Nordic Journal of Computing

TRANSPOSITION INVARIANT PATTERN MATCHING FOR MULTI-TRACK STRINGS

KJELL LEMSTRÖM
University of Helsinki, Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, Finland
Kjell.Lemstrom@cs.Helsinki.Fi

JORMA TARHIO
Helsinki University of Technology, Dept. of Computer Science and Engineering
P.O. Box 5400, FIN-02015 HUT, Finland
Jorma.Tarhio@hut.Fi

Abstract. We consider the problem of *multi-track string matching*. The task is to find the occurrences of a pattern across parallel strings. Given a set S of h strings $s^i = s^i_1 \cdots s^i_n, i \in \{1, \ldots, h\}$, a pattern $p = p_1 \cdots p_m$ has such an occurrence at position j of S if $p_1 = s^{i_1}_j, p_2 = s^{i_2}_{j+1}, \ldots, p_m = s^{i_m}_{j+m-1}$ holds where i_j is in $\{1, \ldots, h\}$ for $j = 1, \ldots, m$. An application of the problem is music retrieval where occurrences of a monophonic query pattern are searched in a polyphonic music database. In music retrieval it is even more pertinent to allow invariance for pitch level transpositions, i.e., the task is to find whether there are occurrences of p in S such that the formulation above becomes $p_1 = s^{i_1}_j + c, p_2 = s^{i_2}_{j+1} + c, \ldots, p_m = s^{i_m}_{j+m-1} + c$ for some constant c. We present several algorithms solving the problem. Our main contribution, the MonoPoly algorithm, is a transposition-invariant bit-parallel filtering algorithm for static databases. It finds candidates for transposition invariant occurrences in time $O(\lceil |\Sigma'|/w \rceil n + m + \lceil m/w \rceil |\Sigma'| \cdot 2^{|\Sigma'|})$ where w and $|\Sigma'|$ denote sizes of the machine word in bits and the underlying (relative) alphabet, respectively. A straightforward algorithm working in time $O(nhm)$ is incrementally used to check whether the candidates are proper occurrences.

Key words: String algorithms, combinatorial pattern matching, bit parallelism, music retrieval

CR Classification: H.3.3, F.2.2, J.5

1. Introduction

String matching is a fundamental problem in many application areas, such as in information retrieval. The most conventional form of the problem is to find exact occurrences of a given query string $p = p_1 \cdots p_m$ within another string $s = s_1 \cdots s_n$, where each of p_i $(1 \leq i \leq m)$ and s_j $(1 \leq j \leq n)$ belongs to an alphabet Σ. The famous and practical solution for this problem was presented by Boyer and Moore [1977] with a worst-case time complexity of $O(nm)$, which was subsequently refined to $O(n + rm)$ where r denotes the number of occurrences [Guibas and Odlyzko 1980]. Later, Baeza-Yates and

Received January 7, 2003.

Gonnet [1992] introduced the SHIFTOR algorithm, an inspiring and efficient solution which uses the word-level bitwise operations of computer hardware. Their bit-parallel algorithm achieves a time complexity of $O(n\lceil\frac{m}{w}\rceil)$, where w is the size of the machine word (e.g. 32 or 64 bits, in practice).

In this paper, we consider some extensions of the exact string matching problem and present several algorithms solving them. Let us suppose that the underlying alphabet is a subset of natural numbers with standard arithmetic. In *multi-track string matching* (also called distributed string matching) the *text* S is composed of h parallel strings, $s^i = s^i_1 \cdots s^i_n, i \in \{1, \ldots, h\}$, called *tracks*, and the *pattern* p is said to have an *occurrence across the tracks* h at j, if $p_1 = s^{i_1}_j, p_2 = s^{i_2}_{j+1}, \ldots, p_m = s^{i_m}_{j+m-1}$ holds where $i_j \in \{1, \ldots, h\}$ for $j = 1, \ldots, m$. Note our distinction between s and S corresponding to a plain string and a multi-track string, respectively.

As it turns out, transposition invariance is a natural and useful property of our application. To this end, we update the formulation of the problem as follows: given the pattern p and the text S comprising h tracks, each track s^i of length $|s^i| = n$ and $i \in \{1, \ldots, h\}$, the task is to find all js such that $p_1 = s^{i_1}_j + c, p_2 = s^{i_2}_{j+1} + c, \ldots, p_m = s^{i_m}_{j+m-1} + c$ holds, for some constant c and for $i_1, \ldots, i_m \in \{1, \ldots, h\}$. We call this *transposition invariant multi-track string matching*.

In the next section we present some background for our study: First we briefly describe our application domain and show how music is represented by using strings. Then we give a brief summary of related work. Section 3 reviews the SHIFTOR algorithm and shows how it can be modified to be applicable for multi-track string matching. This modification, called SHIFTORAND, works in time $O(nh\lceil\frac{m}{w}\rceil)$. In Section 4, we will introduce the three novel algorithms for transposition invariant multi-track string matching. First we describe a straightforward $O(nhm)$ solution called DIRECTCHECK. It is based on a naïve string matching algorithm (see, e.g., [Crochemore and Rytter 1994, p. 34]). Then we show how the problem can be solved more efficiently in practice, by executing a filtering algorithm before DIRECTCHECK (or some other algorithm capable of checking). Having introduced the INTERVALMATCHING on-line filter working in time $O(nh^2\lceil\frac{m}{w}\rceil)$, we will devote more time in a detailed description and careful analyses on our main contribution, i.e., the MONOPOLY filtering algorithm. MONOPOLY is used with *static databases*, i.e. the database is not updated between consecutive queries. The algorithm consists of an $O(nhc)$ preprocessing and an $O(n\lceil\frac{m}{w}\rceil + m + d)$ filtering phase, where c and d denote factors dependent on the size of the alphabet. Before concluding the paper in Section 6, we will show the results of our extensive experiments on MONOPOLY in Section 5.

A preliminary version of this paper appeared in Lemström and Tarhio [2000].

2. Background

Multi-track string matching has an application area in content-based music retrieval (see, e.g., Lemström [2000]). Combinatorial string matching methods become applicable to music retrieval, when music is presented symbolically. For instance, the elements of a string may represent the *pitch* of a note (i.e. the perceived height of the played note). In terms of our specification above, music is said to be *monophonic* if $h = 1$, and it is called *polyphonic* if $h > 1$. In *homophonic music* there is a pitch for every s^i_j. Typically polyphonic music is not homophonic. We use an additional special character λ to denote a missing pitch.

The motivation for the problem under consideration is a typical music retrieval query case, where a monophonic pattern (that may be given, e.g. by humming, by playing an instrument, or just by typing) is searched for in a multi-track text representing a polyphonic music database[1]. Moreover, transposition invariance plays a central role in western music perception, for musical melodies are recognized rather based on the *intervals* between the consecutive pitches than on the absolute pitch sequences constituting the melodies.

2.1 Representing Music

In a rudimentary representation of polyphonic music, symbols of a string represent pitch (or interval) values of notes in one track, and the order of symbols within the string are in accordance with the note order of the represented track. A common underlying alphabet is based on MIDI pitch values [MIDI Manufacturers Association 1996]: $\Sigma_{128} = \{0, \ldots, 127\} \bigcup \{\lambda\}$ where 60 corresponds to the *middle-C*. For example, the excerpt given in Fig. 2.1 can be represented as follows: $s^1 = 65, 64, 62, 60$; $s^2 = 69, 67, 65, 64$; and $s^3 = 72, \lambda, \lambda, 72$.

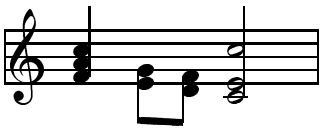

Fig. 2.1: A musical excerpt.

Note that when moving from absolute values to intervals, the size of the underlying alphabet is doubled. Henceforth we make a distinction between

[1] Indeed, a musical melody may occur distributed across several tracks (voices), as it is the case in Elgar's *Cockaigne*, for instance. However, in general a more pertinent case would be to try minimize the number of track shifts within an occurrence. Although this matter falls out of the scope of the current paper, the reader should note that the checking algorithm could be modified to consider the case, for it has the tracking information available.

interval and absolute alphabets: an interval alphabet, denoted by Σ', corresponds to an absolute alphabet Σ. Furthermore, we use a subscript to denote the size of the alphabet.

Due to a pragmatic problem — an alphabet as large as Σ'_{255} (MIDI interval alphabet) would make our principal algorithm impractical — we need a smaller but musically relevant alphabet. Another, musically relevant alphabet distinguishes only 12 pitches (or intervals). By musical terms, two pitches separated by 12 semitones is called *octave*. Among all the intervals octave is very special: it is the only interval whose arbitrary combinations are *consonant* Parncutt [1989]. *Octave equivalence*, "one of the most fundamental axioms of tonal music" [Forte 1962], means that intervals are reduced to (semitonic) scale $0, 1, \ldots, 11$. Technically this is achieved by using alphabet $\Sigma'_{12} = \{0, 1, \ldots, 11\}$ and replacing the original interval h by value h mod 12 (remember that our alphabets are subsets of natural numbers). Thus, an interval of 7 semitones upwards equals the interval of 5 semitones downwards, for instance. Using the alphabet Σ_{12}, the example in Fig. 2.1 would become as follows: $s^1 = 5, 4, 2, 0$; $s^2 = 9, 7, 5, 4$; and $s^3 = 0, \lambda, \lambda, 0$.

Let Σ_ℓ be the alphabet. By S_j we denote an ordered vertical section of the text at j, i.e., $S_j = s_j^1, s_j^2, \ldots, s_j^h$ where $s_j^i \leq s_j^{i+1}$ for $1 \leq i \leq h-1$, i.e. the pitches of S_j are in the nondecreasing order. We call such a vertical section a *chord*. The chords can be represented by bitvectors $\overline{\mathbf{S}}[j]$, where each $\overline{\mathbf{S}}[j]$ is a chord bitvector (cbv) of ℓ bits. To be precise, each cbv is formally a symbol of a cbv alphabet $\overline{\Sigma}$ of size $|\overline{\Sigma}| = 2^\ell$. Nevertheless, as the connection between Σ and $\overline{\Sigma}$ is straightforward, we will mostly avoid the explicit exposition of cbv alphabets to improve the readability. A zero in a cbv corresponds to a present pitch in the chord, while an on-bit indicates absence of the corresponding pitch. For instance, if the underlying alphabet is Σ_{12} (absolute pitches reduced according to octave equivalence), the cbv string $\overline{\mathbf{S}} = \langle\overline{\mathbf{S}}[1]\rangle \cdots \langle\overline{\mathbf{S}}[n]\rangle$ corresponding to Fig. 2.1 would be:

$$\langle 0,1,1,1,1,0,1,1,1,0,1,1\rangle,$$
$$\langle 1,1,1,1,0,1,1,0,1,1,1,1\rangle,$$
$$\langle 1,1,0,1,1,0,1,1,1,1,1,1\rangle,$$
$$\langle 0,1,1,1,0,1,1,1,1,1,1,1\rangle.$$

By $\overline{\mathbf{S}}[j].i$ we denote the ith bit of the cbv $\overline{\mathbf{S}}[j]$; e.g. above $\overline{\mathbf{S}}[3].2 = 0$.

In Section 5, we will show that the musically relevant alphabet Σ'_{12} is practical and effective for our application.

2.2 Related Work

Independently of us, Holub *et al.* [2001] presented bit-parallel algorithms for multi-track string matching. They did not, however, consider transposition invariance. They presented algorithms to find occurrences of (i) multi-track patterns within plain texts (effectively, the original SHIFTOR algorithm); (ii) plain patterns within multi-track texts (effectively same as our

SHIFTORAND); and (iii) multi-track patterns within multi-track texts. The algorithm (iii) requires $O((rm + |\hat{\Sigma}|)\lceil \frac{m}{w} \rceil)$ time for the preprocessing, where r, $\hat{\Sigma}$, and w are the number of the patterns, the set of symbols used in the pattern and the size of the machine word, respectively. Then it works in $O(nh\lceil \frac{m}{w} \rceil)$ time and requires $O(|\hat{\Sigma}|\lceil \frac{m}{w} \rceil)$ space.

Dovey [2001] has considered a modification of the multi-track string matching problem, where the consecutive matching elements of an occurrence may contain *gaps*. Given a gapping parameter t, a *t-gap-occurrence* is as follows: $p_1 = s_{j_1}^{i_1}, p_2 = s_{j_2}^{i_2}, \ldots, p_m = s_{j_m}^{i_m}$, where $j_{l+1} - j_l \leq t + 1$ for $1 \leq l \leq m - 1$ and $i_1, \ldots, i_m \in \{1, \ldots, h\}$. By setting $t = n$, the gaps become unrestricted. Dovey represents music by chord vectors over alphabet Σ_{88} (88 is the common amount of keys in a piano). His algorithm works in time $O(nm\lceil \frac{88}{w} \rceil)$ based on the following dynamic programming recurrence:

$$
\begin{aligned}
d_{00}, d_{i0}, d_{0j} &= 0; \\
d_{ij} &= \begin{cases} t+1, & \text{if } ((p_i \in S_j) \text{ and } (i = 1 \text{ or } d_{i-1,j-1} \neq 0)); \\ d_{i,j-1} - 1, & \text{if } ((p_i \notin S_j) \text{ and } (d_{i,j-1} \neq 0)); \\ 0, & \text{otherwise,} \end{cases}
\end{aligned}
$$

where $1 \leq j \leq n$ and $1 \leq i \leq m$. As usually, the query result is read from the element d_{mj}; an occurrence of value $t+1$ in a bottom row element indicates an t-gap-occurrence, and the actual occurrence can be uncovered by a backtracking procedure. The algorithm requires 87 reiterations for transposition invariant matching.

The recent SIA(M)EX algorithm by Wiggins *et al.* [2002] is capable of transposition invariant multi-track string matching with unrestricted gaps. It is not, however, a string matching algorithm. It works with any n-dimensional datasets; in our case the pattern and the text are formed as pairs (q, t), where q and t represent the pitch and its onset time, respectively. Let $\overline{v} = (a, b)$ be a transfer vector that transfers a pair (q, t) to $\overline{v}[(q, t)] = (q + a, t + b)$. The intuition behind their algorithm is to collect in a matching set $\mathcal{M}$ vectors $\overline{v}$ which transfer pattern points to text points:

$$\mathcal{M} = \{\langle \overline{v}, X \rangle \mid x \in X \iff ((x \in p) \wedge (\overline{v}[x] \in S))$$

The pattern p has a transposition invariant multi-track occurrence in the text S if and only if $|X| = m$ ($X = p$) holds for some pair $\langle \overline{v}, X \rangle$ of $\mathcal{M}$.

Denoting by n' and m' the number of events in the text and in the pattern, respectively, SIA(M)EX runs in time $O(n'm')$ and space $O(m')$ (note that in homophonic music $m' = hm$ and $n' = hn$).

3. SHIFTOR Algorithm

Let us consider the SHIFTOR algorithm by Baeza-Yates and Gonnet [1992]. In describing their algorithm (and henceforth) we will use the symbols $\vee$ and

```
SHIFTOR(s, p, n, m, Σ)
1   for each a ∈ Σ do T[a] ← 2^m − 1
2   for i ← 1 to m do T[p_i] ← T[p_i] − 2^(i−1)
3   E ← 2^m − 1
4   for j ← 1 to n do
5     E ← shiftleft(E) ∨ T[s_j]
6     if E.m = 0 then WRITE(j)
```

Fig. 3.1: The SHIFTOR algorithm.

$\wedge$ representing the bitwise `or` and `and` operators, respectively. The SHIFTOR algorithm searching occurrences of p in s is given in Fig. 3.1.

Lines 1–2 and 3–6 of SHIFTOR form two phases, which we call pattern processing and core phases, respectively. First, for each symbol appearing in the pattern, the pattern processing phase creates a bit-mask appearing as a column of table `T`. At the core phase, a zero bit is released (by the binary `shiftleft` operator) to level 1 at every point of time. Then, the released zero bits either survive to the next level, or die, depending on the bit-mask used with the $\vee$ operator. Whenever a zero bit reaches the level m, an occurrence of the pattern has been found; this is reported on line 6. Fig. 3.2 simulates SHIFTOR in an example case.

SHIFTOR's pattern processing takes $O(\lceil \frac{m}{w} \rceil |\Sigma| + m)$, while the core runs in time $O(\lceil \frac{m}{w} \rceil n)$. The overall space requirement is $O(\lceil \frac{m}{w} \rceil (|\Sigma| + 1))$.

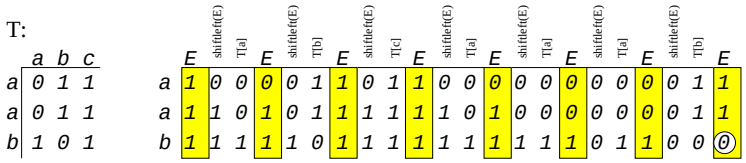

Fig. 3.2: An example of SHIFTOR for $p = aab$ and $s = abcaaab$. The table T (on the left) is created first. The execution of the core phase of the algorithm is illustrated on the right. In the illustration, shifting is done downwards and the found occurrence is shown with a white circle.

3.1 SHIFTORAND — *Algorithm for Multi-Track String Matching*

The SHIFTOR algorithm can be adapted with a minor modification to multi-track string matching. Actually, this is a dual of the string matching problem presented by Baeza-Yates and Gonnet [1992]. They considered cases where elements of a pattern may contain a set of symbols instead of one symbol. In their case they modified the pattern processing phase, while in our case each text position is allowed to contain a set of characters and the modified

```
ShiftOrAnd(S, p, n, m, Σ)
1   for each a ∈ Σ do T[a] ← 2^m − 1
2   for i ← 1 to m do T[p_i] ← T[p_i] − 2^(i−1)
3   E ← 2^m − 1
4   for j ← 1 to n do
5     E ← shiftleft(E) ∨ (⋀({T[a] | a ∈ S_j}))
6     if E.m = 0 then Write(j)
```

Fig. 3.3: The ShiftOrAnd algorithm for multi-track string matching.

phase is the core phase. This is done by adding a bitwise `and` operation, which operates over all the pitches within a chord; see the ShiftOrAnd algorithm in Fig. 3.3, below.

The main loop in lines 4–6 takes $O(n\lceil\frac{m}{w}\rceil h)$ time, and an extra space of $\lceil\frac{m}{w}\rceil$ words is required. The extra space is used for a temporary storage, where the bitwise `and` operation can bring all the required zero bits. Actually, the algorithm can be modified so that the core runs in time $O(n\lceil\frac{m}{w}\rceil)$. In that case, table `T` contains a column for each chord S_j (instead of each character in Σ). Naturally, both the time complexity of pattern processing and the overall space complexity increase noticeably from that of the version given in Fig. 3.3.

4. Transposition Invariant Multi-Track String Matching

The problem of multi-track string matching becomes trickier when taking into account transposition invariance. In this section we present two fast filtering methods to solve the problem, one works on-line the other off-line. In the on-line method all computation is done during a query execution, while the off-line method is tailored to deal with static databases. In the latter case, as much as possible is done in a separate preprocessing phase to enable faster responses to queries.

We start by introducing a straightforward algorithm that serves for two different needs. On one hand, it may be used as a *total algorithm*, i.e., it works on its own to search for occurrences. On the other hand, with a slight modification, it may be used as a *checking algorithm*. In the latter case it only checks whether there is an occurrence at a given position. The algorithm is based on the naïve string matching algorithm, see e.g. [Crochemore and Rytter 1994, p. 34].

4.1 DirectCheck — *Straightforward Algorithm*

For the sake of simplicity, let us consider the case where DirectCheck is used as a total method (see Fig. 4.1), where Exit is a routine that halts the

execution of the innermost loop. At first, the algorithm computes $\overline{\mathbf{S}}$, the cbv representation of the input S. Then the algorithm checks for each position j, $1 \leq j \leq n - m + 1$, and for each pitch $a \in S_j$, whether there is a match starting from a at position j.

```
DIRECTCHECK(S, p, n, m, Σ)
1   COMPUTE S̄(S)
2   for j ← 1 to n − m + 1 do
3     for each a ∈ S_j do
4     found ← true; b ← a
5       for i ← 2 to m do
6         x ← b + p_i − p_{i−1}
7         if ((0 ≤ x < |Σ|) and (S̄[j + i − 1].x = 0))
8           then b ← x
9           else found ← false; EXIT()
10      if found
11        then PRINT(occurrence at S_j ··· S_{j+m−1}); EXIT()
```

Fig. 4.1: DIRECTCHECK for transposition invariant multi-track string matching.

Because each chord S_j holds at most h pitches, the time complexity of forming $\overline{\mathbf{S}}$ is $T_1 = O(nh) + O(n\lceil|\Sigma|/w\rceil)$, where the latter time is needed for initializing chord bitvectors with ones. If a circular buffer of m chords is used and the computation is merged with the matching phase, the initialization takes only $O(m\lceil|\Sigma|/w\rceil)$. A column of the buffer can then be updated in $O(h)$ time by replacing h zeros by ones according to the previous chord and then replacing h ones by zeros according to the new chord.

The time complexity of the rest of the algorithm is $T_2 = O(nhm)$, because there are $(n - m + 1) \cdot h \cdot (m - 1)$ comparisons in the worst case. Because $nh \gg \lceil|\Sigma|/w\rceil$ holds in practice, T_2 dominates over T_1. Therefore we consider $O(nhm)$ as the total time of DIRECTCHECK in the following[2]. The space requirement of the algorithm is $O(m\lceil|\Sigma|/w\rceil)$ with the circular buffer and $O(n\lceil|\Sigma|/w\rceil)$ without it.

Note that in Fig. 4.1, a substring $S_j \cdots S_{j+m-1}$ is reported as an occurrence only once in a case where it actually contains several occurrences. If all the parallel occurrences have to be reported, the time complexity does not change, but the algorithm will be slightly slower in pathological cases.

In the checking version of DIRECTCHECK, j is given as a parameter to the algorithm and the outermost loop (line 2) is absent.

[2] The expected running time, however, is $O(nh)$. This is a characteristic property of the naïve matching algorithm.

```
IntervalMatching(S, p, n, m, Σ')
1   for each d ∈ Σ' do T[d] ← 2^(m-1) − 1
2   for i ← 2 to m do T[p_i − p_(i-1)] ← T[p_i − p_(i-1)] − 2^(i-2)
3   E ← 2^(m-1) − 1
4   for j ← 2 to n do
5     D ← 2^(m-1) − 1
6     for each a ∈ S_(j-1) and b ∈ S_j do D ← D ∧ T[b − a]
7     E ← shiftleft(E) ∨ D
8     if E.m = 0 then Check(j)
```

Fig. 4.2: The IntervalMatching on-line filter for transposition invariant multi-track string matching.

4.2 IntervalMatching — *On-Line Filtering Algorithm*

Let us now introduce a basic on-line filter for transposition invariant multi-track string matching. The IntervalMatching algorithm (Fig. 4.2) uses an interval alphabet. By comparing it with ShiftOrAnd (Fig. 3.3), one can notice two differences. Firstly, in IntervalMatching we introduce a bitvector D which collects all the intervals between two consecutive chords (line 6). These intervals are then used in the shifting similarly as in ShiftOrAnd. The other difference can be noticed on line 8: IntervalMatching is a filtering method, because it only makes sure that a candidate contains the intervals of the pattern in the correct order but does not necessarily 'bind' the corresponding elements of the chords (see Fig. 4.3 for an illustration). Hence, IntervalMatching has to call the checking algorithm for every found candidate.

Clearly, the core of the algorithm runs in time $O(n\lceil\frac{m}{w}\rceil h^2)$. However, the worst case time complexity is that of the checking algorithm, because there might be a candidate at each position, in the worst case. Furthermore, the required extra space is $h^2 + \lceil\frac{m}{w}\rceil$; h^2 is needed for the set D, and $\lceil\frac{m}{w}\rceil$ words for the bitwise `and` operation (as in ShiftOrAnd). Thus, the total space requirement of IntervalMatching is $O(\lceil\frac{m}{w}\rceil(|\Sigma| + 2) + h^2)$.

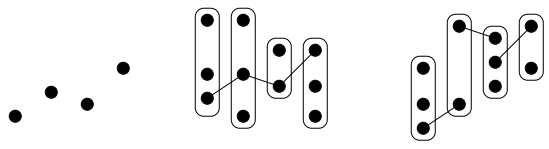

Fig. 4.3: The query pattern, given on the left, has a proper occurrence in the first chord string (the corresponding elements are bound), but only a spurious occurrence in the second (the corresponding elements are not bound). Both are considered as candidates.

4.3 MONOPOLY — *Off-Line Filtering Algorithm*

When the text (corresponding to a music database) is static, it can be preprocessed in order to speed up the retrieving. The benefit of the preprocessing is considerable when the text is subject to several consecutive queries. Besides, if the result of preprocessing can also be stored, it is possible to preprocess the pieces of music, when they are added to the database.

The operation of MONOPOLY is divided into *preprocessing* and *filtering* phases. The preprocessing is necessary only before the first query. The key idea of the algorithm is to store intervals of two consecutive chords as a bit-vector in the preprocessing phase. An array $\overline{\mathtt{S}}'$ of these interval combinations represented as bit-vectors is used as a text for the SHIFTOR algorithm while searching for the interval sequence of the original pattern. An array T' corresponds to the array T of the original SHIFTOR. The bit $\mathtt{T}'[l].i$ is zero, when the kth bit of l is zero (i.e., the interval k belongs to the interval combination l) such that k is the interval in the pattern between p_i and p_{i-1}). MONOPOLY has been designed for moderate interval alphabets, smaller than Σ'_{20}.

4.3.1 Preprocessing Phase

This phase given in Fig. 4.4 forms a string $\overline{\mathtt{S}}'[1]\cdots\overline{\mathtt{S}}'[n-1]$, where each $\overline{\mathtt{S}}'[j]$ is a bit-vector of $|\Sigma'|$ bits storing the intervals between chords S_j and S_{j+1}. Formally,

$$\overline{\mathtt{S}}'[j].i = \begin{cases} 0, \text{ if } i = (x-y) \bmod |\Sigma'| \\ \quad \text{for some } x \in S_j \text{ and } y \in S_{j+1} \ \ (1 \le j \le n-1); \\ 1, \text{ otherwise.} \end{cases}$$

We avoid the apparent $O(h^2\lceil\frac{|\Sigma'|}{w}\rceil)$ time requirement for processing a pair of chords by using bitwise operations and a fact that only a certain subset of the possible h^2 intervals can appear between two consecutive chords: When the intervals from an element x of some chord S_j to the elements of the following chord S_{j+1} have been calculated, the intervals for another element y in the chord S_j can be uncovered just by shifting those calculated intervals by the difference between y and x.

In the algorithm $\mathtt{B}(S_j) = s_j^1$ denotes the lowest pitch value, the *bass*, of a chord S_j. Zeros in $\overline{\mathtt{S}}'[j]$ before line 6 give the intervals between the elements of S_{j+1} and the bass of the chord S_j (see the bitvector in the topleft corner of the example given in Fig. 4.6). Then the zeros are shifted according to the remaining elements of S_j, one-by-one, to give the rest of the intervals occurring between chords S_j and S_{j+1} (see the next two bitvectors in the topleft corner in Fig. 4.6). Finally, $\overline{\mathtt{S}}'[j]$ collects all the zeros that appeared in any of the (shifted) bitvectors (the fourth bitvector in the topleft corner in Fig. 4.6). This can be implemented efficiently by using a right circularshift bitwise operator, denoted $\mathtt{rcs}(\mathtt{a}, b)$, which shifts a bit-vector a by b bits to the

right in a circular manner. For instance, if $\mathtt{a} = 01010$ then $\mathtt{rcs}(\mathtt{a}, 1) = 00101$ and $\mathtt{rcs}(\mathtt{a}, 2) = 10010$. Since at most $(n-1)\cdot(h-1)$ such copying are needed, $\overline{\mathtt{S}}'[1]\cdots\overline{\mathtt{S}}'[n-1]$ can be formed in $O(nh\lceil\frac{|\Sigma'|}{w}\rceil)$ time.

MONOPOLY:PP(S, p, n, m, Σ')
1 **for** $j \leftarrow 1$ **to** $n-1$ **do**
2 $\quad\overline{\mathtt{S}}'[j] \leftarrow 2^{|\Sigma'|} - 1$
3 $\quad$**for each** $a \in S_{j+1}$ **do**
4 $\quad\quad b \leftarrow (a - \mathtt{B}(S_j)) \bmod |\Sigma'|$
5 $\quad\quad$**if** $\overline{\mathtt{S}}'[j].b = 1$ **then** $\overline{\mathtt{S}}'[j] \leftarrow \overline{\mathtt{S}}'[j] - 2^b$
6 $\quad\overline{\mathtt{S}}'[j] \leftarrow (\overline{\mathtt{S}}'[j] \ \wedge (\bigwedge_{a\in(S_j\setminus\mathtt{B}(S_j))} \mathtt{rcs}(\overline{\mathtt{S}}'[j], (a - \mathtt{B}(S_j)) \bmod |\Sigma'|)))$

Fig. 4.4: The preprocessing phase of MONOPOLY.

4.3.2 Filtering Phase

The filtering phase is divided into *pattern processing*, and *core* subphases. The two subphases correspond to the phases of the SHIFTOR algorithm.

The pattern processing subphase constructs a bit-array T' of $(m-1)\times 2^{|\Sigma'|}$ bits corresponding to the bit-array T of SHIFTOR. Instead of having a column for every symbol appearing in the text, T' has a column for every possible value of $\overline{\mathtt{S}}'[j]$.

For efficient computation, two extra arrays are used while composing T'. A bit-array I of $|\Sigma'|\times|\Sigma'|$ bits has a column for every possible interval in Σ', while a bit-array L of $|\Sigma'|\times(m-1)$ bits stores the positions of each interval in the query pattern. Their bits are set as follows (here $1 \le i \le m-1$; $1 \le j, k \le |\Sigma'|$)

$$\mathtt{I}[j].k = \begin{cases} 0, \text{ if } j = k, \\ 1, \text{ otherwise,} \end{cases} \quad \mathtt{L}[j].i = \begin{cases} 0, \text{ if } (p_{i+1} - p_i) \bmod |\Sigma'| = j, \\ 1, \text{ otherwise.} \end{cases}$$

Remember that formally $\overline{\mathtt{S}}'[j] \in \overline{\Sigma}'$ and $|\overline{\Sigma}'| = 2^{|\Sigma'|}$. Thus, $\overline{\mathtt{S}}'[j]$ can be interpreted as an integer l, $l \in [0, 2^{\Sigma'} - 1]$. These values are used as indices to the table T'. Moreover, we use bit-vectors $\mathtt{I}[j]$ to locate intervals within $\overline{\mathtt{S}}'$, by 'sliding' them one-by-one over all the values l. This forms the table T':

$$\mathtt{T}'[l].i = \begin{cases} 0, \text{ if } \mathtt{I}[k].j = 0 \text{ and } l.j = 0 \text{ and } \mathtt{L}[k].i = 0, \\ 1, \text{ otherwise,} \end{cases}$$

where $l.j$ denotes the jth bit of l. In this way, constructing the array T' takes time $O(\lceil\frac{m}{w}\rceil|\Sigma'|\cdot 2^{|\Sigma'|})$.

The core phase is analogous to that of SHIFTOR algorithm, but in this case the pattern will be matched against the string $\overline{\mathtt{S}}'[1]\cdots\overline{\mathtt{S}}'[n-1]$ instead

of s. The algorithm in Fig. 4.5 implements the whole filtering phase, and Fig. 4.6 illustrates the functioning of MONOPOLY on an example case.

MONOPOLY($\overline{\mathtt{S}}', p, n, m, \Sigma'$)
1 **for** $k \leftarrow 1$ **to** $|\Sigma'|$ **do**
2 $\quad \mathtt{I}[k] \leftarrow 2^{|\Sigma'|} - 1$
3 $\quad \mathtt{I}[k] \leftarrow \mathtt{I}[k] - 2^{k-1}$
4 $\quad \mathtt{L}[k] \leftarrow 2^{m-1} - 1$
5 **for** $i \leftarrow 2$ **to** m **do**
6 $\quad b \leftarrow (p_i - p_{i-1}) \bmod |\Sigma'|$
7 $\quad \mathtt{L}[b] \leftarrow \mathtt{L}[b] - 2^{i-2}$
8 **for** $l \leftarrow 0$ **to** $2^{|\Sigma'|} - 1$ **do**
9 $\quad \mathtt{T}'[l] \leftarrow 2^m - 1$
10 $\quad$ **for** $k \leftarrow 1$ **to** $|\Sigma'|$ **do**
11 $\quad\quad$ `ivect` $\leftarrow \mathtt{I}[k]$
12 $\quad\quad$ **if** `ivect` $\vee\, l =$ `ivect` **then** $\mathtt{T}'[l] \leftarrow (\mathtt{T}'[l] \wedge \mathtt{L}[k])$
13 $\mathtt{E} \leftarrow 2^{m-1} - 1$
14 **for** $j \leftarrow 1$ **to** $n - 1$ **do**
15 $\quad \mathtt{E} \leftarrow$ `shiftleft`$(\mathtt{E}) \vee \mathtt{T}'[\overline{\mathtt{S}}'[j]]$
16 $\quad$ **if** $\mathtt{E}.m = 0$ **then** CHECK(j)

Fig. 4.5: The MONOPOLY filter for transposition invariant multi-track string matching.

4.4 Correctness and Analysis of MONOPOLY

We prove first that $\overline{\mathtt{S}}'$ is correctly formed. In the following, the interval between the bass of the chord S_j and a pitch y within the chord S_{j+1} is denoted by $r(y)$, where $r(y) = y - \mathtt{B}(S_j)$. The distance from the bass pitch within the same chord is denoted by $r_0(y)$, i.e. $r_0(y) = y - \mathtt{B}(S_j)$.

LEMMA 1. *Let $x \in S_j$ and $y \in S_{j+1}$ for some j, $1 \leq j \leq n-1$. If $i = (y - x) \bmod |\Sigma'|$ holds, then $\overline{\mathtt{S}}'[j].i$ is a zero bit.*

PROOF. The array $\overline{\mathtt{S}}'$ is computed during the preprocessing phase. There are two cases to be considered: (i) $x = \mathtt{B}(S_j)$ and (ii) $x \neq \mathtt{B}(S_j)$.

(i) The index i for the zero bit is $r(y) \bmod |\Sigma'|$ (line 4). Then the zero bit is assigned to $\overline{\mathtt{S}}'[j].i$ (line 5). Since after that, the only remaining operation that updates $\overline{\mathtt{S}}'[j]$ (line 6) preserves that zero bit ($\wedge$ preserves zeros), $\overline{\mathtt{S}}'[j].i = 0$ holds.

(ii) According to case (i), $\overline{\mathtt{S}}'[j].i = 0$ holds for all the intervals i between $\mathtt{B}(S_j)$ and S_{j+1} before line 6 is executed. Let y be an arbitrary pitch within S_{j+1}, $d = y - \mathtt{B}(S_j)$, and $e = y - x$. Now the difference of e

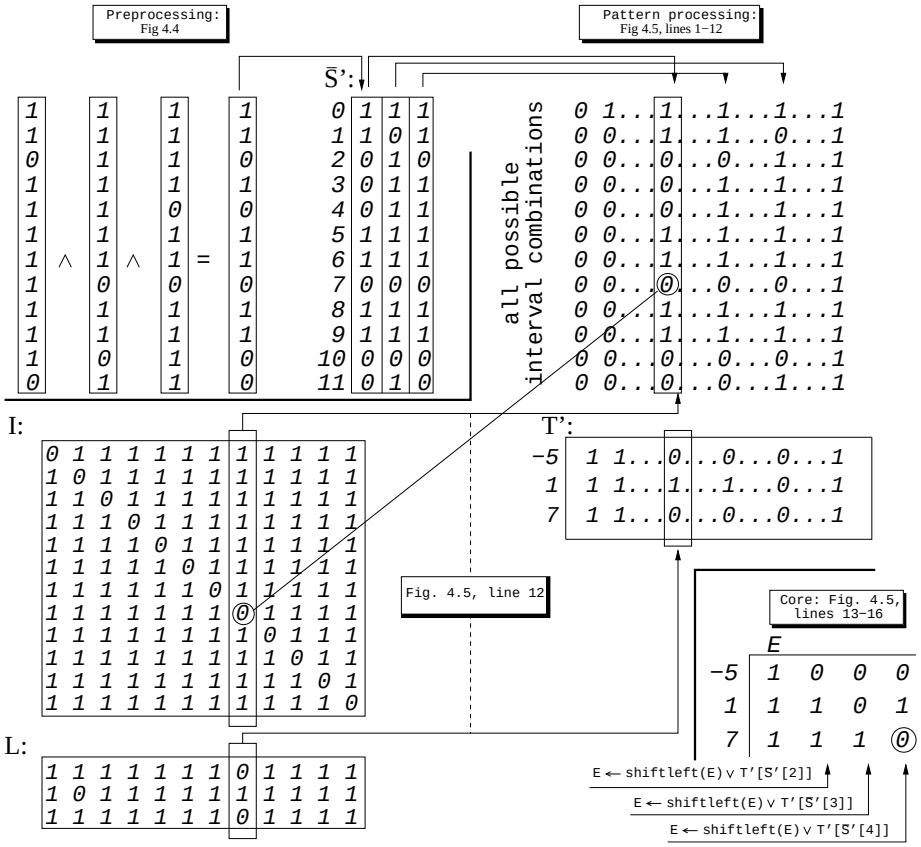

Fig. 4.6: MONOPOLY on an example case: $|\Sigma'| = 12$, $p = 69, 64, 65, 72$ ($p' = -5, 1, 7$) and S is as Fig. 2.1 ($S_1 = \{65, 69, 72\}, S_2 = \{64, 67\}, S_3 = \{62, 65\}$, and $S_4 = \{60, 64, 72\}$).

and d is $\mathtt{B}(S_j) - x$, which by definition equals to $-r_0(x)$. Therefore, since d has already been stored in $\overline{\mathtt{S}}'[j]$, $e = d - r_0(x)$ can be stored by assigning a zero bit at the location $i = e \bmod |\Sigma'|$. This is done on line 6 by the `and` operation with $\mathtt{rcs}(\overline{\mathtt{S}}'[j], r_0(x))$. Again $\overline{\mathtt{S}}'[j].i = 0$ holds.

LEMMA 2. *If $i \neq (y - x) \bmod |\Sigma'|$ holds for every pair $x \in S_j$ and $y \in S_{j+1}$, then $\overline{\mathtt{S}}'[j].i$ is one.*

PROOF. After the execution of line 2, $\overline{\mathtt{S}}'[j].i = 1$ holds. The execution of the for loop in lines 3–4 assigns a zero to $\overline{\mathtt{S}}'[j]$, corresponding to an interval $y - \mathtt{B}(S_j)$ for each $y \in S_{j+1}$. Let A be the value of $\overline{\mathtt{S}}'[j]$ after the loop. In line 5, first $|S_j| - 1$ shifted copies out of A are formed, then they are combined with A by using the $\wedge$ operation (line 6). Each copy holds all the intervals between S_{j+1} and some x that differs from $\mathtt{B}(S_j)$. Clearly the algorithm does not assign superfluous zeros to $\overline{\mathtt{S}}'[j]$. □

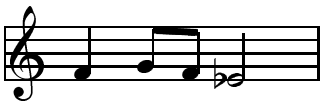

Fig. 4.7: This excerpt has a spurious occurrence in Fig. 2.1.

As a consequence of the lemmas, the following theorem holds.

THEOREM 1. *$\overline{\mathtt{S}}'$ is correctly formed.*

Let us continue by proving that the pattern processing phase works correctly, and thus that table $\mathtt{T}'$ is correctly formed. In the following, if l is an integer, then l_b denotes that the integer is interpreted as a bitvector.

LEMMA 3. *Let i be an integer, $2 \leq i \leq m$. $\mathtt{T}'[l].i = 0$ holds, if and only if $l_b.k$ is zero and $k = (p_i - p_{i-1}) \bmod |\Sigma'|$ holds for some k.*

PROOF. It is sufficient to consider only the processing of the pattern (lines 1–12, Fig. 4.5). Let us assume that $l_b.k$ is zero and $k = (p_i - p_{i-1}) \bmod |\Sigma'|$ holds. In lines 5–7 the intervals of the pattern are stored in the table L at the location corresponding to $(p_i - p_{i-1}) \bmod |\Sigma'|$, for $2 \leq i \leq m$. Clearly there is exactly one zero bit on each row of L, and the zero bits are assigned to the correct positions according to the construction. The only zero bit in $\mathtt{I}[k]$ is the kth bit. Thus, $\mathtt{I}[k] \vee l_b = \mathtt{I}[k]$ holds, and the condition in line 12 is met. Then the zero-preserving operation $\wedge$ is used to assign a zero to $\mathtt{T}'[l].i$.

Let us then assume that $\mathtt{T}'[l].i = 0$ holds. By inspecting line 12 we conclude that there is a k such that $\mathtt{L}[k].i = 0$ holds. According to the construction, $l.k$ must be zero and $k = (p_i - p_{i-1}) \bmod |\Sigma'|$ must hold. □

Considering MONOPOLY without the checking phase, the original problem of finding every transposed occurrence of a music pattern has been transformed to a filtration problem of finding candidate occurrences H of p. Such an H is an interval string of length $m-1$ in $\overline{\mathtt{S}}'$, which contains the intervals of p in the correct order (recall Fig. 4.3). However, the condition that there is a c such that $(p_i + c) \in S_{j+i-1}$ for each i does not necessarily hold any longer. An example of a candidate that is not a spurious occurrence is when the excerpt in Fig. 4.7 represents the pattern and Fig. 2.1 the text. The following theorem shows that filtration works correctly, i.e. MONOPOLY does not skip any proper occurrence.

THEOREM 2. *Let p be the pattern to be searched within the text S. If there is a transposition invariant occurrence starting at S_j, then* MONOPOLY *finds a potential occurrence of p starting at $\overline{\mathtt{S}}'[j]$.*

PROOF. The table $\overline{\mathtt{S}}'$ is correctly constructed according to Theorem 1. The core phase works analogously to that of SHIFTOR. An interval in our setting corresponds to a character. As a conjunction of the vectors $\mathtt{L}[k]$, $\mathtt{T}'[j]$ has got the corresponding intervals belonging to $\overline{\mathtt{S}}'[j]$. The table $\mathtt{T}'$ is correctly constructed according to Lemma 3. The fact that each potential occurrence is identified follows from the characteristics of the SHIFTOR algorithm. □

MONOPOLY's space complexity is $O(n\lceil\frac{|\Sigma'|}{w}\rceil + c)$, where $c = |\Sigma'|\lceil\frac{|\Sigma'|}{w}\rceil + (2^{|\Sigma'|} + |\Sigma'|)\lceil\frac{m}{w}\rceil$. For the preprocessing, $O(nh\lceil\frac{|\Sigma'|}{w}\rceil)$ time is needed. At the beginning of the filtering phase the locations of intervals are gathered in time $O(m)$. After that, each interval mask $\mathtt{I}$ is slid over the values l_b, which takes $O(\lceil\frac{m}{w}\rceil|\Sigma'| \cdot 2^{|\Sigma'|})$. Therefore, by denoting $d = \lceil\frac{m}{w}\rceil|\Sigma'| \cdot 2^{|\Sigma'|}$, the filtering takes time $O(n\lceil\frac{m}{w}\rceil + m + d)$, which is linear in n when $m \leq w$. Again, the worst case time complexity is that of the checking algorithm; there might be a candidate at each position, in the worst case.

MONOPOLY becomes impractical if unlimited interval alphabet, or even Σ'_{255}, is used. The octave equivalence assumption, for instance, keeps the table T' reasonably sized, and thus, MONOPOLY practical.

4.5 Improving MONOPOLY*'s Performance*

Navarro and Raffinot [1998] introduced a crossing of SHIFTOR and the Boyer and Moore [1977] algorithm. Their bit-parallel BNDM (Backward Nondeterministic Dawg Matching) algorithm emulates the BDM algorithm [Czumaj *et al.* 1994] based on a nondeterministic suffix automaton.

BNDM follows the Boyer-Moore principle: the pattern matching starts at the position m of p and s. Then the pattern and text characters are compared in the right-to-left order until the whole pattern is recognized or a mismatch occurs. In each step, bit parallelism is used in a clever way to emulate a nondeterministic suffix automaton, in order to know whether the current suffix of s is a prefix of p. If such a prefix is found, the value of the next shift is updated.

In order to make MONOPOLY filter faster, the core phase (lines 13–16 in Fig. 4.5) could be replaced by BNDM. Although the BNDM algorithm has a worst-case complexity of $O(nm)$, it is faster than SHIFTOR, in practice. As with all Boyer-Moore type algorithms, BNDM becomes faster as pattern gets longer. According Navarro and Raffinot's experiment, BNDM is up to 7 times faster than SHIFTOR, when $m = 32$. Nevertheless, since our problem is different and the patterns are typically rather short, we expect a smaller speed-up in our case.

5. Experiments

In experimenting the practical performance of MONOPOLY, we compared its efficiency against that of DIRECTCHECK. We used the modification, discussed in Subsection 4.1, of DIRECTCHECK as the subroutine for checking.

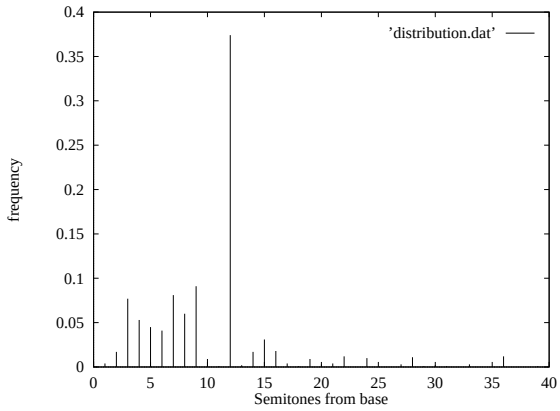

Fig. 5.1: A distribution of intervals within chords in Sibelius' Finlandia.

We did not test INTERVALMATCHING. However, it may be expected that its performance lies somewhere between the two tested approaches.

We made an extensive study on altering the values of the interesting parameters, and observed their effects to the performance. In every piece of experiment, we measured the running times, and both the numbers of the candidates and proper occurrences. The parameters under consideration were m, n, h, and $|\Sigma'|$. The impact of the four parameters was measured by letting only one parameter vary at a time, meanwhile the values of the other parameters were fixed. The experiments were run in a PC with Intel Pentium III of 700 MHz and 768 MB of RAM under the Linux operating system. The length w of a machine word was 32 bits.

The database for the experiments was collected from the Internet. It comprised 7,667 MIDI files, out of which 6,190 were originally monophonic. In the database, the maximum degree of polyphony was 8, but typically there were several monophonic chords between any two polyphonic chords. Although we believe that this is a rather common phenomena (which makes MONOPOLY more efficient due to the fewer false positive hits found), we wanted to bound the degree of polyphony (as it is described in the problem specification). In other words, we forced each piece of music in our database to be homophonic. In order to do that, we first computed the distribution of intervals within chords (as semitones from the bass) in a MIDI file of Jean Sibelius' Finlandia (see Fig. 5.1 for the distribution). Then, for each chord S_j, we inserted random pitches following the measured interval distribution until each S_j became equal to h.

The series of experiments was started by building up the text residing in the main memory. All the homophonic pieces of music in our MIDI database were concatenated into a single string, resulting in $n = 1,484,940$. Observing one parameter at a time, each setting (e.g. $h = 8$, other fixed to default values) was repeated 100 times. At the beginning of each repetition, a new pattern was randomly picked up from the text. Thus, it was guaranteed that

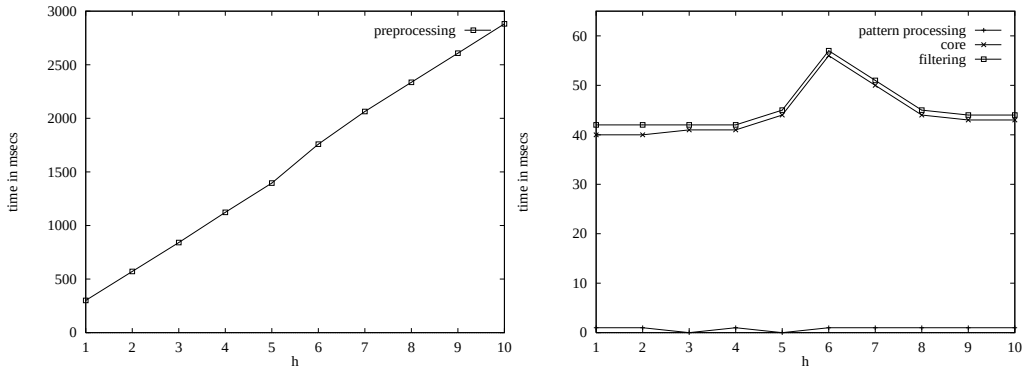

Fig. 5.2: The performance of MONOPOLY when varying h ($m = 12$; $n = 1,484,940$; $|\Sigma'| = 12$). The preprocessing time is given in the graph on the left, pattern processing, core, and total filtering (=pattern processing+core) times in the graph on the right.

at least one occurrence was to be found in each repetition. As results of the experiments, we report the averages of repetitions for each setting.

The default values for the experimented parameters were: $h = 3$; $m = 12$; $n = 1,484,940$; and $|\Sigma'| = 12$.

Fig. 5.2 illustrates the typical behaviour of MONOPOLY. In the two graphs, we have given the average times spent by the different phases of the algorithm, varying the value of h. Firstly, the preprocessing time grows noticeably as h increases (see the graph on the left). In the graph on the right, we give the times spent by the pattern processing (the lowest curve), the core, and by the whole filtering phase. Note the interesting peak in the latter two: As the value of h is increased the number of distinct $\overline{\mathtt{S}}'[i]s$ becomes larger. This cause the execution to get slower because of fewer corresponding $\mathtt{T}'$ values present in the cache. Moreover, when the increasing of h is continued, after some threshold point, here $h = 6$, the number of distinct $\overline{\mathtt{S}}'[i]s$ starts to decrease. This speeds-up the execution due to increased number of cache hits for $\mathtt{T}'$ values.

Henceforth, we will consider two running times for MONOPOLY. The first one represents the running time of a single (or first) query (denoted by *total time*); hence it includes the times spent by all the phases of MONOPOLY (including checking). The other one (denoted by *filtering+checking*) represents the running time of a re-query on the same database (including checking but excluding text preprocessing).

5.1 Varying the Number of Tracks

We started our comparison by measuring the effect of the parameter h, that is, the number of tracks in the text. Fig. 5.3 shows that the number of candidates grows much more rapidly than the number of proper occurrences, as h increases (note the logarithmic scale). From around 350 at $h = 3$, the number of candidates grows to around 11,000 at $h = 4$. However, for

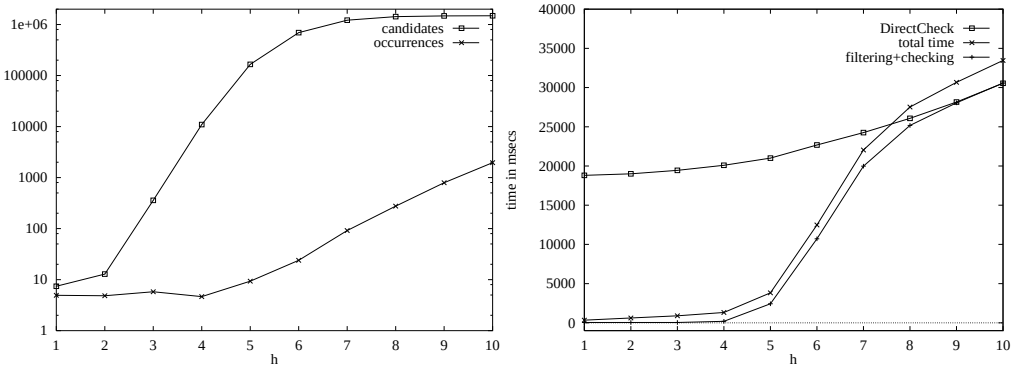

Fig. 5.3: The average effect of h ($m=12; n=1,484,940; |\Sigma'|=12$). Numbers of candidates and proper occurrences (on the left, log scale). Times for a first query (total time) and for re-queries (filtering+checking) of MONOPOLY and for DIRECTCHECK (on the right).

MONOPOLY the first query is faster than for DIRECTCHECK, until h becomes larger than 7. Re-queries with MONOPOLY are clearly faster than with DIRECTCHECK for $h < 9$.

5.2 Varying the Length of the Pattern

Next we experimented on the influence of the length of the pattern (see Fig. 5.4). As the pattern becomes longer, the number of occurrences decreases notably faster than the number of candidates. However, MONOPOLY is considerably faster than DIRECTCHECK with these parameter settings. The left graph illustrates two interesting phenomena. Firstly, the weak discriminating power of short patterns has a clear consequence to the performance of MONOPOLY; the shorter the pattern is the more often the slow checking routine has to be called. Secondly, as mentioned in Subsection 4.1,

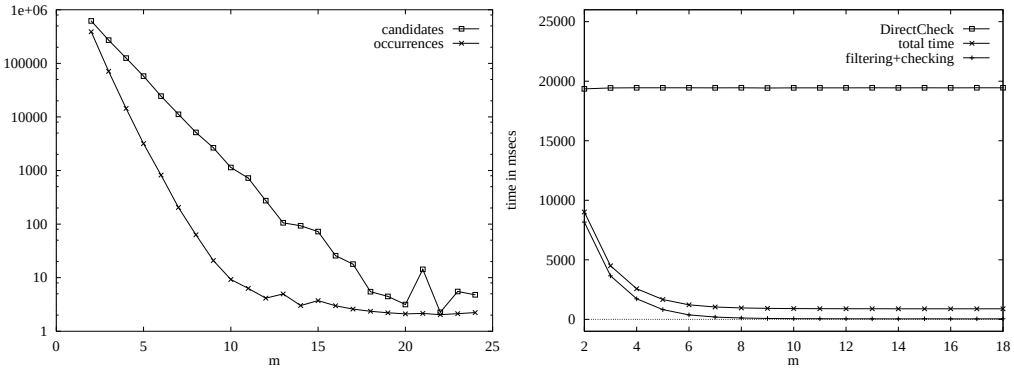

Fig. 5.4: The average effect of m ($h=3; n=1,484,940; |\Sigma'|=12$). Numbers of candidates and proper occurrences (on the left, log scale). Times for a first query (total time) and re-queries (filtering+checking) of MONOPOLY and for DIRECTCHECK (on the right).

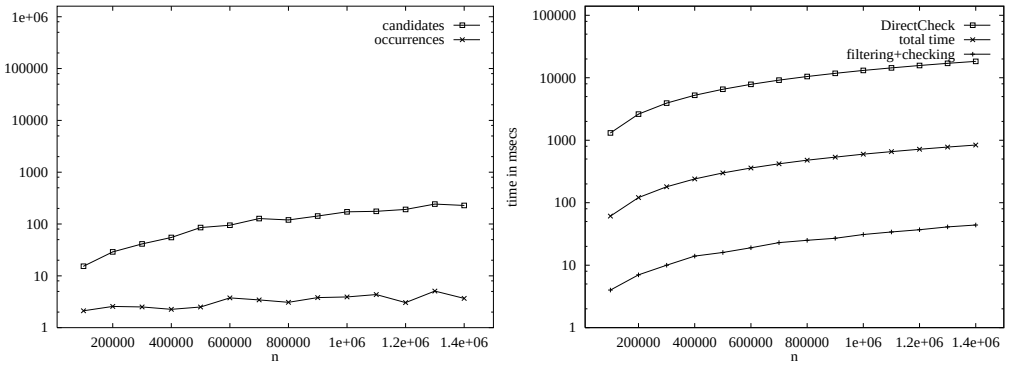

Fig. 5.5: The average effect of $n\,(h = 3; m = 12; |\Sigma'| = 12)$. Numbers of candidates and proper occurrences (on the left, log scale). Times for a first query (total time) and re-queries (filtering+checking) of MONOPOLY and for DIRECTCHECK (on the right, log scale).

the running time of DIRECTCHECK does not depend on the pattern length.

5.3 Varying the Length of the Text

Of all our experiments, the most significant difference between the performances of MONOPOLY and DIRECTCHECK was found when varying the size of the database (see Fig. 5.5). Again, the number of candidates grows faster than the number of occurrences, but there is a significant difference in running times. Although the first query of MONOPOLY takes more time than the re-queries, it is faster than the same query with DIRECTCHECK. Because DIRECTCHECK's running time seems to grow linearly as the database grows (note the log scale), the longer the text is the larger the difference between the performances of the two approaches will be.

5.4 Varying the Size of the Alphabet

Finally, we made experiments on the parameter $|\Sigma'|$. Note that, so far in the experiments, we have used Σ'_{12} with MONOPOLY, while DIRECTCHECK always uses the alphabet Σ'_{255}. It can be seen in Fig. 5.6, that Σ'_{12} works well with MONOPOLY. When observing the number of candidates, the setting $|\Sigma'| = 12$ meets a salient local minimum. Moreover, increasing the size of the alphabet from 12, the number of candidates does not become lower than that before $|\Sigma'| \geq 18$. Naturally this curve of candidates depends on the interval distribution within the chords, but we believe that the distribution we used is typical enough. However, when $|\Sigma'|$ becomes greater than 20, MONOPOLY's performance starts to get slower due to the $O(|\Sigma'| \cdot 2^{|\Sigma'|})$ complexity of the pattern processing phase (in a 600 MHz Pentium III the decreasing of the speed started at $|\Sigma'| = 18$, already).

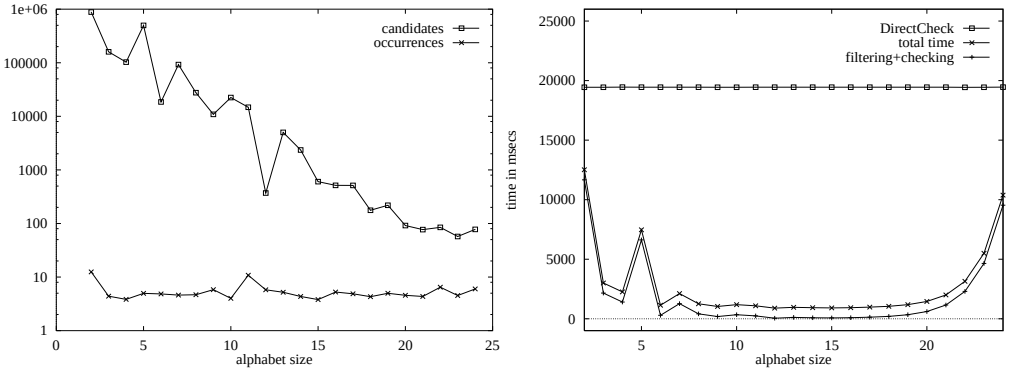

Fig. 5.6: The average effect of $|\Sigma'|$ ($h=3; m=12; n=1,484,940$). Numbers of candidates and proper occurrences (on the left, log scale). Times for a first query (total time) and re-queries (filtering+checking) of MONOPOLY, and for DIRECTCHECK (on the right).

6. Concluding Remarks

We have adapted the SHIFTOR algorithm to music retrieval by introducing three modifications for two distinct variations of the multi-track string matching problem. A summary of the algorithms is given in Table I.

First, we suggested the SHIFTORAND algorithm for the original multi-track string matching problem. Then, we presented two SHIFTOR modifications for transposition invariant multi-track string matching. The INTERVAL-MATCHING filter works on-line, while our main contribution, the MONOPOLY filter, has been optimized to work with static music databases. The results of these filters should be checked in order to find the proper occurrences among the candidates. This can be done, for instance, by using DIRECTCHECK.

We made extensive experiments with MONOPOLY on studying the effect of parameters $m, n, |\Sigma'|$, and h to its performance. In the experiments, a particular alphabet Σ'_{12} (of size 12) corresponding to a musical octave equivalence was found to work very well with MONOPOLY. It was also interesting to observe the consequence of varying the value of h. Although it does not have a direct consequence to the performance of the filtering phase, it has an effect to the efficiency of the filtration, and therefore, to the performance of the checking phase; the larger the h the more false positive hits. Due to our experiments, MONOPOLY, our main contribution, clearly outperforms the straightforward DIRECTCHECK whenever h is reasonably low.

There are several possibilities to refine our algorithms. For INTERVAL-MATCHING we could have used the octave equivalence assumption, as well. Moreover, to compute the set D, one could use a similar method to that that we used in MONOPOLY to compute the chord bitvectors in time $O(nh\lceil\frac{m}{w}\rceil)$ (actually, this is assumed in the time requirement given for INTERVAL-MATCHING in Table I). For MONOPOLY the core can be replaced by the BNDM algorithm of Navarro and Raffinot [1998].

In the both filtering algorithms, a further, practical improvement for the

TABLE I: A summary of the requirements of the presented algorithms.

	TIME		SPACE
	preproc.	running	
Multi-track string matching			
SHIFTORAND	-	$O(nh\mu)$	$O((\lvert\Sigma\rvert+1)\mu)$
Transposition invariant multi-track string matching			
INTERVALMATCHING (filter)	-	$O(nh\mu)$	$O((\lvert\Sigma\rvert+2)\mu+h^2)$
MONOPOLY (filter)	$O(nh\lceil\lvert\Sigma'\rvert/w\rceil)$	$O(n\mu+m+d)$	$O(n\lceil\lvert\Sigma'\rvert/w\rceil+c)$
DIRECTCHECK	-	$O(nhm)$	$O(m\lceil\lvert\Sigma\rvert/w\rceil)$

$m = |p|$, $n = |S|$, $\mu = \lceil m/w \rceil$
h: number of parallel tracks, w: size of machine word in bits,
Σ: underlying (absolute) alphabet, Σ': underlying (relative) alphabet,
$c = |\Sigma'|\lceil|\Sigma'|/w\rceil + (2^{|\Sigma'|} + |\Sigma'|)\mu$, $d = \mu|\Sigma'| \cdot 2^{|\Sigma'|}$.

performance may be obtained by observing the distribution of the symbols (intervals) and by searching first for the least frequent substring of the pattern. In the case of static database, the distribution may be calculated in advance, while in the on-line case an approximation of the distribution may be used. A similar trick may naturally be used also with long patterns (for which $m > w$); the filter is used for locating such substrings (of length m') of p for which $m' \leq w$ and whose sum of the interval probabilities is the lowest. Naturally, this trick may be used for searching polyphonic patterns with our algorithms, as well.

Acknowledgements

The authors are indebted to the insightful and valuable comments of the referees.

References

BAEZA-YATES, R. AND GONNET, G. H. 1992. A new approach to text searching. *Communications of the ACM 35*, 10, 74–82.

BOYER, R. AND MOORE, S. 1977. A fast string searching algorithm. *Communications of the ACM 20*, 10, 762–772.

CROCHEMORE, M. AND RYTTER, W. 1994. *Text Algorithms*. Oxford University Press.

CZUMAJ, A., CROCHEMORE, M., GASIENIEC, L., JAROMINEK, S., LECROQ, T., PLANDOWSKI, W., AND RYTTER, W. 1994. Speeding up two string-matching algorithms. *Algorithmica 12*, 4/5, 247–267.

DOVEY, M.J. 2001. A technique for "regular expression" style searching in polyphonic music. In *the 2nd Annual International Symposium on Music Information Retrieval (ISMIR'2001)*. Bloomington, IND, 179–185.

FORTE, A., EDITOR. 1962. *Tonal Harmony in Concept and Practice*. Holt, Rinehardt and Winston, New York.

GUIBAS, L. J. AND ODLYZKO, A. M. 1980. A new proof of the linearity of the Boyer-Moore string searching algorithm. *SIAM Journal of Computing 9*, 4, 672–682.

HOLUB, J., ILIOPOULOS, C.S., AND MOUCHARD, L. 2001. Distributed string matching using finite automata. *Journal of Automata, Languages and Combinatorics 6*, 2,

191–204.

LEMSTRÖM, K. 2000. *String Matching Techniques for Music Retrieval*. PhD thesis, University of Helsinki, Department of Computer Science.

LEMSTRÖM, K. AND TARHIO, J. 2000. Detecting monophonic patterns within polyphonic sources. In *Content-Based Multimedia Information Access Conference Proceedings (RIAO'2000)*. Paris, 1261–1279.

MIDI MANUFACTURERS ASSOCIATION. 1996. *The Complete Detailed MIDI 1.0 Specification*. Los Angeles, California.

NAVARRO, G. AND RAFFINOT, M. 1998. A bit-parallel approach to suffix automata: Fast extended string matching. In *Proceedings of Combinatorial Pattern Matching*. Piscataway, N.J., 14–33.

PARNCUTT, R. 1989. *Harmony: A Psychoacoustical Approach*. Springer-Verlag.

WIGGINS, G.A., LEMSTRÖM, K., AND MEREDITH, D. 2002. SIA(M): A family of efficient algorithms for translation-invariant pattern matching in multidimensional datasets. (submitted)