text. Different versions of the approximate string matching problem arise in different fields [23], yet those of music retrieval are unique of this area [10, 4, 27].

One approximate matching model of use in music retrieval is (δ, γ)*-matching*. In this model, two strings $a_1 a_2 \ldots a_m$ and $b_1 b_2 \ldots b_m$ of the same length m match if (i) the absolute differences between corresponding characters do not exceed δ, that is, $|a_i - b_i| \leq \delta$ for all $1 \leq i \leq m$ (or, alternatively, $\max_{1 \leq i \leq m} |a_i - b_i| \leq \delta$), and (ii) the sum of those absolute differences does not exceed γ, that is, $\sum_{1 \leq i \leq m} |a_i - b_i| \leq \gamma$. This model accounts for small differences that may arise between two versions of the same melody, setting a limit for the individual absolute differences, as well as a global limit to the overall differences. Searching for pattern p under (δ, γ)-matching consists of finding all the text positions where a text substring that (δ, γ)-matches p appears. Less popular subproblems are δ-matching and γ-matching, which only enforce one of the two conditions.

A second relevant approximate matching model is the *longest common subsequence (LCS)* and its dual *indel distance*. The former, $LCS(a, b)$, is the maximum length of a string that is subsequence both of a and b, that is, $LCS(a, b) = \max\{|s|,\ s \sqsubseteq a,\ s \sqsubseteq b\}$. A string $s = s_1 s_2 \ldots s_r$ is a *subsequence* of string $a_1 a_2 \ldots a_m$, $s \sqsubseteq a$, if s can be obtained by removing zero or more characters from a, that is, $s = a_{i_1} a_{i_2} \ldots a_{i_r}$ for $1 \leq i_1 < i_2 < \ldots < i_r \leq m$. The LCS has been largely used in computational biology to model biological similarity, and it is also relevant to identify musical passages that are similar except for a few extra or missing notes. This is especially relevant because music contains various kind of "decorations", such as grace notes and ornamentations, that are not essential for matching. The indel distance $id(a, b)$ between strings a and b is the number of characters one has to add or remove to a and b to make them equal, $id(a, b) = |a| + |b| - 2 \cdot LCS(a, b)$. Searching for pattern p under indel distance with tolerance k consists of finding all the text positions where a string p' appears so that $id(p, p') \leq k$. Other variants of indel distance, which are less popular in music retrieval, are Levenshtein or edit distance (where substitutions of characters are also permitted) and episode matching (where only insertions in the pattern are permitted).

Finally, a third similarity concept of relevance in music retrieval is *transposition invariance*. Two strings $a = a_1 a_2 \ldots a_m$ and $b = b_1 b_2 \ldots b_m$ are one the transposed version of the other if there is a constant t such that $a + t = (a_1 + t)(a_2 + t) \ldots (a_m + t) = b$. Transposition invariance is very relevant because Western people tend to listen to music analytically, by observing the intervals between consecutive pitch values rather than the actual pitch values themselves. As a result, a melody performed in two distinct pitch levels is perceived as equal regardless of whether it is performed in a lower or higher level of pitches.

As a string matching problem, dealing with transposition invariance is trivial because it suffices to represent text and pattern as differences between consecutive notes and then apply exact string matching. However, the above problems in most cases of interest appear in combined form. In particular, transposition invariance is usually combined with longest common subsequence. The longest common transposition invariant subsequence between two strings a and b, $LCTS(a, b)$, permits transposing a or b as necessary to find the longest common subsequence among them, $LCTS(a, b) = \max_{t \in \mathbb{Z}} LCS(a + t, b)$.

In recent years, there has been much activity around developing specific string matching techniques to solve diverse music retrieval problems, mostly consisting of combinations of those outlined above. Several theoretical and practical results of interest have been achieved. We cover these in the next section.

Our contribution in this paper is to show that a particular approach recently developed for multiple approximate string matching [16] is flexible enough to be successfully adapted to solve most of the combinations of problems sketched above. Basically the same search technique, coupled with slightly different pattern preprocessings, yield algorithms that solve each combination. We also characterize those combinations that cannot be addressed by our approach. In theoretical terms, we show that the resulting algorithms are sublinear (that is, they do not inspect all text characters) and can be argued to be close to optimal. Yet, the most important aspect is the practical side, where we show that our technique largely outperforms all the existing ones in most cases of interest.

2 Related Work

In which follows, we assume that a long text $T = t_1 t_2 \ldots t_n$ is searched for a comparatively short pattern $p = p_1 p_2 \ldots p_m$. Both are sequences over alphabet Σ, a finite contiguous subset of $\mathbb{Z}$, of size σ.

2.1 (δ, γ)-Matching

Several recent algorithms exist to solve this problem. These can be classified as follows:

Bit-parallel: The idea is to take advantage of the intrinsic parallelism of the bit operations inside a computer word of w bits [26], so as to pack several values in a single word and manage to update them all in one step [5, 6, 12]. The best complexity achieved [12] is $O(n\ m \log(\gamma)/w)$ in the worst case and $O(n)$ on average.

Occurrence heuristics: Inspired by Boyer-Moore techniques [3], they skip some text characters according to the position of some characters in the pattern [5, 11]. In general, only δ is used to skip characters, while the γ-condition is used to verify candidates. This makes these algorithms weak for large δ and small γ.

Substring heuristics: Based on suffix automata [14], these algorithms skip text characters according to the position of some pattern substrings [11, 12]. In the second article, they use bit-parallelism to filter the text using both δ and γ, unlike previous approaches. This is shown to be the approach examining the least number of text characters.

FFT-related: It is possible to solve the problem in $O(\delta n \log m)$ time [7] using Fast Fourier Transform (FFT) based techniques. Other FFT based $o(mn)$ solutions exist for related problems, see e.g. [8] and especially related to δ-matching [1, 9]. Matching under γ-restriction is possible in $O(mn/\log_\sigma n)$ time [21] without using FFT (but using the Four-Russians trick).

In practice, the best current algorithms for (δ, γ)-matching are those in [12], as demonstrated by the experiments in [11, 12]. In [12] they present a plain bit-parallel and a substring heuristic. The first is shown to be the best in most cases, but for short patterns and small δ and γ, the character-skipping technique is better.

The FFT based techniques, although elegant, have considerably large overheads to make them practical. Our preliminary tests show that they only become faster than the naive algorithm on very long patterns. Searching for long patterns is not typical in music retrieval. The solution based on the Four-Russians trick is only practical for small alphabets, much smaller than what is required for music retrieval.

2.2 Transposition Invariant LCS and Indel Distance

Plain (non-transposed) LCS among strings p and T can be computed in $O(mn)$ time using dynamic programming [17]. In general, any LCTS algorithm can be adapted to text searching with indel distance. The LCTS problem was first stated in [20], where $O(\sigma mn)$ time was obtained by trying out all the $2\sigma + 1$ possible transpositions one by one. Further solutions to the problem can be classified as follows.

Brute-force: The idea is to pick any LCS algorithm and try it for all the $2\sigma + 1$ possible transpositions. Apart from the original proposal [20], several others have been attempted considering different practical LCS algorithms based on bit-parallelism [13, 18]. The best complexity achieved is $O(\sigma mn/w)$.

Sparse dynamic programming: An evolution over the above scheme is to notice that the $LCS(a + t, b)$ problem for each transposition t has only a few character matches between a and b, mn in total. Those *sparse* problems are best handled by sparse dynamic programming algorithms. This idea lead to several solutions [22, 25, 15]. The best complexity achieved is $O(mn \log\log \min(m, \sigma))$, yet a version with complexity $O(mn \log \sigma / \log w)$ is shown to be better in practice.

Branch and bound: In this case the idea is to search for the best possible transposition t by a backtracking method, recursively dividing the space of $2\sigma + 1$ transpositions into ranges until finding the best one [19]. This yields a best-case complexity of $O((mn + \log\log \sigma) \log \sigma)$, and the method works well in practice. Yet, it cannot be extended to searching with indel distance.

Experiments in [19, 18, 15] demonstrate that the $O(mn \log \sigma / \log w)$ algorithm in [15] is the fastest in practice. This method can be adapted to searching with indel distance.

3 Optimal Multiple Approximate String Matching

In [16], new algorithms for single and multiple approximate string matching were presented. Those algorithms were not only optimal on average, but also very efficient in practice, even in the more competitive area of single approximate string matching. It was shown that, to search for the occurrences of r patterns of length m in a text of length n, all them uniformly distributed over an alphabet of size σ, the algorithm required $O(n(k+\log_\sigma(rm))/m)$ time on average. Here k is the maximum number of missing, extra, or substituted characters permitted to match a pattern against a text string (searching under edit distance). This average complexity is optimal [28, 24].

We first explain how to search for a single pattern p. We first choose a *block length* ℓ, and compute $med(b,p)$ for every possible block $b \in \Sigma^\ell$ (that is, every possible ℓ-*gram*). Here, $med(b,p)$ is the minimum edit distance between b and a substring of p,

$$med(b,p) \quad = \quad \min\{ed(b,p'),\ \exists x,y,\ p = xp'y\},$$

being $ed(b,p')$ the edit distance between b and p'.

Now, the text $T = t_1t_2\ldots t_n$ is scanned as follows. Since the minimum length of an occurrence of $p = p_1p_2\ldots p_m$ in T with edit distance at most k has length at least $m-k$ (when k deletions occur on p), we slide a window of length $m-k$ along the text. For each window tried, $t_{i+1}t_{i+2}\ldots t_{i+m-k}$, we read its ℓ-grams right to left. That is, we read at most $\lfloor (m-k)/\ell \rfloor$ ℓ-grams b_1, b_2, and so on, so that $b_1 = t_{i+m-k-\ell+1}\ldots t_{i+m-k}$ is the rightmost, $b_2 = t_{i+m-k-2\ell+1}\ldots t_{i+m-k-\ell}$ precedes b_1, etc. The invariant is that any occurrence of p starting at positions $\leq i$ has already been reported.

For each such ℓ-gram $b_j = t_{i+m-k-j\ell+1}\ldots t_{i+m-k-j\ell+\ell}$, we find $med(b_j,p)$ in the precomputed table. If, after reading b_j, we have $med(b_1,p) + med(b_2,p) + \ldots + med(b_j,p) > k$, then no possible occurrence of p can contain the text $b_jb_{j-1}\ldots b_2b_1$, thus the window is slid forward to start at the second character of b_j, that is, we set $i \leftarrow i+m-k-j\ell+1$ (as the new window will start at $i+1$).

If, on the other hand, all the ℓ-grams of the window are scanned and yet the window cannot be shifted, then it must be verified for a real occurrence. At this point, we must check if there is an occurrence p' of p starting at text position $i+1$. Since the maximum length of an occurrence is $m+k$ (where k insertions occur into p), any potential p' mush finish between positions $i+m-k$ and $i+m+k$. So we compute

$$led(p,i) \quad = \quad \min\{ed(p, t_{i+1}\ldots t_{i+m-k+d}),\ 0 \leq d \leq 2k\},$$

which can be done in $O(m^2)$ time by computing $ed(\,)$ incrementally in d. If $led(p,i) \leq k$, we report $i+1$ as the starting position of an occurrence. Finally, we advance the window by one position, $i \leftarrow i+1$.

We show now that the way we shift the window is safe, that is, no occurrence can start at positions $i+1$ to $i+m-k-j\ell+1$. Any such occurrence, of length at least $m-k$, must contain the sequence of ℓ-grams $b_j\ldots b_1$. Let $p' = xb_j\ldots b_1y$ be such an occurrence. This is a split of p' into $j+2$ pieces. The main point is that the edit distance is *decomposable*: For any strings p and p', given any split $p' = p'_1\ldots p'_{j+2}$, there is a split $p = p_1\ldots p_{j+2}$ such that $ed(p',p) = ed(p'_1,p_1) + \ldots + ed(p'_{j+2},p_{j+2})$. But each such $ed(p'_s,p_s) \geq med(p'_s,p) \geq 0$, by definition of $med(\,)$.

Hence, in our particular case, $ed(p',p) \geq med(b_j,p) + \ldots + med(b_1,p)$. Thus if the latter exceeds k, there can be no occurrence of p containing $b_j\ldots b_1$.

The extension of the algorithm for multiple patterns is trivial. We only have to change the preprocessing so that p is now a set of patterns $p = \{p^1\ldots p^r\}$ and now $med(b,p) = \min_{1\leq i\leq r} med(b,p^i)$. So $med(b,p)$ is a lower bound to the cost of matching b anywhere inside any pattern of the set.

By appropriately choosing $\ell = \Theta(\log_\sigma(rm))$, we obtain the promised complexity.

3.1 Extensions

Several other improvements are studied in [16]. We briefly review some that are used in our experiments. For more details see [16].

On the windows that have to be verified, we could simply run the verification for every pattern, one by one. A more sophisticated choice is *hierarchical verification* [2]. We form a tree whose nodes have the form $[i, j]$ and represent the group of patterns $p^i \dots p^j$. The root is $[1, r]$, and the leaves have the form $[i, i]$. Every internal node $[i, j]$ has two children $[i, \lfloor (i + j)/2 \rfloor]$ and $[\lfloor (i + j)/2 \rfloor + 1, j]$.

The preprocessing is done first for the leaves, as in the single pattern case, that is, we compute a table for $med(b, p^i)$. The internal nodes contain tables for $\min_{i \le h \le j} med(b, p^h)$, computed as minimizing over the two tables of the subtrees. In the filtering phase, we first use the table for the root, corresponding to the full set of patterns, and if the current window has to be verified with respect to a node in the hierarchy, we rescan the window considering the two children of the current node. It is possible that the window can be discarded for both children, for one, or for none. We recursively repeat the process for every child that does not permit discarding the window. If we process a leaf node and still have to verify the window, then we run the verification algorithm for the corresponding single pattern.

The second improvement is to have *bit-parallel counters*. In this case we reserve only $O(\log_2 k)$ bits to accumulate the differences $med(b_j, p)$. This means that if we have a computer word of w bits, we can process $O(w/\log_2 k)$ patterns in parallel. This technique can also be used with the hierarchical verification, to increase the arity of the tree to $O(w/\log_2 k)$.

The third improvement is to use *ordered ℓ-grams*, where each b_j is permitted to match only in the area of p where it could be aligned in an occurrence starting at $i + 1$. In an approximate occurrence of $b_j \dots b_1$ inside the pattern, b_i cannot be closer than $(i - 1)\ell$ positions to the end of the pattern. Therefore, we compute tables for $med^j(b, p)$, $1 \le j \le \lfloor (m - k)/\ell \rfloor$, where $med^j(b, p) = \min\{ed(b, p'),\ \exists x, y,\ |y| \ge (j - 1)\ell,\ p = xp'y\}$. This allows us to discard a window whenever $med^1(b_1, p) + med^2(b_2, p) + \dots + med^j(b_j, p) > k$. This reduces verifications but increases preprocessing time and space.

Finally, it is possible to improve the preprocessing time by using a trie of all the possible ℓ-grams to reuse preprocessing work. All the improvements can be combined into a single algorithm.

4 Adapting to Music Retrieval

The method above was designed for multiple string matching under edit distance. Yet its main idea is much more general and can be used to solve many other problems. In this section we demonstrate that the idea solves most of the music retrieval problems we have focused on in this paper. We note that this gives immediately a solution to the multipattern version of the same problems.

4.1 Transposition Invariant Indel Distance

Let us start with searching with transposition invariant indel distance. For each ℓ-gram $b \in \Sigma^\ell$, we compute

$$mtid(b,p) \quad = \quad \min\{id(b+t,p'),\ \exists x,y,\ p=xp'y,\ -\sigma \le t \le \sigma\}.$$

This is the minimum transposition invariant indel distance to match b anywhere inside p. The same algorithm of the previous section is used, and the same argument shows that we cannot discard a window that starts an occurrence of p in T. Indel distance is decomposable just like edit distance, that is, for any split $p' = p'_1 \dots p'_{j+2}$, there is a split $p = p_1 \dots p_{j+2}$ such that $id(p',p) = id(p'_1,p_1) + \dots + ed(p'_{j+2},p_{j+2})$. Assume p matches t the current window $xb_j \dots b_1 y$ starting at position $i+1$. That is, there exists a transposition t such that $id(p',p) \le k$, $p' = (x+t)(b_j+t)\dots(b_1+t)(y+t)$. Now, $id(p',p) \ge id(b_j+t,p_2) + \dots id(b_1+t,p_{j+1}) \ge mtid(b_j,p) + \dots mtid(b_1,p)$. Thus if the latter exceeds k we can safely shift the window.

When a window starting at position $i+1$ cannot be shifted, we simply compute $LCTS(p, t_{i+1} \dots t_{i+m-k+d})$ for any $0 \le d \le 2k$, and report position $i+1$ if $LCTS(p, t_{i+1} \dots t_{i+m-k+d}) \ge (m+m-k+d-k)/2 = m-k+d/2$ for some d, as this is equivalent to $id(p, t_{i+1} \dots t_{i+m-k+d}) \le k$ for some transposition t.

Fig. 1 shows simplified pseudocode.

4.2 (δ,γ)-Matching

Alternatively, we can search for (δ,γ)-matches of p in T. In this case the window is of length m, as occurrences are all of the same length. For each ℓ-gram $b \in \Sigma^\ell$, we compute

$$mdg(b,p) \quad = \quad \min\{\gamma',\ \exists x,y,\ p=xp'y,\ b\ (\delta,\gamma')\text{-matches } p'\}.$$

This is the minimum total number of absolute differences obtained by b inside p, where we restrict those positions to δ-match as well. The same algorithm of the previous section is used with this preprocessing (and the threshold is γ instead of k).

Being γ-matching a cumulative measure, the sum of $mdg(b_j,p)$ values is a lower bound to the γ needed to match the window inside p. Consider window $p' = t_{i+1} \dots t_{i+m} = xb_j \dots b_1$. Assume p' (δ,γ)-matches p. Then, by definition of (δ,γ)-matching, b_1 (δ,γ_1)-matches $p_{m-\ell+1} \dots p_m$, and so on until b_j, which (δ,γ_j)-matches $p_{m-j\ell+1} \dots p_{m-j\ell+\ell}$, so that $\gamma_1 + \dots + \gamma_j \le \gamma$. As each b_s (δ,γ_s)-matches $p_{m-s\ell+1} \dots p_{m-s\ell+\ell}$, it holds $mdg(b_s,p) \le \gamma_s$, and thus $mdg(b_j,p) + \dots + mdg(b_1,p) \le k$.

When a window $t_{i+1} \dots t_{i+m}$ cannot be shifted, we check whether p (δ,γ)-matches the window in time $O(m)$, and report position $i+1$ if this is the case.

The pseudocode of Fig. 1 can be easily adapted to this model. One needs only to replace $mtid()$ with $mdg()$, k with γ, and adjust the window size from $m-k$ to m, and verification area from $t_{i+1} \dots t_{i+m+k}$ to $t_{i+1} \dots t_m$.

4.3 Feasible and Unfeasible Combinations

We can also combine transposition invariant indel distance with δ-matching. In this case we count indels, but two characters match whenever they do not differ by more

```
Search ( )
    D ← Preprocess ( )
    i ← 0
    While i ≤ n − (m − k) Do
        pos ← Shift (i, D)
        If pos = i Then
            Run verification in text area t_{i+1} ... t_{i+m+k}
            pos ← pos + 1
        i ← pos

Shift (i, D)
    M ← 0
    c ← m − k
    While c ≥ ℓ Do
        c ← c − ℓ
        M ← M + D[t_{i+c+1} ... t_{i+c+ℓ}]
        If M > k Then Return i + c + 1
    Return i

Preprocess ( )
    ℓ ← Θ(log_σ m)
    For b ∈ Σ^ℓ Do D[b] ← mtid(b, p)
    Return D
```

Figure 1: Simple description of the algorithm. The main variables are global for all the algorithms, to simplify the presentation. The code corresponds to transposition invariant indel.

than δ units. This is easily handled by modifying $mtid(b,p)$ formula so that $id(b+t,p')$ considers matches in the more relaxed way. Transposition invariance can also be combined with (δ,γ)-matching, by using $mtdg(b,p)$ instead of $mdg(b,p)$, so that

$$mtdg(b,p) \quad = \quad \min\{\gamma',\ \exists x,y,\ p = xp'y,\ b+t\ (\delta,\gamma')\text{-matches } p',\ \ -\sigma \le t \le \sigma\}.$$

Interestingly, we cannot directly combine transposition invariant indel distance with (δ,γ)-matching. The reason is that we do not have here a single value to minimize, such as the number of indels or γ, but both of them at the same time. It was possible to combine transposition invariant indel distance with δ-matching because the latter is not a parameter to optimize but a condition for matching. Likewise, it was possible to combine γ-matching with δ-matching to obtain (δ,γ)-matching. Yet, if we want to combine indel distance (even without transposition invariance) with γ-matching, the problem is that each pair (b,p') produces some number of indels and some γ, so different pairs will yield the minimal of each and it is not clear which to choose.

Of course we can count indels and γ separately in different tables (each achieved by a different pair). This is equivalent to filtering each window with k and with γ separately, and verifying those that pass both filters. Yet, this is not the same as a combined filter, but it could be practical.

4.4 Complexity Considerations

We are not able to analyze our algorithms, but we can give some clues about their average case performance. As we have described it, our algorithm for transposition invariant indel distance is equivalent to multipattern search with indel distance for the set $p^1 = p-\sigma$, $p^2 = p-\sigma+1$, ..., $p^{2\sigma+1} = p+\sigma$. Since $id(a,b) \geq ed(a,b)$ for any strings a and b, we can use the analysis of [16] on edit distance for indel distance and the result is pessimistic (yet tight). According to that analysis, searching for $r = 2\sigma + 1$ random patterns in random text yields average complexity $O(n(k + \log_\sigma(rm))/m) = O(n(k + \log_\sigma m)/m)$. This value is optimal even for one pattern [28], and it would show that our algorithm is optimal too.

Yet, the problem is that our $2\sigma + 1$ patterns are *not* random, but are all the transpositions of a random pattern. For example, if $\ell = 1$, then our $2\sigma + 1$ patterns necessarily match any string of length 1, whereas the same number of random patterns do not. Thus our analysis is optimistic and therefore not conclusive. Yet, we conjecture that the result of the analysis is valid.

In case δ-matching is permitted together with transposition invariance indel distance in the model, then the probability of matching is not $1/\sigma$ but $O(\delta/\sigma)$, and therefore the base of the logarithm is not σ but $O(\sigma/\delta)$. Redoing the analysis we get $O(n(k+\log_{\sigma/\delta}(\delta m))/m)$. With δ-matching alone (no transposition invariance) we get $O(n \log_{\sigma/\delta} m/m)$, and with δ-matching with transposition invariance (without indels) we get $O(n \log_{\sigma/\delta}(\delta m)/m)$. We are not able to account for the analytical effect of a γ-restriction in these analyses, but of course they can only improve.

The preprocessing time is $O(m\sigma^{\ell+1}/w)$ for transposition invariant indels, $O(m\sigma^\ell)$ for (δ,γ)-matching, and $O(m\sigma^{\ell+1})$ for transposition invariant (δ,γ)-matching. With ordered ℓ-grams the preprocessing cost for indels increases to $O(m\sigma^{\ell+1})$. For the other models the costs remain the same. The space requirement is $O(\sigma^\ell)$ and $O(\sigma^\ell m/\ell)$ for the basic algorithm and for the ordered ℓ-grams, respectively. These have to be multiplied by $O(\sigma)$ if hierarchical verification is used. All the bounds are polynomial in m (as $\ell = \Theta(\log_\sigma m)$).

5 Experimental Results

We have implemented the algorithms in C, compiled using `icc 8.0` with full optimizations. The experiments were run in a 2GHz Pentium 4, with 512MB RAM, running Linux 2.4.18. The computer word length is $w = 32$ bits.

For the text we used a concatenation of 7543 music pieces, whose total length is 1828089 bytes. The file was obtained by extracting the pitch values from MIDI files. The pitch values are in the range $[0 \ldots 127]$. A set of 100 patterns were randomly extracted from the text. Each pattern was then searched for separately, and we report the average search times. We measured user times. We have separated the preprocessing and search times, which makes it easier to compare the search performance. Our preprocessing cost is considerably high, but this is amortized by large music collections that arise in practical applications.

5.1 Implementation

Several variants of the optimal multipattern algorithm were considered in [16]. For (δ, γ)-matching without transpositions, we used the basic single pattern algorithm. As the transpositions were implemented as multipattern search, we used bit-parallel counters and hierarchical verification in these cases, which give a considerable speed-up. For indels, we used the IndelMYE algorithm [18] for the final verifications. We ran each experiment with and without ordered ℓ-grams. The former is an order of magnitude faster in many cases, but it has higher preprocessing cost, justified only for large texts.

For all experiments we used $\ell = 2$. Due to the considerably large alphabet size, larger ℓ values were not practical. On the other hand, $\ell = 1$ gives in general poor results, especially combined with transpositions (but note that with bit-parallel counters even 1-grams are not guaranteed to match always, as different transposition ranges are mapped to different counters).

As the alphabet size was large (128), but most of the values occur in the middle of the range, we mapped the alphabet into the range $0 \dots 63$. That is, values $32 \dots 95$ were mapped to $0 \dots 63$, values $0 \dots 31$ to 0, and values $96 \dots 127$ to 95. This mapping allows us to use the original δ values. Verification was done using the original alphabet. This improves the preprocessing times, without worsening the search times.

5.2 Preprocessing Time

Table 1 gives the preprocessing times. For $mtid()$ and $mtdg()$ we have considered hierarchical verification because it gave consistently better results, so the preprocessing timings include all the hierarchy construction. Using ordered ℓ-grams increases the preprocessing cost, but improves the search performance.

$mtid()$, $m = 32$	$mdg()$, $m = 8$	$mdg()$, $m = 64$	$mtdg()$, $m = 32$
0.0699 / 0.2680	0.0048 / 0.0052	0.0067 / 0.0092	0.0936 / 0.5177

Table 1: Preprocessing times in seconds for $\ell = 2$. The second timings are for ordered ℓ-grams.

5.3 Transposition Invariant Indel Distance

We compared our approach against the LCTS algorithm [15], whose running time is $O(mn \log \sigma / \log w)$. Although the algorithm solves the dual problem, it could be adapted to searching with indel distance as well. We also compared against the bit-parallel dynamic programming algorithm IndelMYE [18], whose running time for a single transposition is $O(mn/w)$. We superimposed [2] all the transpositioned patterns and used hierarchical verification, in the same manner as in [16] with BPM algorithm. This works very well in practice, although the worst case complexity is still $O(\sigma mn/w)$. Fig. 2 shows the results for $m = 8 \dots 64$ and $k = 1 \dots 5$. Our algorithm is by far the fastest for small k/m. LCTS is competitive only for very large k/m, while IndelMYE is the best choice for moderate k/m. Our algorithm clearly improves with ordered ℓ-grams, at the cost of higher preprocessing effort and memory requirements.

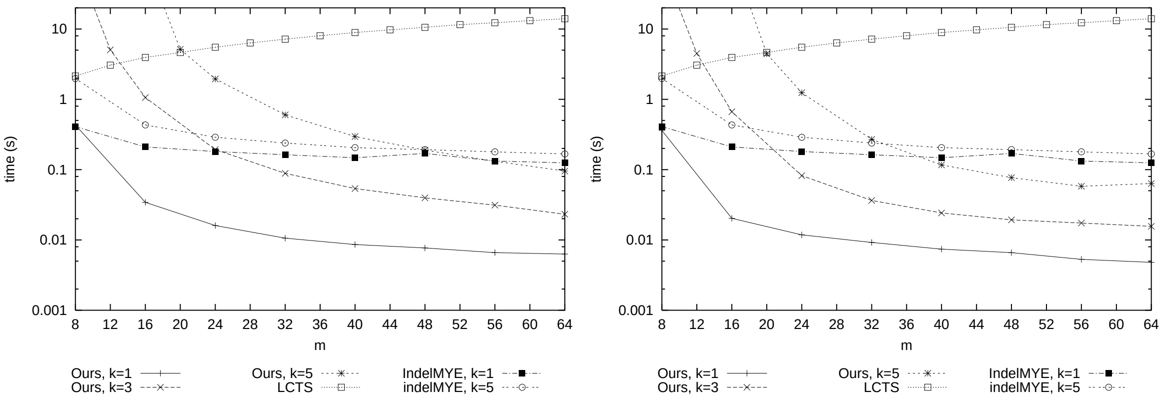

Figure 2: Left: Search time in seconds for transposition invariant indel/LCS for $m = 8...64$. Right: The same with ordered ℓ-grams.

Fig. 3 shows the results for $m = 32$, $k = 1 \ldots 6$ and $\delta = 0 \ldots 2$. The LCTS algorithm cannot be applied for this setting. Being bit-parallel algorithm, IndelMYE can be easily adapted to this case by using classes of characters to implement δ. In this case we are again competitive against IndelMYE for small k/m, but only for very small δ. Ordered ℓ-grams boost the search considerably.

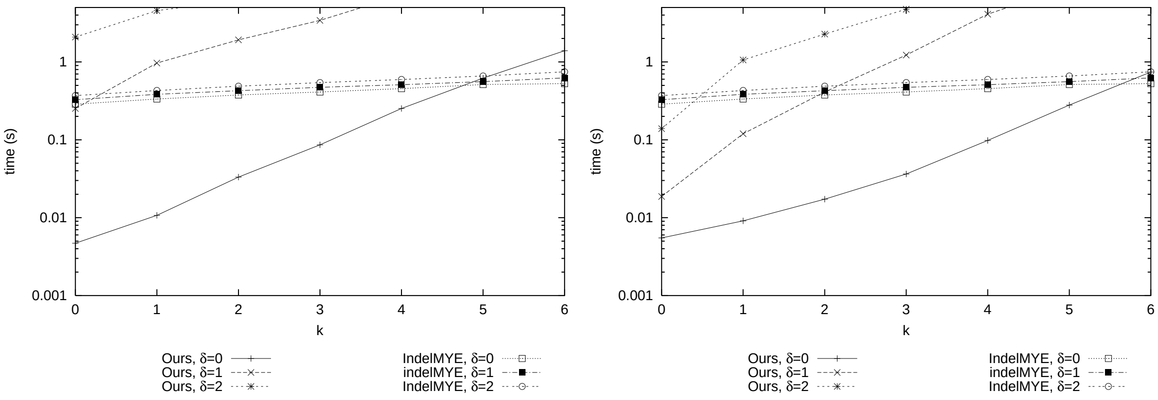

Figure 3: Left: Search times in seconds for transposition invariant indel for $\delta = 1...3$, and $m = 32$. Right: The same with ordered ℓ-grams.

5.4 (δ, γ)-Matching

For (δ, γ)-matching we compared against the bit-parallel Forward matching algorithm (Fwd) of [12]. Fig. 4 shows the results for $m = 8 \ldots 64$, $\delta = 1 \ldots 3$ and $\gamma = m\delta/2$. Our algorithm is much more sensitive to increasing δ than Fwd, but for small δ values we are an order of magnitude faster. Using ordered ℓ-grams makes our algorithm more tolerant for increasing γ (but note that γ/m is constant here).

In [12] they give also bit-parallel backward matching algorithm, that is able to skip some text characters. The implementation restricts the pattern lengths to be at most $\Theta(w/\log_2(\gamma))$. This means that in this experiment this algorithm is applicable only for the case $m = 8$, $\delta = 1$, and $\gamma = 8 * 1/2 = 4$. The algorithm takes 0.0063s average time, in this case, and marginally beats our algorithm (0.0065s)

Timings for $m = 32$, $\delta = 1 \ldots 3$, and $\gamma = 4 \ldots 40$ are shown in Fig. 5. (Note that for $\delta = 1$ there is no point for using $\gamma > m$.) Again, Fwd becomes eventually faster for large δ and γ, while our algorithm dominates for small parameter values. Fig. 6 repeats the experiment for transposition invariant (δ, γ)-matching. Note that no competitors exist in this case, although transposition superimposition and hierarchical verification could be applied for some of the existing (δ, γ) matching algorithms. However, observe that our transposition invariant algorithm is faster than Fwd algorithm (without transpositions) for small δ and γ.

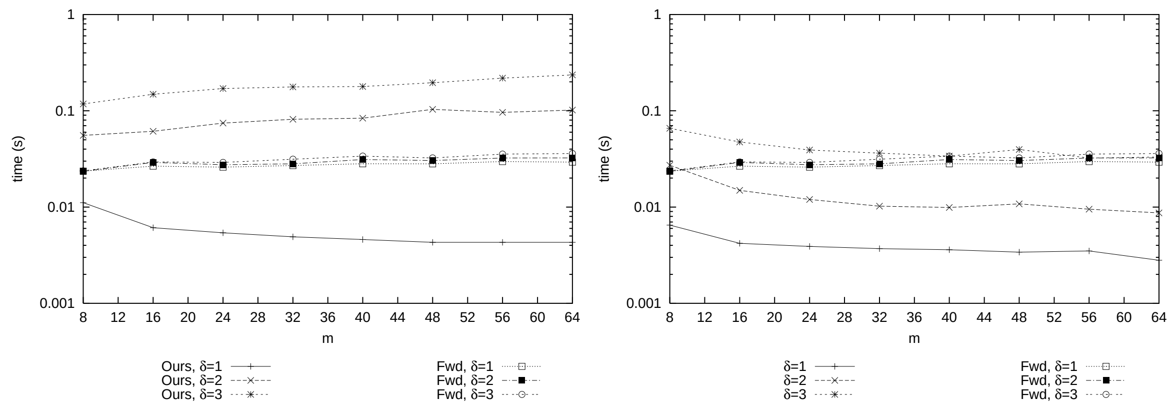

Figure 4: Left: Search times in seconds for (δ, γ)-matching for $m = 8 \ldots 64$ and $\delta = 1 \ldots 3$. For each data point $\gamma = m\delta/2$. Right: The same with ordered ℓ-grams.

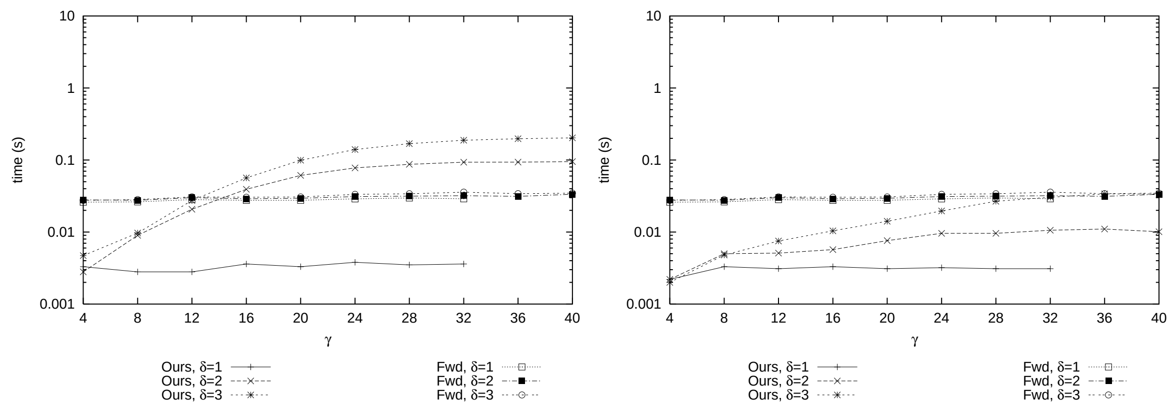

Figure 5: Left: Search times in seconds for (δ, γ)-matching for $m = 32$, $\delta = 1 \ldots 3$, and $\gamma = 4 \ldots 40$. Right: The same with ordered ℓ-grams.

5.5 Comparison

We have separated the preprocessing and searching times in presenting the experimental results. This may seem unfair against the competing algorithms, and so it is for short texts. To show that our algorithms *are* competitive, Table 2 gives estimates

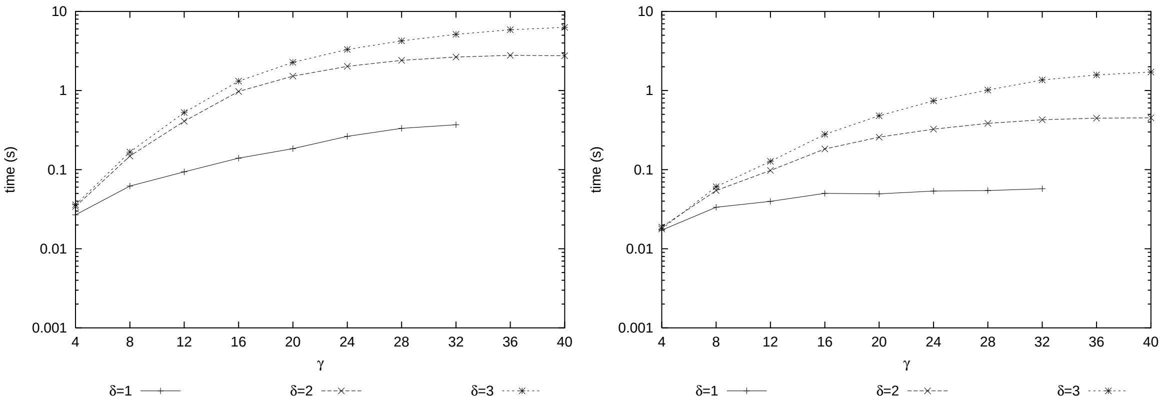

Figure 6: Left: Search times in seconds for (δ, γ)-matching with transpositions for $m = 32$, $\delta = 1 \ldots 3$, and $\gamma = 4 \ldots 40$. Right: The same with ordered ℓ-grams.

for the minimum file sizes required to beat the competing approaches for various problem instances. These limits are quite modest, and for smaller parameter values even shorter files are sufficient.

Indels	
$k = 4, \delta = 0$	$k = 1, \delta = 1$
> 0.61 Mb	> 1.77 Mb

(δ, γ)-matching		
$(1, \infty)$	$(2, \infty)$	$(3, 24)$
> 0.46 Mb	> 0.71 Mb	> 1.52 Mb

Table 2: Examples of music file sizes where we begin to win for a few settings. The first row shows the parameter values, and the second row gives an estimate of the minimum file size where our algorithm wins its competitor. For smaller parameters shorter files would suffice. The estimates are for $m = 32$.

6 Conclusions

We have presented new filtering algorithms for music retrieval. Our algorithms are very efficient in practice, and are conjectured to be optimal on average. The experiments show that for small to moderate error thresholds our algorithms are substantially faster than previous approaches for all but very short texts. These are the parameter values that are most interesting in most music retrieval applications.

The algorithms are extremely flexible. We can solve many different problem variants essentially without any modifications to the search algorithms, only preprocessing changes according to the search model. In particular, we are able to solve some variants where no competing algorithms currently exist. These are transposition invariant indel with $\delta > 0$, and transposition invariant (δ, γ)-matching. Moreover, our algorithms can be used for multipattern search as well.

References

[1] A. Amir and M. Farach. Efficient 2-dimensional approximate matching of half-rectangular figures. *Information and Computation*, 118(1):1–11, 1995.

[2] R. Baeza-Yates and G. Navarro. New and faster filters for multiple approximate string matching. *Random Structures and Algorithms (RSA)*, 20:23–49, 2002.

[3] R. Boyer and J. Moore. A fast string searching algorithm. *Comm. of the ACM*, 20(10):762–772, 1977.

[4] E. Cambouropoulos, T. Crawford, and C. Iliopoulos. Pattern processing in melodic sequences: Challenges, caveats and prospects. In *Proc. Artificial Intelligence and Simulation of Behaviour (AISB'99) Convention*, pages 42–47, 1999.

[5] E. Cambouropoulos, M. Crochemore, C. Iliopoulos, L. Mouchard, and Y. J. Pinzon. Algorithms for computing approximate repetitions in musical sequences. In *Proc. 10th Australasian Workshop on Combinatorial Algorithms (AWOCA'99)*, pages 129–144, 1999.

[6] E. Cambouropoulos, M. Crochemore, C. S. Iliopoulos, L. Mouchard, and Y. J. Pinzon. Algorithms for computing approximate repetitions in musical sequences. *Journal of Computational Mathematics*, 79(11):1135–1148, 2002.

[7] P. Clifford, R. Clifford, and C. Iliopuolos. Faster algorithms for δ, γ-matching and related problems. In *Proc. 16th Ann. Symp. on Combinatorial Pattern Matching (CPM 2005)*, 2005. To appear.

[8] R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proc. Ann. ACM Symp. on Theory of Computing (STOC'02)*, pages 592–601, 2002.

[9] R. Cole, C. Iliopoulos, T. Lecroq, W. Plandowski, and W. Rytter. On special families of morpishms related to δ-matching and don't care symbols. *Information Processing Letters*, 85(5):227–233, 2003.

[10] T. Crawford, C. Iliopoulos, and R. Raman. String matching techniques for musical similarity and melodic recognition. *Computing in Musicology*, 11:73–100, 1998.

[11] M. Crochemore, C. Iliopoulos, T. Lecroq, Y. J. Pinzon, W. Plandowski, and W. Rytter. Occurence and substring heuristics for δ-matching. *Fundamenta Informaticae*, 55:1–15, 2003.

[12] M. Crochemore, C. Iliopoulos, G. Navarro, Y. Pinzon, and A. Salinger. Bit-parallel (δ, γ)-matching suffix automata. *Journal of Discrete Algorithms*, 2004. To appear. Conference version in *Proc. SPIRE'03*, LNCS 2857, pages 211–223.

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

[14] M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.

[15] S. Deorowicz. Speeding up transposition invariant string matching. Technical report, Institute of Computer Science, Silesian University of Technology, Poland, 2005. `http://www-zo.iinf.polsl.gliwice.pl/ sdeor/pub/deo05babs.htm`.

[16] K. Fredriksson and G. Navarro. Average-optimal single and multiple approximate string matching. *ACM Journal of Experimental Algorithmics*, 9(1.4), 2004.

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

[18] H. Hyyrö, Y. Pinzon, and A. Shinohara. New bit-parallel algorithm for approximate string matching under indel distance. In *Proc. 4th International Workshop on Experimental and Efficient Algorithms (WEA 2005)*, LNCS, 2005. To appear.

[19] K. Lemström, G. Navarro, and Y. Pinzon. Practical algorithms for transposition-invariant string-matching. *Journal of Discrete Algorithms*, 2004. To appear. Conference version in *Proc. SPIRE'03*, LNCS 2857, pages 224–237.

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

[21] V. Mäkinen. Sub-quadratic algorithm for weighted k-mismatches problem. Technical Report C-2004-1, Dept. of Computer Science, Univ. of Helsinki, 2004. `http://www.cs.helsinki.fi/u/vmakinen/papers/weightedkmm.ps.gz`.

[22] V. Mäkinen, G. Navarro, and E. Ukkonen. Transposition invariant string matching. *Journal of Algorithms*, 2004. To appear. Conference version in *Proc. STACS'03*, LNCS 2607, pages 191–202.

[23] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[24] G. Navarro and K. Fredriksson. Average complexity of exact and approximate multiple string matching. *Theoretical Computer Science*, 321(2–3):283–290, 2004.

[25] G. Navarro, Sz. Grabowski, V. Mäkinen, and S. Deorowicz. Improved time and space complexities for transposition invariant string matching. Technical Report TR/DCC-2005-4, Dept. of Computer Science, Univ. of Chile, 2005. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/mnloglogs.ps.gz`.

[26] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[27] P. Roland and J. Ganascia. Musical pattern extraction and similarity assessment. In E. Miranda, editor, *Readings in Music and Artificial Intelligence*, pages 115–144. Harwood Academic Publishers, 2000.

[28] A. C. Yao. The complexity of pattern matching for a random string. *SIAM Journal of Computing*, 8(3):368–387, 1979.