Nordic Journal of Computing

DISTRIBUTED STRING MATCHING WITH TRANSLATION INVARIANCE

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 *distributed string matching*. The task is to find whether a pattern has occurrences that are distributed across parallel strings. Given a set S of h strings $s^i = s^i_1 \cdots s^i_n, i \in \{1, \ldots, h\}$ for some constant h, a *pattern* $p = p_1 \cdots p_m$ has a distributed occurrence at position j of S if $p_1 = s^{i_1}_j, p_2 = s^{i_2}_{j+1}, \ldots, p_m = s^{i_m}_{j+m-1}$ for some $\{i_1, \ldots, i_m\} \in \{1, \ldots, h\}$. A real application for the problem is music retrieval where occurrences of a monophonic query pattern is searched in a polyphonic music database. In music retrieval it is even more pertinent to search for certain kind of translation invariant distributed occurrences of the pattern. This invariance allows pitch level transpositions, thus called *transposition invariance*. We present a straightforward algorithm, called DirectCheck, solving the problems. It works in time $O(nhm)$. Moreover, we introduce fast bit-parallel algorithms for the considered problems. Our main contribution, the MonoPoly algorithm, is a transposition-invariant filtering-algorithm for static databases. It finds candidates for transposition invariant distributed occurrences in time $O(n\lceil \frac{m}{w} \rceil + m)$ where w denotes the size of the computer words in bits. DirectCheck is incrementally used to check whether the candidates are proper occurrences. In a typical situation, the number of candidates is rather small, and therefore the combined method is essentially faster than mere DirectCheck. When several subsequent searches are done with the same database, the MonoPoly approach is superior to DirectCheck.

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 (see, e.g., [Crochemore and Rytter 1994],[Gusfield 1997]) is a fundamental problem in many application areas, such as in information retrieval. The most conventional form of the problem is to find whether there are *exact occurrences* of a given query *pattern* $p = p_1 \cdots p_m$ within a

Received December 19, 2001.

text $s = s_1 \cdots s_n$, where each p_i $(1 \leq i \leq m)$ and s_j $(1 \leq j \leq n)$ is from an underlying alphabet Σ (p_i or s_j may equal to an empty character ϵ). 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 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 $O(n\lceil m/w \rceil)$, where w is the size of the computer word (e.g. 32 or 64 bits, in practice). Thus, if $m \leq w$ holds, the algorithm runs in linear time.

In this paper, we consider novel extensions of the exact matching problem and present several algorithms solving them. In *distributed string matching* the source S is composed of h parallel voice strings, $s^i = s^i_1 \cdots s^i_n, i \in \{1, \ldots, h\}$, and p is said to have a distributed occurrence across the voices h at j, if $p_1 = s^{i_1}_j, p_2 = s^{i_2}_{j+1}, \ldots, p_m = s^{i_m}_{j+m-1}$ for some $\{i_1, \ldots, i_m\} \in \{1, \ldots, h\}$. Please, note our distinction between s and S corresponding to a *plain* string and a *compound* string, respectively. The name of the problem was proposed by Crawford *et al.* [1998].

Distributed 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 by using strings where each element represent, for instance, at a rudimentary level, the *pitch* information of a note. In terms of our specification, music is said to be *monophonic* if $h = 1$, while if $h > 1$ it is called *polyphonic*, which is the common case in current western music. A special case of polyphonic music, which is called *homophonic music*, is when none of s^i_js equals to the empty character ϵ. A typical case for a music retrieval query is when a monophonic pattern (that may be given, e.g, by humming, by playing an instrument, or just by typing) is searched for in a polyphonic music database.

There is an underlying translation invariance 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. A method considering such intervals instead of the absolute values is said to be *transposition invariant*. Apparently, transposition invariance is a natural and useful property of an MIR algorithm. To this end, we update the formulation of our problem as follows: given the pattern p and the source S comprising h voices, $s^i = s^i_1 \cdots s^i_n, i \in \{1, \ldots, h\}$, the task is to find all the 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 $\{i_1, \ldots, i_m\} \in \{1, \ldots, h\}$. We call this *transposition invariant distributed string matching*.

We show how the ShiftOr algorithm can be straightforwardly modified to be applicable for distributed string matching. We call this modification, which works in time $O(nh\lceil m/w \rceil)$, ShiftOrAnd. Then for transposition invariant distributed string matching, we first 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 transposition invariant distributed occurrences can be found more efficiently in practice, by executing a filtering algorithm before DirectCheck (or some other algorithm capable of checking). First we introduce the MP filter working *on-line* in time $O(nh^2)$. Then we consider a case where the database under consideration is *static*, i.e., it is not updated between consequent queries. For such case we introduce the main contribution of this paper, the MonoPoly filtering algorithm. It consists of an $O(nh)$ preprocessing and an $O(n\lceil m/w \rceil + m)$ filtering phase.

The paper is organized as follows. In Section 2 we present some background for our study: recall some basic string combinatorics, show how music can be represented by using strings, and review some related work. Section 3 reviews the ShiftOr algorithm and shows how it can be modified to distributed string matching (the ShiftOrAnd algorithm). In Section 4, we will introduce the three novel algorithms for transposition invariant distributed string matching. Having first described DirectCheck and the MP algorithm, we will devote more time in a detailed description and careful analyses on our main contribution, i.e., the MonoPoly algorithm. Before concluding the paper in Section 6, we will show the results of our extensive experiments on MonoPoly in Section 5.

2. Background

2.1 String Combinatorics

Let Σ be a finite set of symbols, an *alphabet*. Then any $x = x_1, x_2, \ldots, x_m$ where each x_i is a symbol in Σ, is a *string* over Σ. Usually we write $x = x_1 \cdots x_m$. The *length* of x is $|x| = m$. The string of length 0 is called the *empty string* and denoted λ. The set of strings of length i over Σ is denoted by Σ^i, and the set of all strings over Σ by Σ^*. If a string x is of form $x = \alpha\beta\gamma$, where $\alpha, \beta, \gamma \in \Sigma^*$, α is a *prefix*, β a *factor* (substring) of x, and γ a *suffix* of x. A string x' is a *subsequence* of x if it can be obtained from x by deleting zero or more symbols, i.e., $x' = x_{i_1} \cdots x_{i_m}$, where $i_1, \ldots, i_m$ is an increasing sequence of indices in x.

2.2 Representing Music

In this paper, we use a rudimentary representation of polyphonic music showing the pitch (or interval) and voice information, and which preserves the original ordering of the pitches. As the underlying alphabet, we start with MIDI pitch values [MIDI Manufacturers Association 1996]; $\Sigma_{midi} = \{0, \ldots, 127\} \bigcup \{\epsilon\}$, 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 $s^3 = 72, \epsilon, \epsilon, 72$ (recall, that $\epsilon \in \Sigma$ represents an empty character).

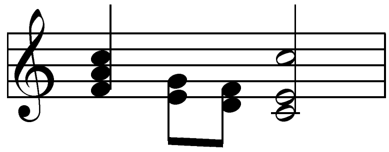

Fig. 2.1: A musical excerpt.

It should be noted that when moving from absolute values to intervals, the size of the underlying alphabet is doubled (the interval alphabet corresponding to an absolute alphabets Σ is denoted by Σ'). Because an alphabet of size $|\Sigma'_{midi}|$ would make our MonoPoly algorithm impractical, we will use a smaller but musically relevant alphabet.

From the musical point of view, an alphabet distinguishing 12 pitches only would still be relevant. By musical terms, two pitches which are separated by 12 semitones form an interval called *octave*. Among all the intervals, octave is very special, for combinations of them are *consonant*, and that is not the case for any other interval [Parncutt 1989]. *Octave equivalence*, which is "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. 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 $s^3 = 0, \epsilon, \epsilon, 0$.

By S_j we denote an ordered vertical section of the source at j, i.e., $S_j = s_j^1, s_j^2, \ldots, s_j^h$ where $s_j^{i+1} \geq s_j^i$ for $1 \leq i \leq h-1$. We call such vertical section a *chord*. The chords can be represented by bitvectors $\overline{\mathtt{S}}[j]$. Each chord bitvector (cbv) is of size of the underlying cbv alphabet. A zero in a cbv corresponds to a present pitch in the corresponding chord, while an on-bit indicates absence of the corresponding pitch. For instance, if the underlying cbv alphabet is Σ_{12} (absolute pitches reduced according to octave equivalence), the cbv string $\overline{\mathtt{S}} = \langle \overline{\mathtt{S}}[1] \rangle \cdots \langle \overline{\mathtt{S}}[n] \rangle$ corresponding to Fig. 2.1 would be:

$$\begin{gathered}
\langle 0,1,1,1,1,0,1,1,1,0,1,1 \rangle, \\
\langle 1,1,1,1,0,1,1,0,1,1,1,1 \rangle, \\
\langle 1,1,0,1,1,0,1,1,1,1,1,1 \rangle, \\
\langle 0,1,1,1,0,1,1,1,1,1,1,1 \rangle.
\end{gathered}$$

By $\overline{\mathtt{S}}[j].i$ we denote the ith bit of the cbv $\overline{\mathtt{S}}[j]$. For instance, in the example above, $\overline{\mathtt{S}}[1].0 = \overline{\mathtt{S}}[2].4 = \overline{\mathtt{S}}[3].5 = 0$.

Being musically relevant, the remaining question to be answered is whether the octave alphabet is discriminative enough for efficient matching and retrieval. Downie [1999, p. 78] measured the frequency of the intervals occurring in a database of 9,354 monophonic folksongs. In his measurement, the vast majority of the intervals occurred between the scale $-5, \ldots, 5$ of semitones. Moreover, the distribution of the intervals had approximately the shape of a normal distribution, centered around the zero interval. Hence in the monophonic case, when applying octave equivalence the distribution of the intervals should behave very nicely for our needs; 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.

In Section 5, we will show that the alphabet Σ'_{12} is practical and effective in the polyphonic case, as well.

2.3 Related Work

Independently of us, Holub *et al.* [1999] presented bit-parallel algorithms for distributed string matching (a preliminary version of this paper appeared in Lemström and Tarhio [2000]). They did not, however, consider transposition invariance. They presented algorithms to find (i) multi-patterns within monophonic sources (effectively, the original ShiftOr algorithm); (ii) monophonic patterns within polyphonic sources (effectively same as our ShiftOrAnd algorithm; and (iii) multi-patterns within polyphonic sources. The algorithm (iii) requires $O((rm + |\overline{\Sigma}|)\lceil \frac{m}{w} \rceil)$ time for the preprocessing, where r, $\overline{\Sigma}$, and w are the number of the patterns, the set of symbols used in the pattern and the size of the computer word, respectively. Then it works in $O(nh\lceil \frac{m}{w} \rceil)$ time and requires $O(|\overline{\Sigma}|\lceil \frac{m}{w} \rceil)$ space. In their paper, they also discussed about approximate distributed matching.

Dovey has considered a modification of the distributed 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: $p_1 = s_{j_1}^{i_1}, p_2 = s_{j_2}^{i_2}, \ldots, p_m = s_{j_m}^{i_m}$ (for some $\{i_1, \ldots, i_m\} \in \{1, \ldots, h\}$), where $j_{l+1} - j_l \leq t + 1$ for $1 \leq l \leq m - 1$. He represents music by chord vectors over cbv alphabet Σ_{88} (88 is the common amount of keys in a piano). First he has described a recursive algorithm of $O(nm\,(t + 1)^{(m-1)})$ worst case time complexity [Dovey 1999]. The algorithm requires 87 reiterations for transposition invariant matching. The algorithm have been refined in [Dovey 2001] to $O(nm)$ complexity by using 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 $1 \le j \le n$ and $1 \le i \le m$. As usually, the query result is read from the row d_{mj}; an occurrence of value $t+1$ in the bottom row indicates an t-gap-occurrence, and the actual occurrence can be uncovered by a backtracking procedure. Again, reiterations are needed for transposition invariance.

The SIA(M)ESE algorithm by Meredith *et al.* [2001] is capable, e.g., of transposition invariant distributed string matching with unlimited gaps. It is not, however, a string matching algorithm. It works with any n-dimensional datasets; in our case the pattern and the source are formed as pairs (q, t), where q and t represents 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 source points:

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

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

Denoting by n', m', and $\mathcal{D}$ the number of events in the source, the number of events in the pattern, and the number of their dimensions, respectively, SIA(M)ESE runs in time $O(n'm' \log(n'm'))$ and space $O(n'm'\mathcal{D})$ [Wiggins *et al.* 2001] (note that in homophonic music $m' = hm$ and $n' = hn$). Therefore, in general, when compared to our MONOPOLY, SIA(M)ESE will not find the exact transposition invariant distributed occurrences as quickly.

3. SHIFTOR Algorithm

Let us now recall the SHIFTOR algorithm by Baeza-Yates and Gonnet [1992]. In describing their algorithm (and henceforth) we will use the symbols $\vee_b$ and $\wedge_b$ representing the bitwise `or` and `and` operators, respectively. The SHIFTOR algorithm searching occurrences of p in s is as follows:

```
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) ∨_b T[s_j]
6      if E.m = 0 then Write(j)
```

Fig. 3.1: The SHIFTOR algorithm.

Lines 1–2 and 3–6 of SHIFTOR (Fig. 3.1) 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

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

Fig. 3.2: An example of ShiftOr working in a case where $p = aab$ and $s = abcaaab$. The table T (to the left) is created first. The execution of the core phase of the algorithm is illustrated to the right. In the illustration, shifting is done downwards and the found occurrence is showed by a white circle.

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 $\vee_b$ 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 the working of ShiftOr on 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 $O(\lceil \frac{m}{w} \rceil m + \lceil \frac{m}{w} \rceil |\Sigma|)$.

3.1 ShiftOrAnd — *Algorithm for Distributed Matching*

The ShiftOr algorithm can be adapted with a minor modification to distributed string matching. Actually, this is a dual of the string matching problem presented by Baeza-Yates and Gonnet [1992], where 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 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; see the ShiftOrAnd algorithm in Fig. 3.3, below (as mentioned earlier, recently Holub *et al.* [1999] also reported this idea).

The main loop in lines 4–6 takes $O(n\lceil \frac{m}{w} \rceil h)$ time, 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 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 Distributed Matching

The problem of distributed string matching becomes trickier when taking into account transposition invariance. In this section we present two fast

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

Fig. 3.3: The ShiftOrAnd algorithm for distributed string matching.

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 $\overline{\texttt{S}}$, 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.

We consider only the alphabet Σ_{midi}. Because each chord S_j holds at most h pitches, the time complexity of forming $\overline{\texttt{S}}$ is $T_1 = O(nh) + O(n\lceil|\Sigma_{midi}|/w\rceil)$, 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 $O(m\lceil|\Sigma_{midi}|/w\rceil)$. 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 $nh \gg \lceil|\Sigma_{midi}|/w\rceil$ holds in practice, T_2 dominates over T_1. Therefore we consider $O(nhm)$ as the total time of DirectCheck in the following. The

DIRECTCHECK(S, p, n, m, Σ)
1 COMPUTE$\overline{\mathtt{S}}(S)$
2 **for** $j \leftarrow 1$ **to** $n - m + 1$ **do**
3   **for each** $a \in S_j$ **do**
4   $found \leftarrow$ **true**; $b \leftarrow a$
5   **for** $i \leftarrow 2$ **to** m **do**
6       $x \leftarrow b + p_i - p_{i-1}$
7   **if** $((0 \leq x < |\Sigma|)$ **and** $(\overline{\mathtt{S}}[j+i-1].(x \bmod |\Sigma|) = 0))$
8   **then** $b \leftarrow x$
9   **else** $found \leftarrow$ **false**; EXIT()
10   **if** $found$
11   **then** PRINT(occurrence at $S_j \cdots S_{j+m-1}$); EXIT()

Fig. 4.1: DIRECTCHECK for transposition invariant distributed string matching.

space requirement of the algorithm is $O(m\lceil|\Sigma_{midi}|/w\rceil)$ with the circular buffer and $O(n\lceil|\Sigma_{midi}|/w\rceil)$ without it.

Note that in the version given above, a factor $S_j \cdots S_{j+m-1}$ 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.

4.2 MP — *On-Line Filter*

Let us now introduce a basic on-line filter for transposition invariant distributed matching. In the MP algorithm, we use the MIDI interval alphabet, $\Sigma'_{midi} = \{-127, \ldots, 127\}$. $\mathtt{T}[d]$ represents a bit-mask corresponding to the interval d.

By comparing SHIFTORAND (Fig. 3.3) with MP (Fig. 4.2), one can notice two differences. Firstly, in MP 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 SHIFTORAND. The other difference can be noticed on line 8: MP 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, MP 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

$\mathrm{MP}(S, p, n, m, \Sigma'_{midi})$

for each $d \in \Sigma'_{midi}$ **do** $\mathtt{T}[d] \leftarrow 2^{m-1} - 1$
for $i \leftarrow 2$ **to** m **do** $\mathtt{T}[p_i - p_{i-1}] \leftarrow \mathtt{T}[p_i - p_{i-1}] - 2^{i-2}$
$\mathtt{E} \leftarrow 2^{m-1} - 1$
for $j \leftarrow 2$ **to** n **do**
  $D \leftarrow \emptyset$
  for each $a \in S_{j-1}$ and $b \in S_j$ **do** $D \leftarrow D \bigcup \{b - a\}$
  $\mathtt{E} \leftarrow \mathtt{shiftleft(E)} \vee_b (\bigwedge_b(\{\mathtt{T}[d] \mid d \in D\}))$
  if $\mathtt{E}.m = 0$ **then** CHECK(j)

Fig. 4.2: The MP on-line filter for transposition invariant distributed string matching.

required extra space is $h^2 + 1$; h^2 is needed for the set D, and one computer word for the bitwise `and` operation (as in SHIFTORAND). Thus, the total space requirement of MP is $O(\lceil \frac{m}{w} \rceil m + \lceil \frac{m}{w} \rceil |\Sigma| + h^2)$.

4.3 MONOPOLY — *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 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 working of MONOPOLY is divided into (source) *preprocessing*, and *filtering* phases. The filtering phase is further divided into *pattern processing*, and *core* subphases. The preprocessing is necessary only before the first database query. The two subphases are in correspondence with the phases of the SHIFTOR algorithm.

4.3.1 Preprocessing Phase

This phase 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 S_j and S_{j+1}. The subphase avoids the apparent $O(nh^2)$ time requirement by using bitwise operations and a fact

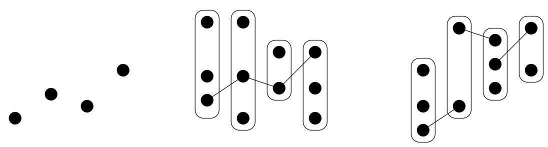

Fig. 4.3: The query pattern, given to the left, has a proper occurrence in the first chord string (the corresponding elements are bound), but only a spurious occurrence in the second (the corresponding elements are not bound). Both are considered as candidates.

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. To this end, let us represent chords S_j, for $2 \leq j \leq n$, by bit-vectors $\varphi[j]$ of $|\Sigma'|$ bits:

$$\varphi[j+1].i = \begin{cases} 0, & \text{if there is an element } x \text{ in } S_{j+1}, \text{ such that} \\ & \quad ((x - \mathtt{B}(S_j)) \bmod |\Sigma'|) = i (1 \leq j \leq n-1); \\ 1, & \text{otherwise,} \end{cases}$$

where $\mathtt{B}(S_j)$ denotes the lowest pitch value, the *bass*, of the chord S_j (recall that the pitches within a chord are ordered, so we always have a direct access to any $\mathtt{B}(S_j)$). Now, a bit-vector $\overline{\mathtt{S}}[j]$ is established as follows:

$$\overline{\mathtt{S}}'[j].i = \begin{cases} 0, & \text{if } \varphi[j+1].i = 0, \\ 0, & \text{if } \varphi[j+1].((i+l) \bmod |\Sigma'|) = 0, \text{ where } l = y - \mathtt{B}(S_j) \\ & \quad \text{and } y \text{ is an element in } S_j, \text{ such that } y \neq \mathtt{B}(S_j) \\ 1, & \text{otherwise.} \end{cases} \tag{4.1}$$

The intuition above is that $\varphi[j+1]$ gives the intervals between the elements of S_{j+1} and the bass of the chord S_j. By shifting these intervals, they are then copied to represent the remaining elements of S_j, as well. Equation (4.1) can be implemented efficiently by using a *right circularshift* bitwise operator, denoted $\mathtt{rcs}(\mathtt{a}, b)$, which shifts a bit-vector $\mathtt{a}$ by b bits to the right in a circular manner. For instance, if $\mathtt{a} = 01010$ then $\mathtt{rcs}(\mathtt{a}, 1) = 00101$ and $\mathtt{rcs}(\mathtt{a}, 2) = 10010$. Thus, each copying can be done in constant time. 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 $O(nh\lceil \frac{m}{w} \rceil)$ time (recall that w denotes the size of the computer word). The algorithmic sketch of the preprocessing is given in Fig. 4.4.

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

Fig. 4.4: The preprocessing phase of MonoPoly.

Once the source, representing a musical database, has been preprocessed and the string $\overline{\mathtt{S}}'[1] \cdots \overline{\mathtt{S}}'[n-1]$ is available, an arbitrary number of queries to that string can be performed by executing the filtering phase, described next.

4.3.2 Filtering Phase

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

To achieve linear time complexity, two extra arrays are used while building T'. A bit-array I of $|\Sigma'| \times |\Sigma'|$ bits has a column for every possible interval in the reduced alphabet, 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 $1 \leq i \leq m-1$; $1 \leq j, k \leq |\Sigma'|$)

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

Note that every $\overline{\mathtt{S}}'[j]$ can be viewed as an integer l (and vice versa), whose value is restricted to the range $0 \leq l \leq 2^{|\Sigma'|} - 1$. These values are used as indices to the table T'. Moreover, we use bit-vectors $\mathtt{I}[j]$ to locate intervals within $\overline{\mathtt{S}}'[j]s$, 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, building the array T' takes time $O(|\Sigma'| \cdot 2^{|\Sigma'|})$.

The core subphase 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.

4.4 Correctness and Analysis of MonoPoly

We start considering the correctness of MonoPoly by proving the correctness of the preprocessing phase, hence proving that $\overline{\mathtt{S}}'$ is correctly formed. We divide the proof in two subcases considered by the following two lemmas. In the following, the interval between the bass of the chord S_{j-1} and an element y within the chord S_j is denoted by $r(y)$, where $r(y) = y - \mathtt{B}(S_{j-1})$. 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 $x \in S_{j-1}$ and $y \in S_j$ for some j, $2 \leq j \leq n$. 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) $x = \mathtt{B}(S_{j-1})$ and (ii) $x \neq \mathtt{B}(S_{j-1})$.

(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 ($\wedge_b$ preserves zeros), $\overline{\mathtt{S}}'[j].i = 0$ holds.

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 $\quad\quad \mathtt{ivect} \leftarrow \mathtt{I}[k]$
12 $\quad\quad$ **if** $\mathtt{ivect} \vee_b l = \mathtt{ivect}$ **then** $\mathtt{T}'[l] \leftarrow (\mathtt{T}'[l] \ \wedge_b \ \mathtt{L}[k])$
13 $\mathtt{E} \leftarrow 2^{m-1} - 1$
14 **for** $j \leftarrow 1$ **to** $n-1$ **do**
15 $\quad \mathtt{E} \leftarrow \mathtt{shiftleft(E)} \vee_b \mathtt{T}'[\overline{\mathtt{S}}'[j]]$
16 $\quad$ **if** $\mathtt{E}.m = 0$ **then** Check(j)

Fig. 4.5: The MonoPoly filter for transposition invariant distributed string matching.

(ii) According to case (i), $\overline{\mathtt{S}}'[j]$ holds all the intervals between $\mathtt{B}(S_{j-1})$ and S_j before line 6 is executed. Let y be an arbitrary pitch within S_j, $d = y - \mathtt{B}(S_{j-1})$, and $e = y - x$. Now the difference of e and d is $\mathtt{B}(S_{j-1}) - x$, 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 in line 6 by the $\mathtt{and}$ 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 $x \in S_{j-1}$ and $y \in S_j$, 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 $y - \mathtt{B}(S_{j-1})$ for each $y \in S_j$. Let A be the value of $\overline{\mathtt{S}}'[j]$ after the loop. In line 5, first $|S_{j-1}| - 1$ shifted copies out of A are formed, then they are combined with A by using the $\wedge_b$ operation (line 6). Each copy holds all the intervals between S_j and some x that differs from $\mathtt{B}(S_{j-1})$. Clearly the algorithm does not assign superfluous zeros to $\overline{\mathtt{S}}'[j]$. □

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

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

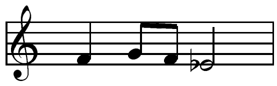

Fig. 4.6: This excerpt has a spurious occurrence in Fig. 2.1.

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 $\mathtt{L}$ 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 $\mathtt{L}$, 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, $\mathtt{I}[k] \vee_b l_b = \mathtt{I}[k]$ holds, and the condition in line 12 is met. Then the zero-preserving operation $\wedge_b$ 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. 4.6 represents the pattern and Fig. 2.1 the source. 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 source S. If there is a transposition invariant distributed 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. □

4.4.1 Complexity of MonoPoly

MonoPoly's space requirement is $O(n + c)$, where $c = 2^{|\Sigma'|} + 2|\Sigma'|$. The time complexity of the preprocessing phase is $O(nh)$. At the beginning of the filtering phase the locations of intervals are gathered in time $O(m)$. After that, each interval mask `I` is slid over the values l_b, which takes $O(|\Sigma'| \cdot 2^{|\Sigma'|})$. Therefore, by denoting $d = |\Sigma'| \cdot 2^{|\Sigma'|}$, the filtering takes time $O(n\lceil \frac{m}{w} \rceil + m + d)$, which is linear 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'_{midi}, is used. The octave equivalence assumption, for instance, keeps the table `T`' reasonably sized, and thus, MonoPoly practical.

4.5 Possibility for 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 source 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) is 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. We did not test MP (Fig. 4.2). However, it may be expected that the performance of MP lies somewhere between the two tested approaches.

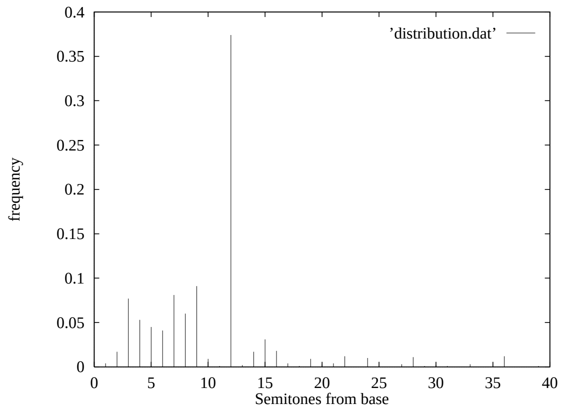

Fig. 5.1: A distribution of intervals within chords in Sibelius' Finlandia.

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 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 source 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 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: $h = 3$; $m = 12$; $n = 1,484,940$; and $|\Sigma'| = 12$.

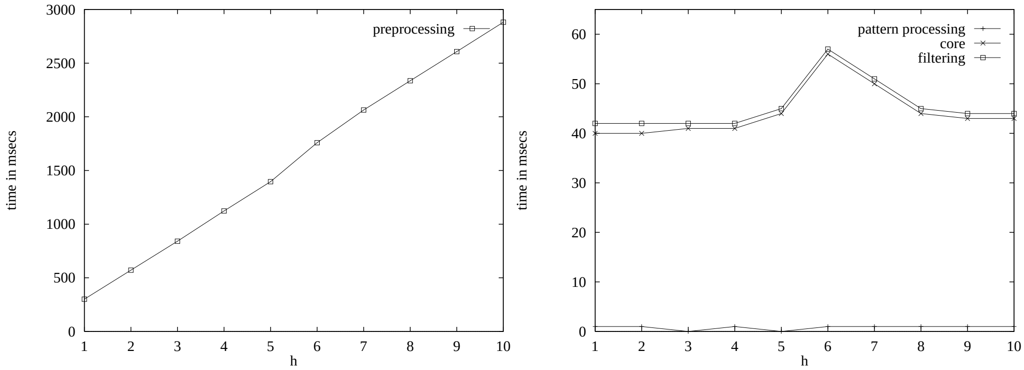

Fig. 5.2: The performance of MonoPoly when varying h ($m = 12; n = 1,484,940; |\Sigma'| = 12$). The preprocessing time is given in the graph to the left, pattern processing, core, and filtering (i.e., pattern processing + core) times in the graph to the right.

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 to the left). In the graph to 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 source preprocessing).

5.1 Varying the Number of Voices

We started our comparison by measuring the effect of the parameter h, that is, the number of voices in the source. 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 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 $h < 9$.

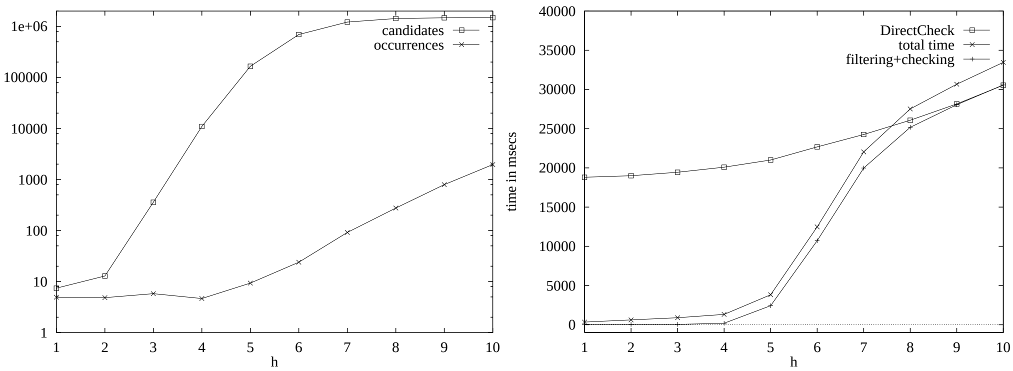

Fig. 5.3: The average effect of h ($m = 12; n = 1,484,940; |\Sigma'| = 12$). The numbers of candidates and proper occurrences (to the left, log scale). The times for a first query (total time) and for re-queries (filtering+checking) of MonoPoly and for DirectCheck (to the right).

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, the running time of DirectCheck does not depend on the pattern length. This is a characteristic property of the naïve matching algorithm (which is the basis of DirectCheck).

5.3 Varying the Length of the Source

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 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.

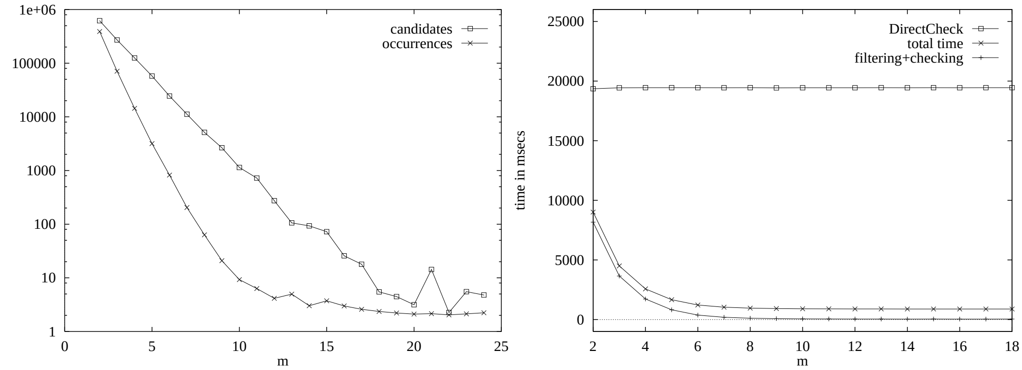

Fig. 5.4: The average effect of m ($h = 3; n = 1,484,940; |\Sigma'| = 12$). The numbers of candidates and proper occurrences (to the left, log scale). The times for a first query (total time) and re-queries (filtering+checking) of MonoPoly and for DirectCheck (to the right).

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'_{midi}. 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

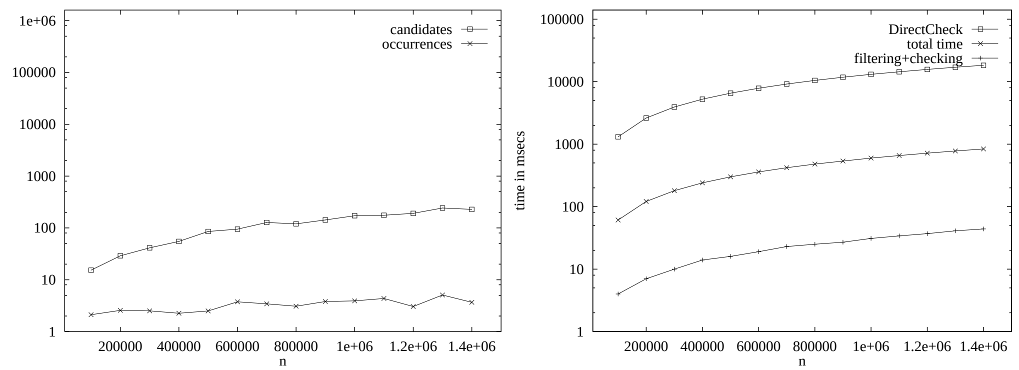

Fig. 5.5: The average effect of n ($h = 3; m = 12; |\Sigma'| = 12$). The numbers of candidates and proper occurrences (to the left, log scale). The times for a first query (total time) and re-queries (filtering+checking) of MonoPoly and for DirectCheck (to the right, log scale).

pattern processing phase (in a 600 MHz Pentium III the decreasing of the speed started at $|\Sigma'| = 18$, already).

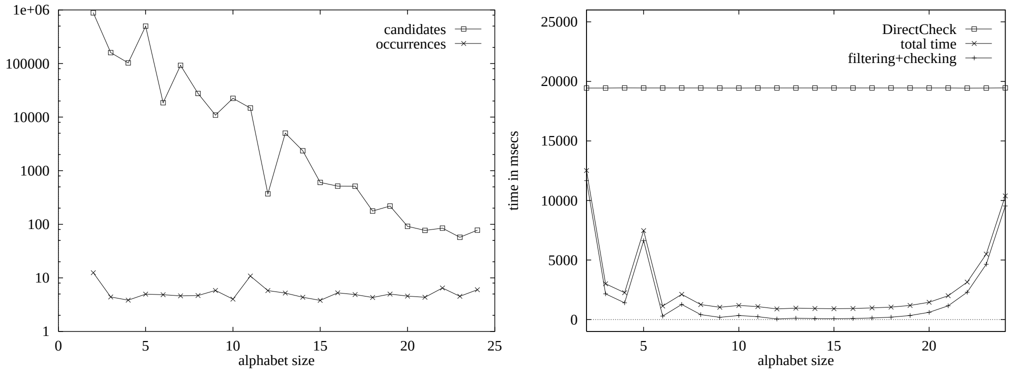

Fig. 5.6: The average effect of $|\Sigma'|$ $(h = 3; m = 12; n = 1,484,940)$. The numbers of candidates and proper occurrences (to the left, log scale). The times for a first query (total time) and re-queries (filtering+checking) of MonoPoly, and for DirectCheck (to the right).

6. Concluding Remarks

In this paper, we have adapted the ShiftOr algorithm to the context of music retrieval, where good solutions for the distributed string matching problem are needed. We have presented three modifications of ShiftOr for two distinct variations of the problem. We suggested the ShiftOrAnd algorithm (Fig. 3.3) for the original distributed matching problem: given a pattern $p_1 \cdots p_m$ and a set S (source) of h strings, $s^i = s^i_1 \cdots s^i_n, i \in \{1,\ldots,h\}$ for some constant h, the task is to find whether p has an occurrence in S, such that $p_1 = s^{i_1}_j, p_2 = s^{i_2}_{j+1}, \ldots, p_m = s^{i_m}_{j+m-1}$ for some j and $\{i_1,\ldots,i_m\} \in \{1,\ldots,h\}$. ShiftOrAnd runs in time $O(n\lceil \frac{m}{w} \rceil h)$ and space $O(\lceil \frac{m}{w} \rceil m + \lceil \frac{m}{w} \rceil |\Sigma|)$, where w and $|\Sigma|$ denote the sizes of the computer word in bits and of the underlying alphabet, respectively.

The two other algorithms are for transposition invariant distributed string matching, where an occurrence of p is found if $p_1 = s^{i_1}_j + c, p_2 = s^{i_2}_{j+1} + c, \ldots, p_m = s^{i_m}_{j+m-1} + c$, for some constant c, and $\{i_1,\ldots,i_m\} \in \{1,\ldots,h\}$. First, we introduced an $O(nhm)$ solution, called DirectCheck. Then we introduced two fast filtering algorithms based on ShiftOr. Both of the filters uses a modification of DirectCheck for checking whether the found candidates are proper occurrences. The MP filter works on-line and it runs in time $O(n\lceil \frac{m}{w} \rceil h^2)$, with the total space complexity of $O(\lceil \frac{m}{w} \rceil m + \lceil \frac{m}{w} \rceil |\Sigma| + h^2)$. The other, MonoPoly, has been optimized to work with static music databases. The source is first preprocessed, which takes $O(nh)$ time. After that the filtering phase runs in time $O(n\lceil \frac{m}{w} \rceil + m + d)$, where $d = |\Sigma'| \cdot 2^{|\Sigma'|}$ ($|\Sigma'|$ denotes the size of a reduced alphabet). Thus, the filter runs in linear time, when $m \leq w$. The overall space complexity is $O(n+c)$, where $c = 2^{|\Sigma'|} + 2|\Sigma'|$.

We made extensive experiments with MonoPoly on studying the effect of the 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. Even though it does not have a direct consequence to 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, our main contribution, the MonoPoly algorithm, clearly outperforms the straightforward DirectCheck whenever h is reasonably low.

There are several possibilities to refine our algorithms. For MP (Fig. 4.2), 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 $O(nh)$. For MonoPoly, as suggested in Subsection 4.5, the core can be replaced by the BNDM algorithm of Navarro and Raffinot [1998].

In both of the filtering algorithms, a further, practical improvement for the performance may be obtained by observing the distribution of the symbols (intervals) and by searching only for the least frequent factors 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 factors (of length m') of p for which $m' \leq w$ and whose sum of the interval probabilities is the lowest. Naturally, the trick may be used with polyphonic patterns, as well.

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.

Crawford, T., Iliopoulos, C.S., and Raman, R. 1998. String Matching Techniques for Musical Similarity and Melodic Recognition. *Computing in Musicology 11*, 71–100.

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

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.

Downie, J.S. 1999. *Evaluating a Simple Approach to Music Information Retrieval: Con-*

ceiving Melodic n-grams as Text. PhD thesis, The University of Western Ontario, Faculty of Information and Media Studies.

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.

GUSFIELD, D. 1997. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press.

HOLUB, J., ILIOPOULOS, C.S., MELICHAR, B., AND MOUCHARD, L. 1999. Distributed string matching using finite automata. In *Proceedings of the 10th Australasian Workshop On Combinatorial Algorithms*. Perth, 114–128.

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.

MEREDITH, D., WIGGINS, G.A., AND LEMSTRÖM, K. 2001. Pattern induction and matching in polyphonic music and other multi-dimensional datasets. In *the 5th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI'2001)*, Volume X. Orlando, FLO, 61–66.

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. 2001. SIA(M)ESE: An efficient algorithm for transposition invariant pattern matching in multidimensional datasets. (prep.)