# Searching Monophonic Patterns within Polyphonic Sources

**Kjell Lemström[*] & Jorma Tarhio[‡]**

[*] Department of Computer Science
University of Helsinki, P.O. Box 26
FIN-00014 University of Helsinki, Finland
klemstro@cs.helsinki.fi

[‡] Department of Computer Science
University of Joensuu, P.O. Box 111
FIN-80101 Joensuu, Finland
tarhio@cs.joensuu.fi

**Abstract**

The string matching problem for strings in which one should find the occurrences of a pattern string within a text, is well-studied in the past literature. The problem can be solved efficiently, e.g., by using so-called *bit-parallel* algorithms. We adapt the bit-parallel approach to music information retrieval. We consider a situation where the pattern is monophonic and the text (the musical *source*) is polyphonic, that is, the pattern is a sequence of symbols, while the source is a sequence of *sets* of symbols (i.e., chords). The application of the bit-parallel approach is straightforward, if the *transposition invariance* is not allowed in the matching. However, the problem becomes trickier with transposition invariance, a necessary property in practice. We present algorithms for both cases. Our main contribution is a linear-time transposition-invariant filtering-algorithm for static music databases. We show by experiments that, if the average size of the chords keeps reasonably low, our filtering method clearly outperforms a straightforward approach.

## 1 Introduction

Efficient solutions for the problem of finding substrings of a text $s_1 \cdots s_n$ similar to a given query pattern $p_1 \cdots p_m$, are required in many information retrieval applications. The special case of *exact matching* can be solved by various algorithms (see e.g., Boyer & Moore 1977, Knuth, Morris & Pratt 1977). An inspiring and efficient solution is the shift-or algorithm developed by Baeza-Yates & Gonnet (1992). This algorithm applies *bit parallelism*, and achieves a time complexity of $O(n\lceil m/w \rceil)$, where $w$ is the size of the computer word (e.g. 32 or 64 bits, in practice). Thus, if $m \leq w$ holds, the shift-or algorithm runs in linear time.

In this paper we consider a rather new application domain for string matching, that is, music information retrieval (MIR). As the music can be presented symbolically (with some restrictions) as linear discrete sequences (e.g. by MIDI encoding), general string matching algorithms can be adapted for finding the occurrences of a musical query pattern within a music database (see e.g., Ghias, Logan, Chamberlin & Smith 1995, McNab, Smith, Bainbridge & Witten 1997, Lemström & Laine 1998). Since in this application domain it would be somewhat confusing to use the word 'text' for the string where the pattern is to be searched, we will henceforth mainly speak of *source* instead. We will also use the term database, reminding of the fact that it is straightforward to extract such a source string from a music database.

When a musical query pattern is to be found within a database of 'real' music, a particular problem arises: real music is practically always polyphonic, which means that several notes are to be played simultaneously. The MIR algorithms presented in literature usually run with carefully established databases containing only monophonic sources, or one of the simultaneous notes (e.g. the highest) is heuristically selected to represent the musical event. Here we consider unrestricted monophonic-polyphonic search. The problem is to find all locations in a polyphonic source that contain the given monophonic pattern (exactly with an arbitrary transposition). To find a matching pattern, any note of each chord can be selected. Since an exhaustive evaluation of all possible melody lines, that the polyphonic source contains, would be very slow, faster solutions are needed. Some solutions for the problem have been presented in the literature (Crawford, Iliopoulos & Raman 1998, Uitdenbogerd & Zobel 1998, Dovey 1999, Holub, Iliopoulos, Melichar & Mouchard 1999).

Since western music melody recognition is based rather on the *intervals* between the consecutive pitches than on the absolute pitch sequence, *transposition invariance* is a natural and useful property of a MIR algorithm. Though using the absolute pitch values, the pattern $p = \mathtt{c_1\,d_1\,e_1\,c_1}$, for example, does not match the prefix of the source $s = \mathtt{d_1\,e_1\,f_1\sharp\,d_1}\cdots$, a MIR algorithm allowing transposition invariance would find a transposed occurrence of $p$ at $s_1 \cdots s_4$, due to a common interval sequence (i.e. in *semitones*: $2\ \ 2\ \ -4$). In MIR algorithms the transposition invariance is achieved simply by working with intervals, that is, with the differences of the pitch values of successive notes.

We present the polyphonic *source* formally as a sequence $S = S_1S_2 \cdots S_n$ of sets. Each $S_i$ models a chord and is formally a subset of the basic alphabet $\Sigma$. We assume for simplicity, that $\Sigma = \{0, 1, \dots, r\}$ for some $r$, that is, the notes (pitches) are encoded as small integers. In the MIDI representation, for example, $r$ equals 127. The conversion from the actually used music notation to $\Sigma$ should be obvious, and we will not fix the details.

Some additional notation is necessary. The size of $\Sigma$ is denoted $|\Sigma|$. By $q$ we denote the size of the largest chord in $S$, that is, $q = \max\{|S_i| \mid 1 \le i \le n\}$. If $|S_i| = q$ for all $i$, then we denote $|S| \equiv q$.

Given a source $S = S_1 \cdots S_n$ and a (monophonic) pattern $p = p_1 \cdots p_m$ where each $p_i \in \Sigma$, our problem of finding monophonic patterns within polyphonic sources is formulated as follows: Find all positions of $S$ that have an *occurrence* of $p$. By definition, there is an occurrence of $p$ at position $j$, if $p_i \in S_{j+i-1}$ holds for $i = 1, \dots, m$, and there is a *transposition invariant occurrence* of $p$ at position $j$, if there is an integer $d$, such that each $(p_i + d) \in S_{j+i-1}$.

For the problem given above, we consider three settings (out of the four possible combinations) involving choices between using transposition invariance or not, and on-line/off-line searching. In off-line searching the source is preprocessed to speed-up the subsequent searches. As solutions, we present algorithms that can be embedded in an Internet search engine for music information retrieval. The settings are:

1. on-line searching, without transposition invariance (Algorithm 2);
2. on-line searching, with transposition invariance (Algorithm 3);
3. off-line searching, with transposition invariance (Algorithm 4).

At the end of Section 2, we present the original shift-or algorithm, and show how it is adapted to setting 1 (Algorithm 2). In Section 3, we deal with transposition invariant settings 2 and 3. Firstly, we introduce a straightforward solution based on the naive pattern matching algorithm. It runs in time $O(nq(q+m))$. Then we apply the bit parallel approach to settings 2 and 3. Our Algorithm 3 for setting 2 is short and elegant, and serves as a gradual step towards Algorithm 4, which in turn is more complicated.

Algorithm 3 is an on-line filtering-method that works in two phases. The first phase (scanning) finds starting positions of potential occurrences, which we call *candidates*. Every proper occurrence is also a candidate, in other words, the filter will not miss any proper occurrence. In the second phase, each of the candidates is checked whether it is a proper occurrence or not. Typically, a filtering method becomes efficient, when the scanning phase is fast, and the number of found candidates is not too large as compared to the number of proper occurrences. The checking phase might be somewhat slower.

Algorithm 4 is an off-line filtering-method for static databases. By static database we mean that the source is not updated between the queries. Hence, before scanning we can build indexing structures in order to speed up the matching. An off-line method can be designed to be more efficient than an on-line one in the case where several consecutive queries are made to the same source. The off-line Algorithm 4 has three phases. The source is preprocessed in the first phase. The other phases are scanning and checking like in an on-line method. Preprocessing is made only once but scanning and checking are done for every query pattern.

If the pattern is short enough as compared to the length of a computer word[1], i.e. if $m \leq w$, the scanning phases of Algorithms 2 and 3 run in time $O(nq)$ and $O(nq^2)$, respectively. However, we show that in the case of a static music database, by spending $O(nq)$ time for source preprocessing, scanning can be done in linear time $O(n)$. To our knowledge, our solution is the fastest that has been presented for setting 3.

In Section 2 we explain our encoding of the musical sequences, present some related work, and discuss the concepts of *interval classification* and *octave equivalence*. The interval classification is eventually used to make our final algorithm more efficient. To our point of view, octave equivalence is a particularly interesting special case of interval classification.

Before concluding the paper, we review our extensive experiments (Section 4). Since the earlier algorithms (described in Section 2) are not designed for the transposition invariant case, we compared the performance of Algorithm 4 with that of the straightforward algorithm in our experiments. We studied the effect of various parameters to the performance.

## 2 Background

### 2.1 Representing Music Events

We represent the musical excerpts as sequences of symbol sets, which correspond to chords of notes. In the representation, the ordering of the note onsets (stored as the pitch information) is observed. If a previous note is still on while another has its onset, the onset of the previous note is repeated. We completely omit all the other musical information, e.g. empty events (rests).

Let $S = S_1\, S_2 \cdots S_n$ be a musical sequence, where each $S_i$ is a subset of an underlying alphabet. The set (or chord) $S_1$ contains the pitch information of the notes occurring first in the corresponding piece of music, $S_2$ contains the pitches occurring next, etc. The pitch components within the chords are ordered from low to high. The lowest component of a chord $S_i$ is called the base of the chord, and is denoted by $\mathtt{base}(S_i)$.

As the underlying alphabet for the music sequences, we can use, e.g., the music theoretical names for the pitches $(\ldots, \mathtt{c}_1, \mathtt{c}_1\sharp, \mathtt{d}_1, \ldots)$ or MIDI values. For example, the sample melody given in Fig. 1 is encoded by the following sequence: $S = \langle \mathtt{f}_1, \mathtt{a}_1, \mathtt{c}_2\rangle\langle \mathtt{e}_1, \mathtt{g}_1, \mathtt{c}_2\rangle\langle \mathtt{d}_1, \mathtt{f}_1\rangle\langle \mathtt{c}_1, \mathtt{e}_1, \mathtt{c}_2\rangle$ (here we have used the angle braces in order to show the chords).

[1]Thus we can omit the $\lceil \frac{m}{w} \rceil$ factor from the complexity.

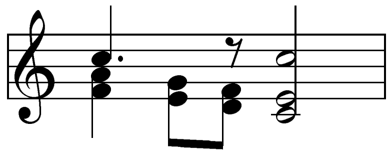

Figure 1: A sample melody.

## 2.2 Music Information Retrieval

As already mentioned, it is important to be able to deal with polyphonic databases in music information retrieval. The lack of such MIR algorithms is pointed out in several papers (e.g., Blackburn & DeRoure 1998, Tseng 1999), and it would be too slow to extract all possible sequence combinations 'on-the-fly' and match the pattern against them all. However, there have been only few methods presented for locating either monophonic or polyphonic patterns within polyphonic sources.

Uitdenbogerd & Zobel (1998) used a music psychological approach in order to overcome the problem. Their idea was to reduce the 2-dimensional problem into a 1-dimensional one. With eight human listeners they experimented with four different heuristics which extract monophonic 'melodies' from polyphonic music: (1) Whenever a set of notes occurs, the highest note is chosen. (2) For every MIDI channel first heuristic 1 is applied, and then the channel with the highest average pitch is chosen. (3) Again, for every MIDI channel first heuristic 1 is applied, then the channel with the highest first-order predictive entropy is chosen. (4) Each channel is first split into parts heuristically, and then the part with the highest entropy is chosen.

In their experiments, heuristic 1 outperformed the others. However, in many cases, e.g. in choral music, the highest voice does not necessarily contain the melody; it is even possible that the melody is distributed across several distinct voices. Furthermore, the user may want to find a musical excerpt that is not at all considered to be melody. Thus, the capability to cope with all possible combinations of the polyphonic source, would be useful.

Crawford, Iliopoulos & Raman (1998) presented a sketch of an algorithm based on the Aho & Corasick (1975) automaton: First an automaton accepting all the prefixes of a pattern is created. Then, at each point of time, all occurrences of the pattern prefixes are monitored; an existing prefix is either extended or voided depending on whether the next symbol occurs at that point of time. The time complexity of the algorithm is $O(nm)$. Crawford et al. also claimed that the algorithm can be adapted to the case of transposition invariance using only $|\Sigma|$ extra locations. However, the technical details of their transposition invariant algorithm are not available.

Dovey (1999) made experiments with a straightforward implementation for locating patterns (which could also be polyphonic in his algorithm) within polyphonic sources. He used a 'piano-roll' representation for the music (only 'Note On' messages of MIDI were considered): every point of time, where a Note On occurs, is represented as a bit string of 88 pitches. Dovey's algorithm has an additional parameter $t$ indicating the allowed spacing between consecutive elements of the pattern within the source: By setting $t = 0$ no spacing is allowed, thus for example, an occurrence of a monophonic pattern $p$ is found, if and only if $p_i \in S_{j+i-1}$ holds for $i = 1, \ldots, m$ $(1 \leq j \leq n - m + 1)$. If $t$ equals 1, an occurrence of *maximal span* is found, if $p_i \in S_{j+2(i-1)}$ $(1 \leq j \leq n - 2(m+1))$. The algorithm tries to find $p$ recursively in every possible location, with every allowed spacing. We analyzed the method and concluded that the worst case time complexity is $O(nm\,(t+1)^{(m-1)})$. Furthermore, to deal with transposition invariance the algorithm has to do 88 reiterations.

Recently, Holub, Iliopoulos, Melichar & Mouchard (1999) have independently presented ideas that are closely related to our work. However, they have not considered the transposition invariance. They use a bit-parallelism approach in order to find (i) multi-patterns within monophonic sources (the original shift-or algorithm); (ii) monophonic patterns within polyphonic sources (similar to our Algorithm 2); and (iii) multi-patterns within polyphonic sources. Letting $w$ be the number of bits in the machine word and assuming that $m \leq w$, in the case of exact matching the algorithm (iii) requires $O(rm + |\Sigma'|)$ time for the preprocessing, where $r$ and $\Sigma'$ are the number of the patterns and the set of symbols used in the pattern, respectively. The actual running time is $O(nq)$ and the algorithm requires $O(|\Sigma'|)$ space. In their paper they have also considered the $k$ *mismatches problem*.

### 2.3 Interval Classification and Octave Equivalence

Downie has recently presented results of a comprehensive statistical analysis on which parameters affect the performance of a MIR query (Downie 1999*a*, Downie 1999*b*). To our point of view, the most interesting analysis concerned *interval classification*. He made experiments with classifications of four sizes: 3 (C3), 7 (C7), 15 (C15), and intervals as they appear originally (CU). The classifications were constructed as follows: In case C3, only the *contour* of the music was considered; the interval alphabet $\Sigma_3$ was comprised of a set $\{b, a, B\}$, where the characters of the alphabet were assigned to downward, same as previous, and upward intervals, respectively. In case C7, $\Sigma_7 = \{d, c, b, a, B, C, D\}$, where $d$ was assigned to intervals of $-3$ semitones at the most; $c$ to $-2$; $b$ to $-1$; $a$ to 0; $B$ to 1; $C$ to 2; and finally, $D$ was assigned to intervals of 3 semitones at least. Accordingly, in case C15, the alphabet $\Sigma_{15}$ was comprised of a set $\{h, g, f, e, d, c, b, a, B, C, D, E, F, G, H\}$, and $h$ was assigned to intervals of $-7$ semitones at the most, etc.

As has also been observed earlier (see e.g., Uitdenbogerd & Zobel 1998, Lemström & Laine 1998), the discriminating power of C3 is much too low when dealing with databases large enough. Dovey found out that C15 and CU are superior to C7. However, he did not find any significant difference between C15 and CU. Furthermore, he measured the frequency of the intervals (Downie 1999*a*, p. 78) occurring in a database of 9,354 folksongs. In his measurement the great majority of the intervals occurred between the scale $-5, \ldots, 5$ of semitones.

An interval classification distinguishing pitches only within an octave[2] is very natural in many respects. As stated by Forte (1962), *octave equivalence* is "one of the most fundamental axioms of tonal music". The main reason why the octave has become the equivalence interval in music, is that combinations of them are *consonant*. This is not true for any other interval (Parncutt 1989). Moreover, due to Pinker (1997, p. 530) not only humans, but all the mammals perceive pitches to have the same quality of pitch, if they are separated by an octave (or by multiple octaves).

The octave equivalence in this paper means that all intervals should be reduced to $0, 1, \ldots, 11$ semitones. Technically this is achieved by using alphabet $\{0, 1, \ldots, 11\}$ and replacing the actual interval value $h$ (in semitones) by the value $h \bmod 12$. Thus an interval of 7 semitones upwards equals the interval of 5 semitones downwards, for instance. Assuming that this classification would be almost as good as the classification C15, the discriminating power should be sufficient. Furthermore, as the original interval distribution can be supposed to approximately have the shape of a normal distribution that is centered around the zero interval (due to Downie's experiments), the distribution of the intervals becomes different when octave equivalence is applied; the peak of the original distribution is divided into two parts which are located at the two ends of the new value range. Moreover, since the variance of the distribution is rather small, only the tails of the divided two parts should collide, thus resulting to a more uniform distribution of intervals than that of the original. Actually, this observation may be of use in some IR applications.

[2]The interval between any two pitches which are separated by 12 semitones.

T:

| | a | b | c |
|---|---|---|---|
| a | 0 | 1 | 1 |
| a | 0 | 1 | 1 |
| b | 1 | 0 | 1 |

| | E | shiftleft(E) | T[a] | E | shiftleft(E) | T[b] | E | shiftleft(E) | T[c] | E | shiftleft(E) | T[a] | E | shiftleft(E) | T[a] | E | shiftleft(E) | T[a] | E | shiftleft(E) | T[b] | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| a | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| b | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |

Figure 2: An example of the shift-or algorithm 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; note that shifting is done downwards. The found occurrence is emphasized (with a white circle).

### 2.4 Shift-Or Algorithm

Next we present the original shift-or algorithm developed by Baeza-Yates & Gonnet (1992). Henceforth we will use the symbols $\vee_b$ and $\wedge_b$ representing the bitwise operators `or` and `and`, respectively. Moreover, as we use a Pascal-like notation, by $p[j]$ and $s[i]$ we denote $p_j$ and $s_i$, respectively. The algorithm is as follows:

**Algorithm 1 (shift-or)**
**begin**
*1.* **for each** $a \in \Sigma$ **do** $T[a] := 2^m - 1$;
*2.* **for** $j := 1$ **to** $m$ **do** $T[p[j]] := T[p[j]] - 2^{j-1}$;

*3.* $E := 2^m - 1$;
*4.* **for** $i := 1$ **to** $n$ **do begin**
*5.* $E :=$ `shiftleft`$(E) \vee_b T[s[i]]$;
*6.* **if** $E.m = 0$ **then** `write`$(i)$
**end**

Lines 1–2 and 3–6 of Algorithm 1 form two subphases, which we call pattern preprocessing and core phases, respectively. Together they form the scanning phase (in this case the algorithm does not have any other phases). First, for each symbol appearing in the pattern, the pattern preprocessing phase creates a bit-mask as a column of table $T$. At the core phase, a zero bit is released (by the binary `shift` 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 by the $\vee_b$ operator. Whenever a zero bit reaches the level $m$, an occurrence of the pattern has been found; this is reported on line 6. We illustrate in Fig. 2 how the algorithm works, and refer to Baeza-Yates & Gonnet (1992) for further information.

The exact time complexity of the pattern preprocessing phase is $O(\lceil \frac{m}{w} \rceil |\Sigma| + m)$, while the core of the algorithm runs in time $O(\lceil \frac{m}{w} \rceil n)$. The overall space requirement is $O(\lceil \frac{m}{w} \rceil m + \lceil \frac{m}{w} \rceil |\Sigma|)$.

**On Searching without Transposition Invariance.** The shift-or algorithm can be adapted with a minor modification to searching monophonic music patterns within polyphonic sources, if transposition invariance is not required. Actually, this is a dual of the string matching problem presented by Wu & Manber (1992); They considered cases where the positions of a pattern was allowed to contain a set of characters. In their case they modified the pattern preprocessing phase, while in our case each source position is allowed to contain a set of characters and the modified phase is the core phase. This is done by adding a bitwise `and` operation, which operates over all the pitches within a chord (the modification is grayed in the algorithm below).

**Algorithm 2 (on-line, without transposition invariance)**
**begin**
*1.* **for each** $a \in \Sigma$ **do** $T[a] := 2^m - 1$;
*2.* **for** $j := 1$ **to** $m$ **do** $T[p[j]] := T[p[j]] - 2^{j-1}$;

*3.* $E := 2^m - 1$;
*4.* **for** $i := 1$ **to** $n$ **do**
*5.* $\quad E := \texttt{shiftleft}(E) \vee_b (\bigwedge_b(\{T[a] \mid a \in S_i\}))$;
*6.* $\quad$ **if** $E.m = 0$ **then** $\texttt{write}(i)$
**end**

This idea was recently presented by Holub, Iliopoulos, Melichar & Mouchard (1999), as well. The main loop in lines $4-6$ takes $O(nq)$ time (when $m \leq w$), and an extra space of only one computer word is required. The extra space is used as a temporary storage, where the bitwise `and` operation can bring all the required zero bits. Actually, the algorithm can be formulated in such a way that the core runs in linear time. In that case, table $T$ contains a column for each chord in $S$ (instead of each character in $\Sigma$). Naturally, both the time complexity of pattern preprocessing and the overall space complexity increase noticeably from that of Algorithm 2.

## 3 Searching with Transposition Invariance

Our problem of searching monophonic music patterns within polyphonic music databases becomes trickier when allowing transposition invariance. In this section we present two fast filtering methods to solve the on-line and off-line versions of the problem. In the on-line method all preprocessing is done during the scanning phase, while in the off-line method, dealing with a static database, as much as possible is done in a separate source preprocessing phase to enable faster scanning.

We start by introducing a checking algorithm which can be used with our filtering methods. The same algorithm is used later as a reference method in our experiments. In the former use the algorithm checks only the candidates, and in the latter use the algorithm is a total method which checks all the source positions.

### 3.1 Checking Algorithm

Let us next describe ideas of our implementation of a checking algorithm capable of identifying the proper occurrences of a given query pattern, since some details of it are not trivial. The algorithm is based on the naive string matching algorithm (see e.g., Crochemore & Rytter 1994, p. 34). In the following, we call it Algorithm C, where C refers to checking.

For the sake of simplicity, let us assume that all the source positions are to be checked. The pattern is aligned with each position of the source, step by step. At each alignment the corresponding characters (or in our case intervals) are compared from left to right, one by one, until a mismatch occurs, or the end of the pattern is reached and an occurrence is reported. After that the pattern is shifted one position to the right.

At first, the algorithm has to check if the interval $p_2 - p_1$ equals any interval appearing between the chords $S_i$ and $S_{i+1}$, when matching $p_1 \cdots p_m$ against $S_i \cdots S_{i+m-1}$. Moreover, $p_2 - p_1$ may occur up-to $q$ times (because a chord can contain $q$ elements at the most) at one alignment, and each of the occurrences must be taken into account. At each alignment $i$, $1 \le i \le n-m+1$, Algorithm C calculates all the intervals appearing between $S_i$ and $S_{i+1}$, and stores the occurrences of $p_2 - p_1$ to a linked list. Having calculated all the possible first intervals, the linked list is checked. If it is not empty, i.e. $p_2 - p_1$ matches some $x_2 - x_1$ (where $x_1 \in S_i$ and $x_2 \in S_{i+1}$), the next step is to check if there is $x_3 \in S_{i+2}$ such that $p_3 - p_2$ matches $x_3 - x_2$. Intervals $p_j - p_{j-1}$ for $j = 4, \ldots, m$ are treated accordingly. Whenever a matching fails, the next non-checked occurrence of $p_2 - p_1$ in the linked list is tried for the matching.

Though the number of possible intervals between two consecutive chords is $q^2$, only $q$ of them at the most are the same. Therefore, if Algorithm C checks all the source positions, its time complexity is $O(n(q^2 + qm)) = O(nq(q+m))$ in the worst case, because the computation of intervals takes $O(q^2)$ and matching the pattern up to $q$ times takes $O(qm)$ for each source position.

We decided that if several occurrences of the pattern occur in parallel (each starting at the same position $S_i$), the position is reported only once. 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.

### 3.2 On-line Filtering Algorithm

Next we present a basic on-line filtering algorithm with transposition invariance. In the algorithm, the underlying alphabet is comprised of MIDI values $\{0, \ldots, 127\}$, and $T[d]$ is the bit-mask for the interval (difference) $d$; due to 128 different MIDI values the scale for $d$ is $-127, \ldots, 127$. Note that since we are considering intervals (because we want to achieve transposition invariance), we begin the loops with the value 2.

**Algorithm 3 (On-line filter with transposition invariance)**
**begin**
*1.* **for each** $d, d = -127, \ldots, 127$ **do** $T[d] := 2^{m-1} - 1$;
*2.* **for** $j := 2$ **to** $m$ **do** $T[p[j] - p[j-1]] := T[p[j] - p[j-1]] - 2^{j-2}$;

*3.* $E := 2^{m-1} - 1$;
*4.* **for** $i := 2$ **to** $n$ **do**
*5.* $\quad D := \emptyset$;
*6.* $\quad$ **for each** $a \in S_{i-1}$ *and* $b \in S_i$ **do** $D := D \bigcup \{b - a\}$;
*7.* $\quad E := \texttt{shiftleft}(E) \vee_b (\bigwedge_b \{T[d] \mid d \in D\})$;
*8.* $\quad$ **if** $E.m = 0$ **then** `check(`$i$`);`
**end**

By comparing Algorithm 3 with Algorithm 2, one can notice two important differences. Firstly, in Algorithm 3 we introduce a set $D$, where all the intervals between two consecutive chords are collected (line 6). These intervals are then used in the shifting similarly as in Algorithm 2. The other difference can be noticed in line 8. This algorithm is a filtering method, because it only checks that the candidate contains the intervals of the pattern in the correct order (we will formalize that more precisely in subsection 3.4). Thus Algorithm 3 calls a checking algorithm for every found candidate.

Clearly, the core of the algorithm runs in time $O(nq^2)$. 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 $q^2 + 1$; $q^2$ is needed for the set $D$, and one computer word for the bitwise `and` operation (as in Algorithm 2). Thus, the total space requirement is $O(m + q^2)$. The checking (line 8) can be done, e.g., by using Algorithm C.

### 3.3 Static Filtering Algorithm

When a music database is static, the source can be preprocessed in order to speed up the retrieving. The benefit of the preprocessing is considerable when a music database is an object for 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.

In Algorithm 4, to be presented, we assume, that $S$ and $p$ are given in an alphabet $\Sigma$. For example, $\Sigma$ could be the MIDI alphabet $\{0, \dots, 127\}$. Algorithm 4 will internally use another alphabet $\Sigma' = \{0, 1, \dots, |\Sigma'| - 1\}$ for intervals. For each interval $x$ we have a symbol $x \bmod |\Sigma'|$ in $\Sigma'$. Note that selecting $|\Sigma'| = 12$ means that we reduce the interval space in the octave equivalent way.

Before we present the algorithm, let us introduce some notations, data structures, and functions that we use. If the value of an integer variable is interpreted as a bit string, we denote this by using $b$ as subscript (e.g. $l_b$). We use two auxiliary arrays, an array $I$ for all the possible intervals[3] between two successive chords, and an array $L$ for tracking the locations of the intervals within the pattern. Every element of $I$ is a $|\Sigma'|$-bit vector, and every element of $L$ is a $m - 1$ bit vector. Furthermore, $L$ and $I$ have both $|\Sigma'|$ elements, one for each interval. We use a function, $\texttt{rcs}(a, b)$ (right circularshift), that shifts a bit vector $a$ by $b$ bits to the right in a circular manner. For example, if $a = 01010$, then $\texttt{rcs}(a, 1) = 00101$ and $\texttt{rcs}(a, 2) = 10010$. Moreover, recall that $\texttt{base}(S_i)$ denotes the base of the chord $S_i$.

**Algorithm 4 (LT-filter with transposition invariance for static $S$)**
**begin**

**for** $i := 2$ **to** $n$ **do**
  $\overline{S}[i] := 2^{|\Sigma'|} - 1$;
  **for each** $a \in S_i$ **do**
    $a := (a - \texttt{base}(S_{i-1})) \bmod |\Sigma'|$;
    **if** $\overline{S}[i].a = 1$ **then** $\overline{S}[i] := \overline{S}[i] - 2^a$;
  $\overline{S}[i] := (\overline{S}[i] \ \wedge_b \ (\bigwedge_{a \in (S_{i-1} \setminus \texttt{base}(S_{i-1}))} \texttt{rcs}(\overline{S}[i], (a - \texttt{base}(S_{i-1})) \bmod |\Sigma'|)))$;

**for** $k := 1$ **to** $|\Sigma'|$ **do** $\{I[k] := 2^{|\Sigma'|} - 1,\ I[k] := I[k] - 2^{k-1}\}$;
**for** $k := 1$ **to** $|\Sigma'|$ **do** $L[k] := 2^{m-1} - 1$;
**for** $j := 2$ **to** $m$ **do**
  $b := (p[j] - p[j-1]) \bmod |\Sigma'|$;
  $L[b] := L[b] - 2^{j-2}$;
**for** $l := 1$ **to** $2^{|\Sigma'|}$ **do**
  $T[l] := 2^m - 1$;
  **for** $k := 1$ **to** $|\Sigma'|$ **do**
    $\texttt{ivect} := I[k]$;
    **if** $\texttt{ivect} \vee_b l_b = \texttt{ivect}$ **then** $T[l] := T[l] \wedge_b L[k]$;

$E := 2^{m-1} - 1$;
**for** $i := 2$ **to** $n$ **do**
  $E := \texttt{shiftleft}(E) \vee_b T[\overline{S}[i]]$;
  **if** $E.m = 0$ **then** $\texttt{check}(i)$;
**end**

[3] Actually, the interval array ($I$) is only used to support understanding; each item of it can be computed in constant time when accessed.

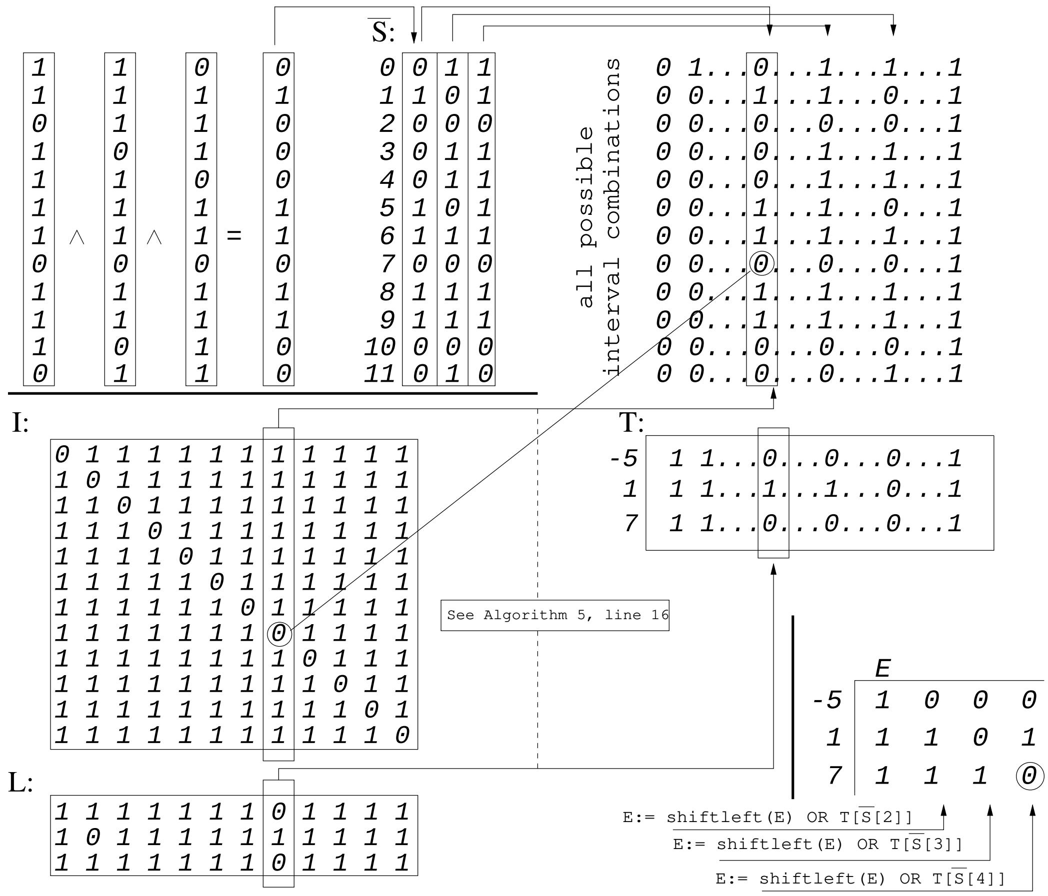

Figure 3: An illustration of the data structures of Algorithm 4, when $p = \mathtt{a}_1\, \mathtt{e}_1\, \mathtt{f}_1\, \mathtt{c}_2$ (thus the intervals between consecutive elements are $-5$; 1; and 7) and $S = \langle \mathtt{f}_1, \mathtt{a}_1, \mathtt{c}_2\rangle\langle \mathtt{e}_1, \mathtt{g}_1, \mathtt{c}_2\rangle\langle \mathtt{d}_1, \mathtt{f}_1\rangle\langle \mathtt{c}_1, \mathtt{e}_1, \mathtt{c}_2\rangle$ (the sample of Fig. 1).

We define $\overline{S}_i = \{(y - x) \bmod |\Sigma'| \mid x \in S_{i-1}, y \in S_i\}$ to represent all the intervals between $S_{i-1}$ and $S_i$. We denote $\overline{S}_2 \cdots \overline{S}_n$ by $\overline{S}$. In the algorithm, each $\overline{S}_i$ is represented as a $|\Sigma'|$-bit vector.

Let us look closer at the details of this algorithm. The algorithm has three phases: the source preprocessing phase (lines 1–6), the scanning phase (lines 7–19), and the checking phase (line 20). The scanning phase contains pattern preprocessing (lines 7–16) as its subphase. If there are several patterns to be searched for, the source preprocessing phase is executed only once. The checking phase can be done, e.g., with Algorithm C. Line 4 is a good example to observe how the transposition invariance and octave equivalence (in the case of $|\Sigma'| = 12$) are achieved. The transposition invariance is achieved by storing intervals to $\overline{S_i}$, i.e. storing the differences $(a - \mathtt{base}(S_{i-1}))$. At the end of the same line the expression $\bmod\ |\Sigma'|$ reduces the size of the alphabet to $|\Sigma'|$, and determines the number of the bits in each $S_i$.

In the source preprocessing phase, a bit vector $\overline{S}_i$ is formed for each $i$. In the beginning, every $\overline{S}_i$ is filled with ones on line 2. After that the 'shape' of the chord $S_i$ is stored into $\overline{S}_i$; each element of $S_i$ brings a zero bit to $\overline{S}_i$. At this point, the place of the zero bit is determined by the interval from the base of $S_{i-1}$. An example is given in Fig. 3; see the leftmost vector in the upper section (note that the interval 0 is on top in the vectors). The final form of $\overline{S}_i$ is achieved by applying the bitwise and operation on this vector and $|S_{i-1}| - 1$ other vectors which are got from the first one by shifting it according to other pitches in $S_{i-1}$. Thus the number of zero bits in $\overline{S}_i$ (denoted by $c$) is restricted as follows: $1 \le c \le \min\{|S_{i-1}| \cdot |S_i|, |\Sigma'|\}$.

The goal of the pattern preprocessing phase is to build a table $T$ of size $2^{|\Sigma'|} \times (m-1)$ bits, in linear time $O(n)$. This array $T$ has a column (bit-mask) for every possible interval combination, while the corresponding original shift-or table needed a column for every symbol appearing in the pattern. However, the zero bits are assigned to the correct locations analogously. The phase is started by the rows 7 and 8, which builds the aforementioned auxiliary array $I$, and initializes the array $L$, respectively. Then, the location table $L$ is computed (lines 9–11). A single zero is assigned on every row $L[j]$ according to the interval $p_{j+1} - p_j$. Thus the table can be used to track the positions of the intervals appearing within the pattern $p$. Then, in lines 12–16 every $I[k]$ (representing the intervals) is slid over all the possible interval combinations; for example, let $I[k]$ occur at interval combination $l$ (see the enboxed text in the middle of Fig. 3, and line 16 of Algorithm 4). Then the zero bit appearing in the corresponding location vector $L[k]$ is copied (by $\vee_b$ operation) to the table $T[l]$, for which one entry for each interval combination is allocated. When all the intervals have been slid over all the interval combinations, the table $T$ is complete, and the core phase, similar to the original shift-or algorithm but applied to $\overline{S}$, can be started.

### 3.4 Correctness and Analysis of Algorithm 4

We start considering the correctness of Algorithm 4 by proving the correctness of the source preprocessing phase, thus proving that $\overline{S}$ is correctly formed. We divide the proof in two subcases considered by the following two lemmas. In the following, the interval between the base of the chord $S_{i-1}$ and an element $y$ within the chord $S_i$ is denoted by $r(y)$, where $r(y) = y - \texttt{base}(S_{i-1})$. The distance from the base pitch within the same chord is denoted by $r_0(y)$, i.e. , $r_0(y) = y - \texttt{base}(S_i)$.

**Lemma 1** *Let $x \in S_{i-1}$ and $y \in S_i$ for some $i$, $2 \leq i \leq n$. If $j = (y - x) \bmod |\Sigma'|$ holds, then $\overline{S}_i.j$ is a zero bit.*

**Proof.** The array $\overline{S}$ is computed during the source preprocessing phase. There are two cases to be considered: (i) $x = \texttt{base}(S_{i-1})$ and (ii) $x \neq \texttt{base}(S_{i-1})$.

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

(ii) According to case (i), $\overline{S}_i$ holds all the intervals between $\texttt{base}(S_{i-1})$ and $\{S_i\}$ before line 6 is executed. Let $y$ be an arbitrary pitch within $S_i$, $d = y - \texttt{base}(S_{i-1})$, and $e = y - x$. Now the difference of $e$ and $d$ is $\texttt{base}(S_{i-1}) - x$, which by definition equals to $-r_0(x)$. Thus, since $d$ has already been stored in $\overline{S}_i$, $e = d - r_0(x)$ can be stored by assigning a zero bit at the location $j = e \bmod |\Sigma'|$. This is done in line 6 by the $\texttt{and}$ operation with $\texttt{rcs}(\overline{S}_i, r_0(x))$. Again $\overline{S}_i.j = 0$ holds.

$\square$

**Lemma 2** *If $j \neq (y - x) \bmod |\Sigma'|$ holds for every pair $x \in S_{i-1}$ and $y \in S_i$, then $\overline{S}_i.j$ is one.*

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

□

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

**Theorem 3** *$\overline{S}$ is correctly formed.*

Let us continue by proving that the pattern preprocessing phase works correctly, and thus that table $T$ is correctly formed.

**Lemma 4** *Let $j$ be an integer, $i$, $2 \leq i \leq m$. $T[l].j = 0$ holds, if and only if $l_b.k$ is zero and $k = (p_j - p_{j-1}) \bmod |\Sigma'|$ holds for some $k$.*

**Proof.** It is sufficient to consider only the preprocessing of the pattern (lines 7–16). Let us assume that $l_b.k$ is zero and $k = (p_j - p_{j-1}) \bmod |\Sigma'|$ holds. In lines 9–11 the intervals of the pattern are stored in the table $L$ at the location corresponding to $(p_j - p_{j-1}) \bmod |\Sigma'|$, for $2 \leq j \leq m$. 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 $I[k]$ is the $k$ th bit. Thus, $I[k] \vee_b l_b = I[k]$ holds, and the condition in line 16 is met. Then the zero-preserving operation $\wedge_b$ is used to assign a zero to $T[l].j$.

Let us then assume that $T[l].j = 0$ holds. By inspecting line 16 we conclude that there is a $k$ such that $L[k].j = 0$ holds. According to the construction, $l.k$ must be zero and $k = (p_j - p_{j-1}) \bmod |\Sigma'|$ must hold.

□

Considering Algorithm 4 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 *potential occurrences* $H$ of $p$. Such an $H$ is an interval sequence of length $m - 1$ in $\overline{S}$, which contains the intervals of $p$ in the correct order. However, the condition that there is a $d$ such that $(p_i + d) \in S_{j+i-1}$ for each $i$ does not necessarily hold any longer (here $j$ is such that $H$ occurs at location $j + 1$ in $\overline{S}$). The following theorem shows that filtration works correctly, i.e. the algorithm does not skip any proper occurrence.

**Theorem 5** *Let $p$ be the pattern to be searched within the source $S$. If there is a transposition invariant occurrence starting at $S_i$, then Algorithm 4 finds a potential occurrence of $p$ starting at $\overline{S}_{i+1}$.*

**Proof.** The table $\overline{S}$ is correctly constructed according to Theorem 3. The scanning operation works in the same way as the shift-or algorithm in the case where each text position holds a set of characters. An interval in our setting corresponds to a character. $T[i]$ is got as a conjunction of the vectors $L[k]$ corresponding to the intervals belonging to $i$. The table $T$ is correctly constructed according to Lemma 4. The fact that each potential occurrence is identified follows from the characteristics of the shift-or algorithm.

□

Because Algorithm 4 is a filtering algorithm, its scanning phase can also find spurious occurrences (false positive hits). Let us consider an example: The scanning phase finds a spurious occurrence of a pattern $p = \mathtt{c}_2\mathtt{b}_1\mathtt{a}_1\mathtt{g}_1$ within the source given in Fig. 1.

**Complexity.** The space requirement of the Algorithm 4 is $O(n + c)$, where $c = 2^{|\Sigma'|} + 2|\Sigma'|$. The time complexity of the source preprocessing phase is $O(nq)$. At the beginning of the scanning phase the locations of intervals are gathered in time $O(m)$. After that, each interval mask is slid over the chord array, which takes $O(|\Sigma'| \cdot 2^{|\Sigma'|})$. Thus by denoting $d = |\Sigma'| \cdot 2^{|\Sigma'|}$, the scanning phase takes time $O(n + m + d)$, which is linear with respect to $n$ (recall that we assumed that $m \leq w$). Again, the worst case time complexity is that of the checking algorithm, because there might be a candidate at each position, in the worst case.

The algorithm is impractical if unlimited or even maximal MIDI intervals are allowed. So interval classification is a presupposition. For example, when applying the octave equivalence discussed earlier, the maximum number of different intervals between two consecutive chords, i.e. the size of the alphabet, is 12.

## 4 Experiments

In order to be able to evaluate the practical performance of Algorithm 4, we compared the performance of Algorithm 4 with the performance of Algorithm C when it checks all the source positions. In our experiments Algorithm C also has another role, because Algorithm 4 uses it as a subroutine for checking. We did not test Algorithm 3. However, we expect that the speed of that approach is somewhere between the two tested approaches.

We made an extensive study on altering the values of some interesting parameters, and observed how they affected the performance. Both the running times and the numbers of the candidates/proper occurrences were measured. The parameters we considered were $m, n, q$, and $|\Sigma'|$. We decided to measure the impact of the four parameters by letting only one parameter at a time vary, and using a fixed value for the other parameters. 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 of a machine word was 32 bits.

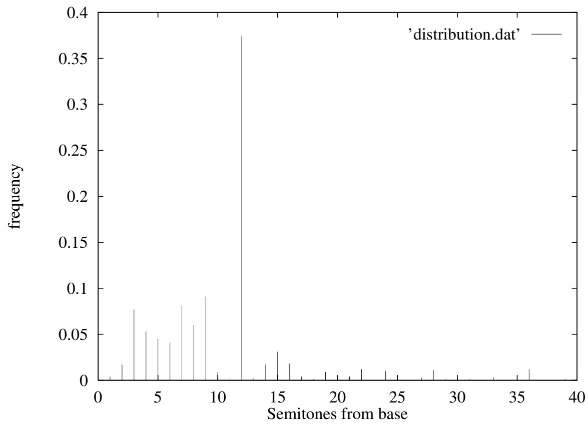


Figure 4: A distribution of intervals within chords in Sibelius' Finlandia.

The database for the experiments was collected from the Internet. It was comprised of 7,667 MIDI files; 6,190 files of them 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. Though we believe that this is a rather common phenomena (which makes Algorithm 4 more efficient due to the fewer false positive hits found), we wanted to bound the degree of polyphony. In order to do that, we first computed the distribution of intervals within chords (as semitones from the base) in a MIDI file of Jean Sibelius' Finlandia (see Fig. 4 for the distribution). Then, for each chord $S_i$, we inserted random pitches following the measured interval distribution of Finlandia until $|S_i|$ became equal to $q$. When we computed the distribution, we completely omitted MIDI tracks that contained, at the most, two distinct

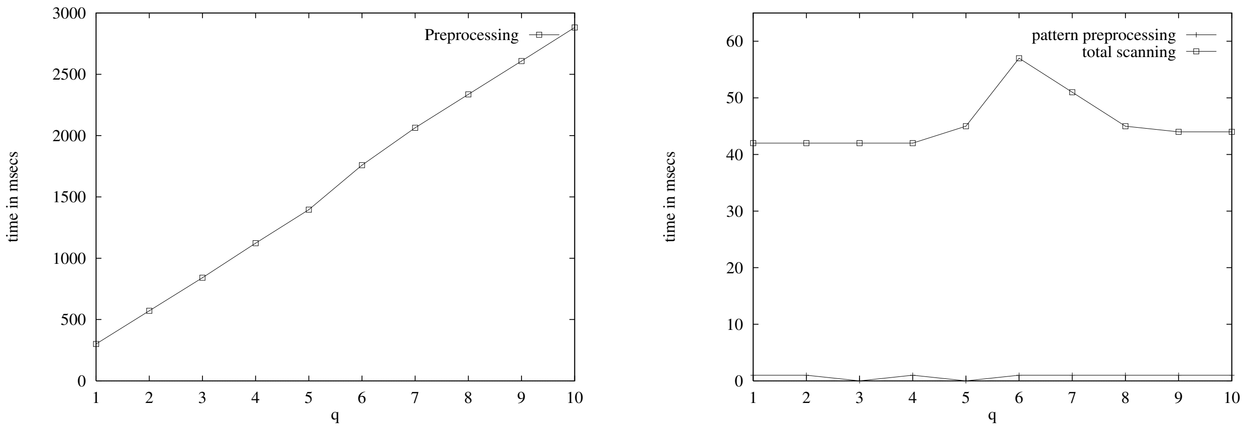


Figure 5: The performance of Algorithm 4 when varying $q$ with the setting: $m = 12; n = 1,484,940; |\Sigma'| = 12$. The source preprocessing time is given on the left, the pattern preprocessing and the total scanning time on the right.

pitches within chords, since those tracks probably contained information for percussion instruments.

The series of experiments was started by building up the source residing in the main memory. All the chords appearing in our MIDI database were concatenated into a single string, resulting in a source of the total length of 1,484,940 chords. Observing one parameter at a time, each setting (e.g. $q = 8$, other fixed to default values) was repeated 100 times. At the beginning of each repetition, a new pattern was randomly picked up from the source. Thus, it was guaranteed that 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 chosen to be as follows: $q = 3$ (more precisely, $|S| \equiv 3$); $m = 12$; $n = 1,484,940$; and $|\Sigma'| = 12$.

Fig. 5 illustrates the typical behaviour of Algorithm 4. In the two graphs, we have given the average times spent by the phases of the algorithm, when the parameter $q$ is varying with the assumption $|S| \equiv q$. Firstly, the source preprocessing time grows noticeably, when $q$ increases (see the graph on the left). In the graph on the right, we give the times spent by the pattern preprocessing (the lower curve) and by the total scanning phase (recall that the pattern preprocessing is a subphase of scanning).

Henceforth, we will consider two different running times for Algorithm 4. The first one represents the running time of a single (first) query (denoted by 'total time'); thus it includes the times spent by all the phases of Algorithm 4. The other one (denoted by 'scanning+checking') represents the running time of a re-query on the same database.

### 4.1 Varying the Degree of Polyphony

We started our comparison by measuring the effect of the parameter $q$, that is, the chord size. As discussed above, we ran this piece of experiment by letting $q$ vary with the condition $|S| \equiv q$. Naturally, in real cases the condition will hardly never be satisfied, but this is the only fair way to study the effect of this particular parameter.

Fig. 6 shows that the number of candidates grows much more rapidly than the number of proper occurrences, as $q$ increases (note the logarithmic scale). From around 350 at $q = 3$, the number of candidates grows to around 11,000 at $q = 4$. However, for Algorithm 4 the first query is faster than for the stand-alone Algorithm C, until $q$ becomes greater than 7; and re-queries with Algorithm 4 are clearly faster than with Algorithm C for $q < 9$.

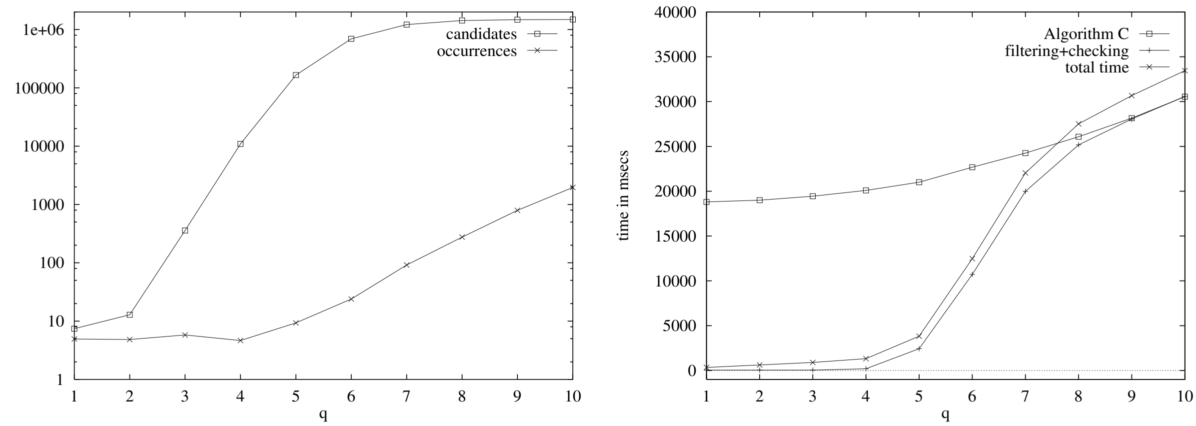


Figure 6: The average effect of $q$ ($m = 12; n = 1,484,940; |\Sigma'| = 12$). The numbers of candidates and proper occurrences (on the left, the log scale). The times for the first query (total time) and re-queries (filtering+checking) of Algorithm 4, and for stand-alone Algorithm C (on the right).

### 4.2 Varying the Length of the Pattern

Next we experimented on the influence of the length of the pattern (see Fig. 7). Recall that henceforth $|S| \equiv 3$ holds. As the pattern becomes longer, the number of occurrences decreases notably faster than the number of candidates. However, Algorithm 4 is considerably faster than Algorithm C with these parameter settings. The graph on the right illustrates two interesting phenomena. Firstly, the weak discriminating power of short patterns has a clear consequence to the performance of Algorithm 4; the shorter the pattern the more often the slow Algorithm C has to be executed. Secondly, the running time of Algorithm C does not depend on the pattern length. This is a well-known property of the naive matching algorithm (which was the basis of Algorithm C).

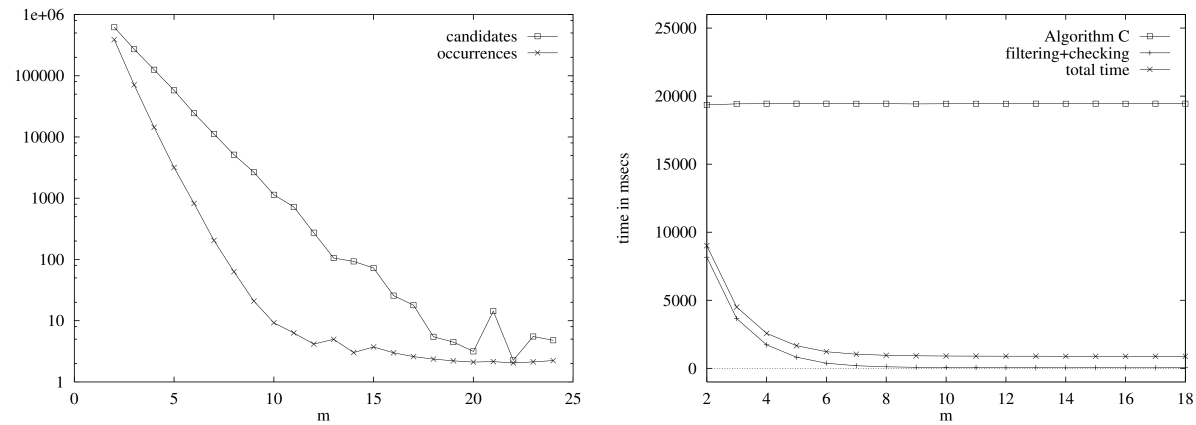


Figure 7: The average effect of $m$ ($q = 3; n = 1,484,940; |\Sigma'| = 12$). The numbers of candidates and proper occurrences (on the left, the log scale). The times for the first query (total time) and re-queries (filtering+checking) of Algorithm 4, and for stand-alone Algorithm C (on the right).

### 4.3 Varying the Length of the Source

Of all our experiments, the most significant difference was found when varying the size of the database (see Fig. 8). 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 Algorithm 4 takes much more time than the re-queries, it is much faster than the same query with Algorithm C. Because the running time of Algorithm C seems to increase linearly as the database grows (note the log scale), the longer the source is the larger the difference between the performances of the two approaches will be.

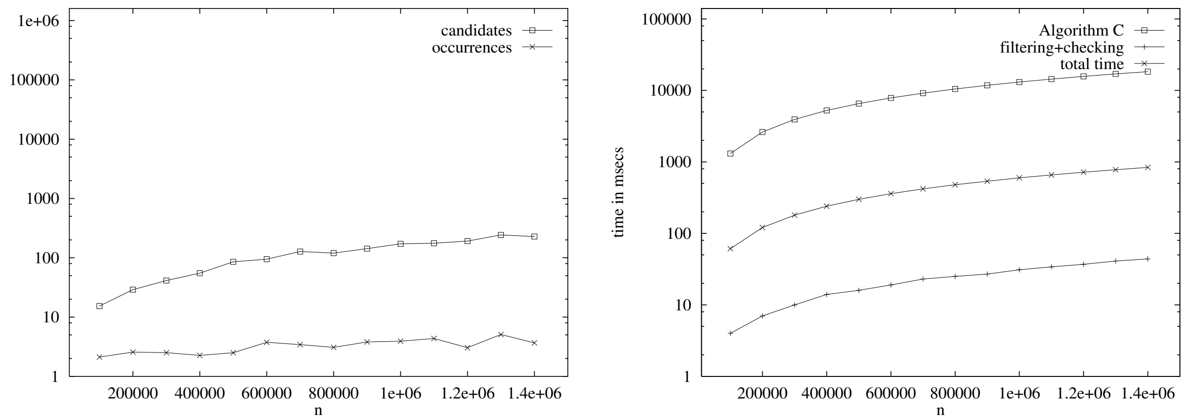


Figure 8: The average effect of $n$ ($q = 3; m = 12; |\Sigma'| = 12$). The numbers of candidates and proper occurrences (on the left, the log scale). The times for the first query (total time) and re-queries (filtering+checking) of Algorithm 4, and for stand-alone Algorithm C (on the right, the log scale).

### 4.4 Varying the Size of the Alphabet

Finally, we made experiments on the parameter $|\Sigma'|$. Recall that, so far, we have assumed octave equivalence with Algorithm 4, while Algorithm C uses the full MIDI alphabet. It can be seen in Fig. 9, that the octave equivalence with Algorithm 4 is a good presumption also in relation to efficiency; 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 $|\Sigma'| \geq 18$.[4] However, when $|\Sigma'|$ becomes greater than 20, the speed of Algorithm 4 starts to decrease due to the $O(|\Sigma'| \cdot 2^{|\Sigma'|})$ complexity of the pattern preprocessing phase (in a 600 MHz Pentium III the decreasing of the speed started at $|\Sigma'| = 18$ already). We did not experiment on cases when $|\Sigma'| > w$, because Algorithm C would have clearly outperformed Algorithm 4, due to the rapid speed decrease of Algorithm 4.

It would have been possible to define the whole matching problem to allow octave equivalence in the search results. In such a case, it is important to use the setting $|\Sigma'| = 12$ with the checking phases as well. Otherwise the algorithm does not necessarily find all possible occurrences of the pattern.

## 5 Concluding Remarks

We have adapted the shift-or algorithm to the context of music information retrieval, where good solutions for the problem of locating an exact monophonic pattern $p_1 \cdots p_m$ within a polyphonic source $S_1 \cdots S_n$ are needed. We have presented three modifications of the original algorithm for three variations of the problem. In Algorithm 2, the pattern has to be found exactly, and has to be presented by the same pitch sequence as it appears in the database (in other words, transposition invariance is not allowed). Letting $q$ denote the maximum of the chord sizes, the algorithm runs in time $O(nq)$ and space $O(m + \Sigma)$ when the pattern fits in a computer word ($m \leq w$).

[4] Naturally the curve of candidates depends on the interval distribution within the chords, but we believe that the distribution we have used is typical enough.

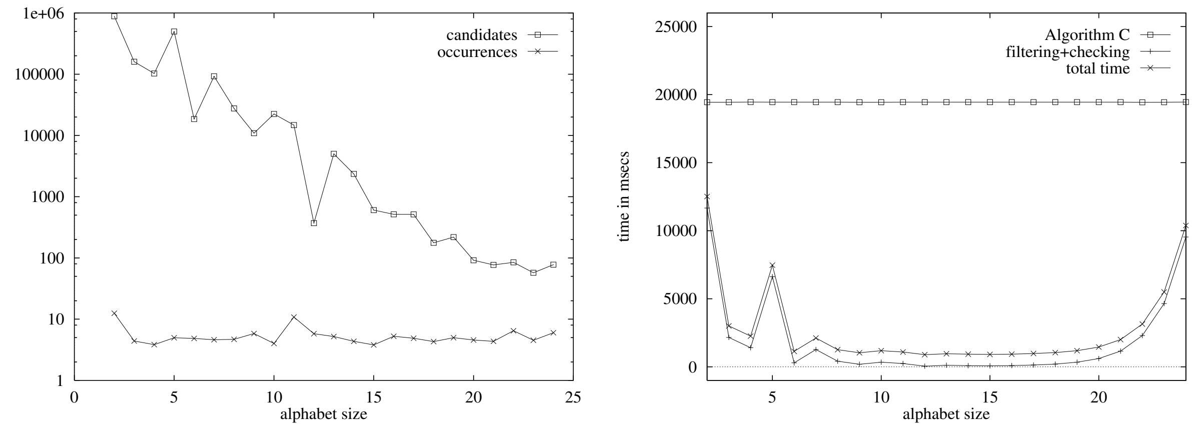


Figure 9: The average effect of $|\Sigma'|$ ($q = 3; m = 12; n = 1,484,940$). The numbers of candidates and proper occurrences (on the left, the log scale). The times for the first query (total time) and re-queries (filtering+checking) of Algorithm 4, and for stand-alone Algorithm C (on the right).

The two other variations of the problem deal with transposition invariance. As solutions to those problems we introduced two filtering algorithms. Algorithm 3 works on-line and its core runs in time $O(nq^2)$, with the total space complexity of $O(m + q^2)$. Algorithm 4 has been optimized to a case of static music database; the source is first preprocessed, which takes $O(nq)$ time. After that the scanning phase runs in time $O(n + m + d)$, where $d = |\Sigma'| \cdot 2^{|\Sigma'|}$ ($|\Sigma'|$ denotes the size of the reduced alphabet). Thus, the scanning phase runs in linear time (again assuming that $m \leq w$ holds). The overall space complexity is $O(n+c)$, where $c = 2^{|\Sigma'|} + 2|\Sigma'|$. Moreover, we introduced an $O(nq(q+m))$ time checking Algorithm (referred to Algorithm C), which is capable of locating all the proper occurrences of a given monophonic pattern within a polyphonic source. Furthermore, it can be used to check whether a candidate starting at a given location is a proper occurrence or not.

We made extensive experiments on Algorithm 4 by studying the effect of the parameters $m, n, |\Sigma'|$, and $q$ to its performance. Due to our experiments the octave equivalence is not only a musically meaningful concept; it also assigns a good setting for the size of the alphabet to our Algorithm 4, with respect to a competent performance. Even though constraining the size of the underlying alphabet mostly affects the performance of the source preprocessing, the whole algorithm becomes impractical if unlimited or even maximal MIDI intervals are allowed (recall that the size of the alphabet defines the size of the bit-vectors $\overline{S}$). It is also interesting to observe the consequence of varying the value of $q$. Though it does not have a direct consequence to the speed of the scanning phase, it affects the efficiency of the filtration; the larger the $q$ the more false positive hits.

When comparing the performance of Algorithm 3 (using Algorithm C in the checking phase) with that of stand-alone Algorithm C, according to our experiments, our algorithm clearly outperforms Algorithm C, if the average size of chords is reasonably low.

There are several possibilities to make the algorithms faster. First of all, the octave invariance assumption could have been used in Algorithm 3 as well. Secondly, the method that we used in Algorithm 4 in order to compute $\overline{S}_i$ in time $O(q)$, can also be applied in Algorithm 3 to compute set $D$.

Secondly, the scanning phase of Algorithm 4 can be made more effective by using a crossing (Navarro & Raffinot 1998) of the shift-or algorithm and the Boyer-Moore approach. Although the algorithm has a worst-case complexity of $O(nm)$, it is faster than the shift-or algorithm in practice. As with all Boyer-Moore type algorithms, their BNDM algorithm becomes faster, when the pattern gets longer. According to the experiments of Navarro and Raffinot, BNDM is up to 7 times faster than the shift-or algorithm, when $m = 32$.

Moreover, the distribution of the intervals may be used for locating only the less frequent subsequences of the pattern with the filtering algorithm. 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. Naturally, it would be possible to use a similar trick with longer patterns, i.e. for the cases $m > w$: In such a case one should use a $\leq w$ length subsequence of $p$ having the sum of interval probabilities as low as possible (in other words, highest possible entropy), in the first pass.

Polyphonic patterns could also be found by using a similar heuristic to that of above. A monophonic pattern (having the highest entropy) is first extracted out of the polyphonic pattern and then passed to the filtering algorithm. Finally, the candidates are checked, e.g., by an algorithm similar to Algorithm C.

## Acknowledgements

The authors are indebted to Esko Ukkonen for useful and critical remarks that improved the manuscript. We thank Mika Klemettinen for his comments on the manuscript, and Sami Perttu for letting us use his MIDI-file processing-routines. The work is supported by the Finnish Cultural Foundation and the Academy of Finland.

## References

Aho, A. V. & Corasick, M. J. (1975). Efficient string matching. *Communications of the ACM* **18**(6), 333–340.

Baeza-Yates, R. & Gonnet, G. H. (1992). Fast text searching allowing errors. *Communications of the ACM* **35**(10), 74–82.

Blackburn, S. & DeRoure, D. (1998). A tool for content based navigation of music. In *ACM Multimedia 98 Proceedings* (pp. 361–368). Bristol.

Boyer, R. & Moore, S. (1977). A fast string searching algorithm. *Communications of the ACM* **20**(10), 762–772.

Crawford, T., Iliopoulos, C. S. & Raman, R. (1998). String matching techniques for musical similarity and melodic recognition. *Computing in Musicology* **11**, 71–100.

Crochemore, M. & Rytter, W. (1994). *Text Algorithms*. Oxford University Press.

Dovey, M. J. (1999). An algorithm for locating polyphonic phrases within a polyphonic musical piece. In *Proceedings of the AISB'99 Symposium on Musical Creativity* (pp. 48–53). Edinburgh.

Downie, J. S. (1999*a*). Evaluating a Simple Approach to Music Information Retrieval: Conceiving Melodic n-grams as Text. PhD thesis. The University of Western Ontario, Faculty of Information and Media Studies.

Downie, J. S. (1999*b*). Music retrieval as text retrieval: simple yet effective. In *Proceedings of the 22nd International Conference on Research and Development in Information Retrieval* (pp. 297–298). Berkeley, CA.

Forte, A. (1962). *Tonal Harmony in Concept and Practice*. New York: Holt, Rinehardt and Winston.

Ghias, A., Logan, J., Chamberlin, D. & Smith, B. C. (1995). Query by humming - musical information retrieval in an audio database. In *ACM Multimedia 95 Proceedings* (pp. 231–236). San Francisco, CA.

Holub, J., Iliopoulos, C. S., Melichar, B. & Mouchard, L. (1999). Distributed string matching using finite automata. In *Proceedings of the 10th Australasian Workshop On Combinatorial Algorithms* (pp. 114–128). Perth, WA, Australia.

Knuth, D., Morris, J. & Pratt, V. (1977). Fast pattern matching in strings. *SIAM Journal of Computing* (6), 323–350.

Lemström, K. & Laine, P. (1998). Musical information retrieval using musical parameters. In *Proceedings of the 1998 International Computer Music Conference* (pp. 341–348). Ann Arbor, MI.

McNab, R., Smith, L., Bainbridge, D. & Witten, I. (1997). The New Zealand digital library MELody inDEX. *D-Lib Magazine*.

Navarro, G. & Raffinot, M. (1998). A bit-parallel approach to suffix automata: Fast extended string matching. In *Proceedings of Combinatorial Pattern Matching* (pp. 14–33). Piscataway, NJ.

Parncutt, R. (1989). *Harmony: A Psychoacoustical Approach*. Springer-Verlag.

Pinker, S. (1997). *How the mind works*. New York: W. W. Norton & Company.

Tseng, Y.-H. (1999). Content-based retrieval for music collections. In *Proceedings of the 22nd International Conference on Research and Development in Information Retrieval* (pp. 176–182). Berkeley, CA.

Uitdenbogerd, A. L. & Zobel, J. (1998). Manipulation of music for melody matching. In *ACM Multimedia 98 Proceedings* (pp. 235–240). Bristol.

Wu, S. & Manber, U. (1992). Fast text searching allowing errors. *Communications of the ACM* **35**(10), 83–91.