Geometric Algorithms for Transposition Invariant Content-Based Music Retrieval *

Esko Ukkonen, Kjell Lemström, and Veli Mäkinen
Department of Computer Science, University of Helsinki
P.O. Box 26 (Teollisuuskatu 23), FIN-00014 Helsinki, Finland
{ukkonen,klemstro,vmakinen}@cs.helsinki.fi

Abstract

We represent music as sets of points or sets of horizontal line segments in the Euclidean plane. Via this geometric representation we cast transposition invariant content-based music retrieval problems as ones of matching sets of points or sets of horizontal line segments in plane under translations. For finding the exact occurrences of a point set (the query pattern) of size m within another point set (representing the database) of size n, we give an algorithm with running time $O(mn)$, and for finding partial occurrences another algorithm with running time $O(mn \log m)$. We also use the total length of the overlap between the line segments of a translated query and a database (i.e., the shared time) as a quality measure of an occurrence and present an $O(n \log n + mn \log m)$ algorithm for finding translations giving the largest possible overlap. Some experimental results on the performance of the algorithms are reported.

1 Introduction

The methods introduced for content-based music retrieval on symbolically encoded music have largely been applications of conventional approximate string matching techniques based on the edit distance (Mongeau and Sankoff, 1990; Ghias et al., 1995; McNab et al., 1997; Lemström, 2000) or discrete time-warping (Zhu and Shasha, 2003). The techniques work nicely with monophonic music, as such music can be represented with linear sequences of discrete pitch levels. By using intervals between successive pitches instead of absolute pitch levels, matching becomes transposition invariant. Similarly, one achieves an invariance on tempi by considering the duration ratios between successive notes.

Problems get more complicated, however, when dealing with polyphonic music like symphony orchestra scores. Such a music may have very complex structure with several notes simultaneously on and several musical themes developing in parallel. One might want to find similarities or other interesting patterns in it, for example, in order to make musicological comparative analysis of the style of different composers or even for copyright management purposes. Formulating various music-psychological phenomena and models such that one can work with them using combinatorial algorithms becomes a major challenge.

Quite recently, Meredith et al. (2001) suggested a geometric, piano-roll-like music representation and content-based music retrieval algorithms working on such representation. We will use here an extended representation that includes the durations, as well. Using this representation, the excerpt of Fig. 1 given in the common music notation, is represented geometrically in Fig. 2. The content should be evident: each horizontal bar represents a note, its location in the y-axis gives its pitch level and the start and end points in the x-axis give the time interval when the note is on. The piano-roll representation as such is an old invention; in music retrieval research it has been used earlier e.g. by Dovey (1999).

We will use this simple two-dimensional geometric representation of music. In this representation, a piece of music is a collection of horizontal line segments (at times simply points) in the Euclidean two-dimensional space $\mathbb{R}^2$; the horizontal axis refers to the time, the vertical to the pitch values.

Given two such representations, P and T, we want to find the common patterns shared by P and T when P is translated with respect to T. Translating P means that some $f \in \mathbb{R}^2$ is added to all elements of P. Obviously, the vertical component of the translation yields transposition invariance of the pattern matching while the horizontal component means shifting in time. When designing the algorithms we assume that T represents a large music database while P is a shorter query pattern.

Three problems will be considered.

($P1$) Find translations of P such that all starting points of the line segments of P match with some starting points of the line segments in T. Hence the on-set times of all notes of P must match. We also consider a variant in which the note durations must match, too, or in which the time segment of T covered by the translated P is not allowed to contain any extra (i.e., unmatched) notes.

($P2$) Find all translations of P that give a partial match of the

*A work supported by the Academy of Finland (grant 201560).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. ©2003 Johns Hopkins University.

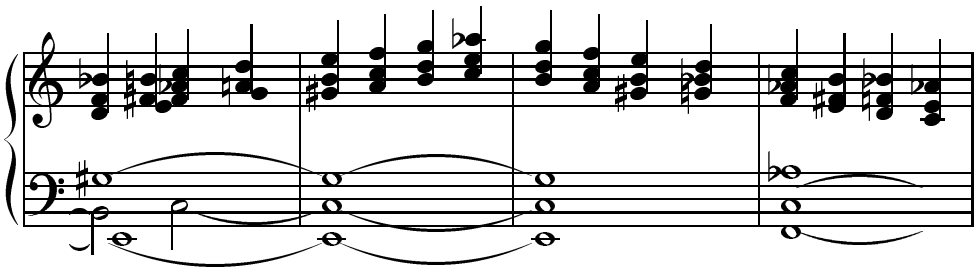

Figure 1: An excerpt of polyphonic music from Einojuhani Rautavaara's opera *Thomas* (1985). Printed with the permission of the publisher Warner/Chappell Music Finland Oy.

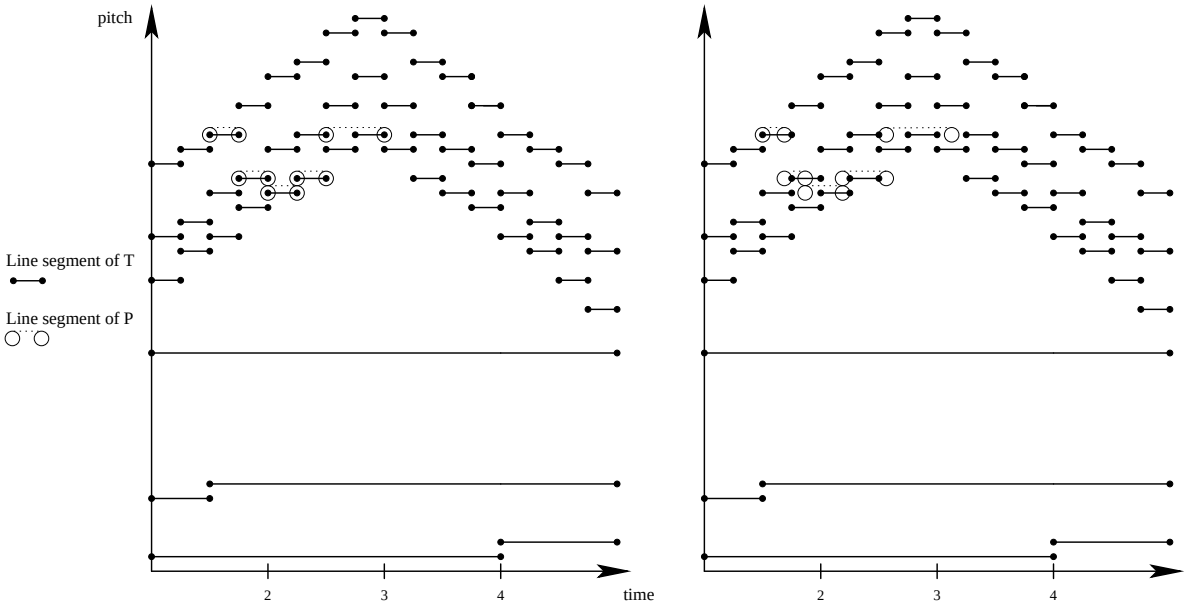

Figure 2: The example of Fig. 1 is given in piano-roll representation. Consider a query pattern $P = a\flat, f, e, f, a\flat$ (given in pitch class symbols) where the first 4 events are quarter notes and the last a half note. In the left panel, the location of P does not give a solution to $(P1)$ since the fifth note onset does not match: the reader may check that no other translation of P will give any better match. However, for problems $(P2)$ and $(P3)$ the left panel gives a quite good solution: 4 of the 5 onset times match (problem $(P2)$), and the shared time is 5 quarter notes. If the query is slightly out of time (the right panel) $(P2)$ becomes useless while $(P3)$ still founds a satisfactory match.

on-set times of P with those of T. In a partial match, only a subset of the on-set times of P should match.

$(P3)$ Find translations of P that give longest common shared time with T, i.e., the longest total length of the line segments that are obtained as intersection of the line segments of T and the translated P.

Fig. 2 illustrates all three problems. There is no solution of $(P1)$ in either of the examples, a good solution to $(P2)$ and $(P3)$ in the example on the left, and a satisfactory result to $(P3)$ in the example on the right.

Our problems are basic content-based music retrieval questions when transpositions are allowed and note additions and deletions are modeled as partial matches. $(P1)$ and $(P2)$ work well with ideal P and T where the start points of the notes are exact. However, this is not the case neither with a query by humming pattern, nor with a database created by playing a MIDI instrument. Problem $(P3)$ is of a novel type we have not seen to be considered before. It allows local changes in note start points and durations, thus providing needed tolerance for such cases, see Fig. 2. Moreover, it is not deteriorated anyway by note fragmentations or consolidations (Mongeau and Sankoff, 1990). Tolerance to interval changes could be incorporated by representing the notes as narrow rectangles instead of line segments.

For the problem $(P1)$ we give in Sect. 3 an algorithm that needs time $O(mn)$ and working space $O(m)$. In practice the average running time is $O(n)$. Here m is the size (number of the line segments) of P and n is the size of T. For the problem $(P2)$ we give in Sect. 4 an algorithm with running time $O(mn \log m)$ and space $O(m)$, and for the problem $(P3)$ we describe in Sect. 5 a method that needs time $O(n \log n + mn \log m)$ and space $O(m + n)$. We assume that T and P are given in the lexicographic order of the starting points of the line segments. Otherwise extra $O(n \log n + m \log m)$ time and $O(n+m)$ space is needed for sorting. All algorithms are based on a sweepline type scanning of T, a technique widely used in computational geometry (Bentley and Ottmann, 1979). In Sect. 6 we demon-

strate by empirical evaluation that all our algorithms are fast enough to be useful in practice.

A preliminary report of the algorithms in this paper appeared as (Ukkonen et al., 2003).

1.1 Related work

In this paper, we improve the results by Wiggins et al. (2003) (see also (Meredith et al., 2001)). Our solution for $(P1)$, being closely related to their SIA(M)EX algorithm, is presented here for completeness, and making it easier to understand the solutions for $(P2)$ and $(P3)$. For $(P2)$ we have improved their time bound (from $O(mn\log(mn))$ to $O(mn\log m)$) and, more noteworthy, their space requirement from an excessive $O(mn)$ bound to a practical $O(m)$ bound. To our knowledge $(P3)$ has not been studied in the literature.

The majority of the literature on symbolic content-based music retrieval algorithms deals only with monophonic music. The polyphonic music retrieval algorithms based on the edit distance framework consider only the order of the notes out of the rhythmic information. Examples of such studies are by Dovey (2001), who uses dynamic programming for finding polyphonic patterns with restricted gaps, by Lemström and Tarhio (2003), who apply bit–parallelism for finding transposed monophonic patterns within polyphonic music, and by Holub et al. (2001), who use bit–parallelism for finding approximate occurrences of patterns with group symbols. A recent bit-parallel algorithm by Lemström and Navarro (2003) is capable of dealing with gaps in polyphonic music.

The PROMS system (Clausen et al., 2000) is another example of polyphonic content-based music retrieval systems that accounts for note-on information (in addition to pitch information).

2 Line segment patterns

A *line segment pattern* in the Euclidean space $\mathbb{R}^2$ is any finite collection of horizontal line segments. Such a segment is given as $[s, s']$ where the *starting point* $s = (s_x, s_y) \in \mathbb{R}^2$ and the *end point* $s' = (s'_x, s'_y) \in \mathbb{R}^2$ of the segment are such that $s_y = s'_y$ and $s_x \leq s'_x$. The segment consists of the points between its end points. Two segments of the same pattern may overlap.

We will consider different ways to match line segment patterns. To this end we are given two line segment patterns, P and T. Let P consist of m segments $[p_1, p'_1], \ldots, [p_m, p'_m]$ and T of n segments $[t_1, t'_1], \ldots, [t_n, t'_n]$. Pattern T may represent a large database while P is a relatively short query to the database in which case $m \ll n$. It is also possible that P and T are about of the same size, or even that they are the same pattern.

We assume that P and T are given in the lexicographic order of the starting points of their segments. The lexicographic order of points $a = (a_x, a_y)$ and $b = (b_x, b_y)$ in $\mathbb{R}^2$ is defined by setting $a < b$ iff $a_x < b_x$, or $a_x = b_x$ and $a_y < b_y$. When representing music, the lexicographic order corresponds the standard reading of the notes from left to right and from the lowest pitch to the highest.

So we assume that the lexicographic order of the starting points is $p_1 \leq p_2 \leq \cdots \leq p_m$ and $t_1 \leq t_2 \leq \cdots \leq t_n$. If this is not true, a preprocessing phase is needed, to sort the points which would take additional time $O(m\log m + n\log n)$.

A *translation* in the real plane is given by any $f \in \mathbb{R}^2$. The translated P, denoted by $P + f$, is obtained by replacing any line segment $[p_i, p'_i]$ of P by $[p_i + f, p'_i + f]$. Hence $P + f$ is also a line segment pattern, and any point $v \in \mathbb{R}^2$ that belongs to some segment of P is mapped in the translation as $v \mapsto v + f$.

3 Exact matching

Let us denote the lexicographically sorted sets of the starting points of the segments in our patterns P and T as $\overline{P} = (p_1, p_2, \ldots, p_m)$ and $\overline{T} = (t_1, t_2, \ldots, t_n)$. We now want to find all translations f such that $\overline{P} + f \subseteq \overline{T}$. Such a $\overline{P} + f$ is called an *occurrence* of $\overline{P}$ in $\overline{T}$. As all points of $\overline{P} + f$ must match some point of $\overline{T}$, $p_1 + f$ in particular must equal some t_j. Hence there are only n potential translations f that could give an occurrence, namely the translations $t_j - p_1$ where $1 \leq j \leq n$.

To check for some translation $f = t_j - p_1$, whether also the remaining points $p_2 + f, \ldots, p_m + f$ of $\overline{P} + f$ match, we utilize the lexicographic order. The method will be based on the following simple lemma.

Denote the potential translations as $f_j = t_j - p_1$ for $1 \leq j \leq n$. Let $p \in \overline{P}$, and let f_j and $f_{j'}$ be two potential translations such that $p + f_j = t$ and $p + f_{j'} = t'$ for some $t, t' \in \overline{T}$. That is, when p_1 matches t_j then p matches t, and when p_1 matches $t_{j'}$ then p matches t'.

Lemma 1 *If $j < j'$ then $t < t'$.*

Proof. If $j < j'$, then $t_j < t_{j'}$ by our construction. Hence also $f_j < f_{j'}$, and the lemma follows. □

Our algorithm makes a traversal over $\overline{T}$, matching p_1 against the elements of $\overline{T}$. At element t_j we in effect are considering the translation f_j. Simultaneously we maintain for each other point p_i of $\overline{P}$ a pointer q_i that also traverses through $\overline{T}$. When q_i is at t_j, it in effect represents translation $t_j - p_i$. This is compared to the current f_j, and the pointer q_i is updated to the next element of $\overline{T}$ after q_i if the translation is smaller (the step $q_i \leftarrow next(q_i)$ in the algorithm below). If it is equal, we have found a match for p_i, and we continue with updating q_{i+1}. It follows from Lemma 1, that no backtracking of the pointers is needed.

The resulting Algorithm 1 is given below.

ALGORITHM 1.

```
(1)  for i ← 1,...,m do q_i ← −∞
(2)  q_{m+1} ← ∞
(3)  for j ← 1,...,n − m do
(4)      f ← t_j − p_1
(5)      i ← 1
(6)      do
(7)          i ← i + 1
(8)          q_i ← max(q_i, t_j)
(9)          while q_i < p_i + f do q_i ← next(q_i)
(10)     until q_i > p_i + f
(11)     if i = m + 1 then output(f)
(12) end for.
```

Note that the main loop (line 3) of the algorithm can be stopped when $j = n-m$, i.e., when p_1 is matched against t_{n-m}. Matching p_1 beyond t_{n-m} would not lead to a full occurrence of $\overline{P}$ because then all points of $\overline{P}$ should match beyond t_{n-m}, but there are not enough points left.

That Algorithm 1 correctly finds all f such that $\overline{P} + f \subseteq \overline{T}$ is easily proved by induction. The running time is $O(mn)$, which immediately follows from that each q_i traverses through $\overline{T}$ (possibly with jumps!). Also note that this bound is achieved only in the rare case that $\overline{P}$ has $\Theta(n)$ full occurrences in $\overline{T}$. More plausible is that for random $\overline{P}$ and $\overline{T}$, most of the potential occurrences checked by the algorithm are almost empty. This means that the loop 6–10 is executed only a small number of times at each check point, independently of m. Then the expected running time under reasonable probabilistic models would be $O(n)$. In this respect Algorithm 1 behaves analogously to the brute-force string matching algorithm.

It is also easy to use additional constraints in Algorithm 1. For example, one might want that the lengths of the line segments must also match. This can be tested separately once a full match of the starting points has been found. Another natural requirement can be, in particular if P and T represent music, that there should be no extra points in the time window covered by an occurrence of $\overline{P}$ in $\overline{T}$. If $\overline{P} + f$ is an occurrence, then this time window contains all members of $\overline{T}$ whose x-coordinate belongs to the interval $[(p_1+f)_x, (p_m+f)_x]$. When an occurrence $\overline{P}+f$ has been found in Algorithm 1, the corresponding time window is easy to check for extra points. Let namely $t_j = p_1 + f$ and $t_{j'} = p_m + f$. Then the window contains just the points of $\overline{T}$ that match $\overline{P} + f$ if and only if $j' - j = m - 1$ and t_{j-1} and $t_{j'+1}$ do not belong to the window.

4 Largest common subset

Our next problem is to find translations f such that $\left(\overline{P} + f\right) \cap \overline{T}$ is nonempty. Such a $\overline{P} + f$ is called a *partial occurrence* of $\overline{P}$ in T. In particular, we want to find f such that $\left(\overline{P} + f\right) \cap \overline{T}$ is largest possible.

There are $O(mn)$ translations f such that $\left(\overline{P} + f\right) \cap \overline{T}$ is nonempty, namely the translations $t_j - p_i$ for $1 \le j \le n$, $1 \le i \le m$. Checking the size of $\left(\overline{P} + f\right) \cap \overline{T}$ for each of them solves the problem. A brute-force algorithm would typically need time $O(m^2 n \log n)$ for this. We will give a simple algorithm that will do this during m simultaneous scans over $\overline{T}$ in time $O(mn \log m)$.

Lemma 2 *The size of $\left(\overline{P} + f\right) \cap \overline{T}$ equals the number of disjoint pairs (j, i) (i.e., pairs sharing no elements) such that $f = t_j - p_i$.*

Proof. Immediate. □

By Lemma 2, to find the size of any non-empty $\left(\overline{P} + f\right) \cap \overline{T}$ it suffices to count the multiplicities of the translation vectors $f_{ji} = t_j - p_i$. This can be done fast by first sorting them and then counting. However, we can avoid full sorting by observing that translations $f_{1i}, f_{2i}, \ldots, f_{ni}$ are in the lexicographic order for any fixed i. This sorted sequence of translations can be generated in a traversal over $\overline{T}$. By m simultaneous traversals we get these sorted sequences for all $1 \le i \le m$. Merging them on-the-fly into the sorted order, and counting the multiplicities solves our problem.

The detailed implementation is very standard. As in Algorithm 1, we let $q_1, q_2, \ldots, q_m$ refer to the entries of $\overline{T}$. Initially each of q_i refers to t_1, and it is also convenient to set $t_{n+1} \leftarrow \infty$. The translations $f_i = q_i - p_i$ are kept in a priority queue F. Operation $min(F)$ gives the lexicographically smallest of translations f_i, $1 \le i \le m$. Operation $update(F)$ deletes the minimum element from F, let it be $f_h = q_h - p_h$, updates $q_h \leftarrow next(q_h)$, and finally inserts the new $f_h = q_h - p_h$ into F.

Then the body of the pattern matching algorithm is as given below.

```
(1) f ← −∞; c ← 0;
    do
(2)     f' ← min(F); update(F)
(3)     if f' = f then c ← c + 1
(4)     else {output(f, c); f ← f'; c ← 1}
(5) until f = ∞
```

The algorithm reports all (f, c) such that $\left|\left(\overline{P} + f\right) \cap \overline{T}\right| = c$ where $c > 0$.

The running time of the algorithm is $O(mn \log m)$. The m-fold traversal of $\overline{T}$ takes mn steps, and the operations on the m element priority queue F take time $O(\log m)$ at each step of the traversal.

The above method finds all partial occurrences of $\overline{P}$ independently of their size. Concentrating on large partial occurrences gives possibilities for faster practical algorithms based on *filtration*. We now sketch such a method. Let $\left|\left(\overline{P} + f\right) \cap \overline{T}\right| = c$ and $k = m - c$. Then $\overline{P} + f$ is called an occurrence with k *mismatches*.

We want to find all $\overline{P} + f$ that have at most k mismatches for some fixed value k. Then we partition $\overline{P}$ into $k + 1$ disjoint subsets $\overline{P}_1, \ldots, \overline{P}_{k+1}$ of (about) equal size. The following simple fact which has been observed earlier in different variants in string matching literature will give our filter.

Lemma 3 *If $\overline{P} + f$ is an occurrence of $\overline{P}$ with at most k mismatches then $\overline{P}_h + f$ must be an occurrence of $\overline{P}_h$ with no mismatches at least for one h, $1 \le h \le k + 1$,*

Proof. By contradiction: If every $\overline{P}_h + f$ has at least 1 mismatch then $\overline{P} + f$ must have at least $k + 1$ mismatches as $\overline{P} + f$ is a union of disjoint line segment patterns $\overline{P}_h + f$. □

This gives the following filtration procedure: First (the filtration phase) find by Algorithm 1 of Section 3 all (exact) occurrences $\overline{P}_h + f$ of each P_h. Then (the checking phase) find for each f produced by the first phase, in the ascending order of the translations f, how many mismatches each $\overline{P} + f$ has.

The filtration phase takes time $O(mn)$, sorting the translations takes $O(r \log k)$ where $r \le (k + 1)n$ is the number of translations the filter finds, and checking using an algorithm similar to the algorithm given previously in this section (but now priority queue is not needed) takes time $O(m(n + r))$. It should again be obvious, that the expected performance can be much better whenever k is relatively small as compared to m. Then the filtration would take expected time $O(kn)$. This would dominate

the total running time for small r if the checking is implemented carefully.

5 Longest common time

Let us denote the line segments of P and T by $\pi_i = [p_i, p_i']$ and $\tau_j = [t_j, t_j']$, respectively.

Our problem in this section is to find a translation f such that the line segments of $P + f$ intersects T as much as possible. For any horizontal line segments L and M, let $c(L, M)$ denote the length of their intersection line segment $L \cap M$. Then let

$$C(f) \quad = \quad \sum_{i,j} c(\pi_i + f, \tau_j).$$

Our problem is to maximize this function. $C(f)$ is nonzero only if the vertical component f_y of $f = (f_x, f_y)$ brings some π_i to the same vertical position as some τ_j, i.e., only if $f_y = (t_j)_y - (p_i)_y$ for some i, j.

Let H be the set of different values $(t_j)_y - (p_i)_y$ for $1 \le i \le m$, $1 \le j \le n$. Note that H is a standard set, not a multiset; the size of H is $O(mn)$.

As $C(f)$ gets maximum when $f_y \in H$ we obtain that

$$\max_f C(f) \quad = \quad \max_{f_x} \{C((f_x, f_y)) \mid f_y \in H\}. \qquad (1)$$

We will now explicitly construct the function $C((f_x, f_y)) = C_{f_y}(f_x)$ for all fixed $f_y \in H$. To this end, assume that $f_y = (t_j)_y - (p_i)_y$ and consider the value of $c_{ij}(f_x) = c(\pi_i + (f_x, f_y), \tau_j)$. This is the contribution of the intersection of $\pi_i + (f_x, f_y)$ and τ_j to the value of $C_{f_y}(f_x)$. The following elementary analysis characterizes the behaviour of $c_{ij}(f_x)$ when f_x grows, i.e., when $\pi_i + (f_x, f_y)$ slides from left to right and its intersection with τ_j is first empty, then starts growing, then stays the same until starts shrinking and finally gets again empty. When f_x is small enough, $c_{ij}(f_x)$ equals 0. When f_x grows, at some point the end point of $\pi_i + (f_x, f_y)$ meets the starting point of τ_i and then $c_{ij}(f_x)$ starts to grow linearly with slope 1 until the starting points and the end points of $\pi_i + (f_x, f_y)$ and τ_j meet, whichever comes first. After that, $c_{ij}(f_x)$ has a constant value (minimum of the lengths of the two line segments) until the starting points or the end points (i.e., the remaining pair of the two alternatives) meet, from which point on, $c_{ij}(f_x)$ decreases linearly with slope -1 until it becomes zero at the point where the starting point of π_i hits the end point of τ_j. An easy exercise shows that the only turning points of $c_{ij}(f_x)$ are the four points described above and their values are

$$\begin{array}{ll}
f_x = (t_j)_x - (p_i')_x & \text{slope 1 starts} \\
f_x = \min\left((t_j)_x - (p_i)_x, (t_j')_x - (p_i')_x\right) & \text{slope 0 starts} \\
f_x = \max\left((t_j)_x - (p_i)_x, (t_j')_x - (p_i')_x\right) & \text{slope } -1 \text{ starts} \\
f_x = (t_j')_x - (p_i)_x & \text{slope 0 starts.}
\end{array}$$

Hence the slope changes by $+1$ at the first and the last turning point, while it changes by -1 at the second and the third turning point.

Now, $C_{f_y}(f_x) = \sum_{i,j} c_{ij}(f_x)$, hence C_{f_y} is a sum of piecewise linear continuous functions and therefore it gets its maximum value at some turning point of the c_{ij}s. Let g be a turning point, and the value associated to it is the corresponding horizontal transposition. Moreover, let $G_y = \{g_1 \le g_2 \le \cdots \le g_K\}$ be the multiset of the turning points ordered in increasing order; note that distinct functions c_{ij} may have a turning point at same location. So, for each i, j, the four values

$$\begin{array}{ll}
(t_j)_x - (p_i')_x & \text{(type 1)} \\
(t_j)_x - (p_i)_x & \text{(type 2)} \\
(t_j')_x - (p_i')_x & \text{(type 3)} \\
(t_j')_x - (p_i)_x & \text{(type 4)}
\end{array}$$

are in the lists of the G_ys, and each knows its "type" shown above.

To evaluate $C_{f_y}(f_x)$ at its all turning points we scan the turning points g_k and keep track of the changes of the slope of the function C_{f_y}. Then it is easy to evaluate $C_{f_y}(f_x)$ at the next turning point from its value at the previous one. Let v be the previous value and let s represent the slope and let `best` store the largest value. The evaluation is given below.

(1) $v \leftarrow 0$; $s \leftarrow 0$; `best` $\leftarrow 0$
(2) **for** $k \leftarrow 1, \ldots, K$ **do**
(3) $\quad v \leftarrow v + s(g_k - g_{k-1})$
(4) $\quad$ **if** g_k is of type 1 or type 4 **then** $s \leftarrow s + 1$
(5) $\quad$ **else** $s \leftarrow s - 1$.
(6) $\quad$ **if** $v >$ `best` **then** `best` $\leftarrow v$

This should be repeated for all different $f_y \in H$. We next describe a method that generates the turning points in increasing order simultaneously for all different f_y. The method will traverse T using four pointers (the four "types") per an element of P. A priority queue is again used for sorting the translations given by the $4m$ pointers; the x-coordinates of the translations then give the turning points in ascending order. At each turning point we update the counters v_h and s_h where h is given by the y-coordinate of the translation.

We need two traversal orders of T. The first is the one we have used so far, the lexicographic order of the starting points t_j. This order is given as $\overline{T}$. The second order is the lexicographic order of the end points t_j'. Let $\overline{T}'$ be the end points in the sorted order.

Let q_i^1, q_i^2, q_i^3, and q_i^4 be the pointers of the four types, associated with element π_i of P. Pointers q_i^1 and q_i^2 traverse $\overline{T}$, and pointers q_i^3 and q_i^4 traverse $\overline{T}'$. The translation associated with the current value of each pointer is given as

$$\begin{array}{ll}
tr(q_i^1) = q_i^1 - p_i' & tr(q_i^2) = q_i^2 - p_i \\
tr(q_i^3) = q_i^3 - p_i' & tr(q_i^4) = q_i^4 - p_i.
\end{array}$$

So, when the pointers refer to t_j or t_j', the x-coordinate of these translations give the turning points, of types 1, 2, 3, and 4, associated with the intersection of π_i and τ_j. The y-coordinate gives the vertical translation $(t_j)_y - (p_i)_y$ that is needed to classify the turning points correctly.

During the traversal, all $4m$ translations tr given by the $4m$ pointers are kept in a priority queue. By repeatedly extracting the minimum from the queue (and updating the pointer that gives this minimum tr) we get the translations in ascending lexicographic order, and hence the x-coordinate of the translations gives all turning points in ascending order.

Let $f = (f_x, f_y)$ be the next translation obtained in this way. Then we retrieve the slope counter s_{f_y} and the value counter

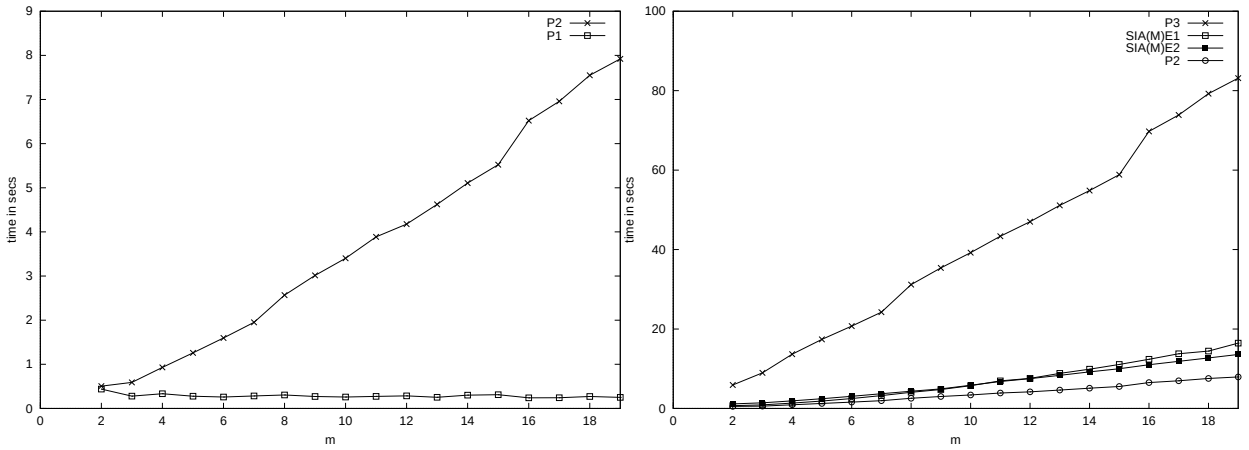

Figure 3: Experimenting on the length of the query pattern. To the left, comparing P1 vs. P2; to the right P2 vs. P3, SIA(M)E$_1$ and SIA(M)E$_2$.

v_{f_y}. Assuming that we have also stored the last turning point z_{f_y} at which v_{f_y} was updated, we can now perform the following updates. If $f_x \neq z_{f_y}$, then let $v_{f_y} \leftarrow v_{f_y} + s_{f_y}(f_x - z_{f_y})$ and $z_{f_y} \leftarrow f_x$. Moreover, if f is of types 1 or 4, then let $s_{f_y} \leftarrow s_{f_y} + 1$ otherwise $s_{f_y} \leftarrow s_{f_y} - 1$.

In this way we obtain the values v_h for each function C_h and each $h \in H$, at each turning point. By (1), the maximum value of C must be among them.

The described procedure needs $O(n \log n)$ time for sorting T into $\overline{T}$ and $\overline{T}'$, and $O(mn \log m)$ time for generating the $O(mn)$ turning points in increasing order. At each turning point we have to retrieve the corresponding slope and value counters using the vertical translation $h \in H$ as the search key. Since we are dealing with MIDI intervals, $|H| \leq 255$ (independently of m and n), we can use bucketing to manage these counters. This gives a total time requirement $O(n \log n + mn \log m)$ and space requirement $O(m + n)$.

Let us finally point out an anomaly of our algorithm. It is quite common that a note occurs several times simultaneously on different tracks (instruments). Then it is possible that our algorithm finds occurrences whose length is larger than the total length of the line segments of P, because a line segment of P can intersect several line segments of T simultaneously. Formally, however, the algorithm solves problem ($P3$) as we defined. It is rather easy to remove the anomaly: just replace any overlapping line segments in P and T by one long segment covering their joint time intervals. Another possibility, which does not destroy the original structure, is to insist that the overlaps between P and T that are counted to the common time must be one-to-one (instead of many-to-one or one-to-many). This can be achieved using appropriate counters associated with each line segment of P.

6 Experiments

We made brief experiments on our algorithms solving problems ($P1$), ($P2$), and ($P3$). Let us call the algorithms P1, P2, and P3, respectively. To have something to compare with, we implemented the SIA(M)E$_1$ and SIA(M)E$_2$ algorithms (Meredith et al., 2001; Wiggins et al., 2003). SIA(M)E$_1$ uses hashing [1] to maintain the translation vectors in order and has a worst case time bound of $O((mn)^2)$ but works in time $O(mn)$ on the average. SIA(M)E$_2$ merges sorted sets of translation vectors and has a worst case time bound of $O(mn \log(mn))$. In the experiments, we used the database of the Mutopia project [2]. For the experiments we created 10 copies of it. So, in total our database contained 2790 musical documents that comprised 1,215,520 chords and 2,158,840 notes. The degree of polyphony within the database is 1.78 on the average (max 15).

First, we studied the difference in performances of P1 and P2, and then those of the four algorithms, P2, P3, SIA(M)E$_1$, and SIA(M)E$_2$, capable of dealing with approximate matching. Recall that P2, SIA(M)E$_1$, and SIA(M)E$_2$ solve problem ($P2$) (while P3 solves problem ($P3$)). The experiments were carried out in a PC with Intel Pentium IV of 2.26GHz and 1 GB of RAM under a modified RedHat Linux with kernel 2.4.18 and gcc compiler 3.2.2.

We experimented on different lengths of query patterns for it is the more interesting parameter out of the parameters m and n. The range of considered lengths was $[2, 19]$ that, we claim, covers the lengths of the patterns normally used in query by humming applications.

At each considered length, 25 query patterns were randomly picked up from the database. Thus, it was guaranteed that at least one occurrence was to be found with each possible query. The times we report are the averages of the 25 repetitions of each considered query pattern length. We measured the true, elapsed times for carrying out the query tasks.

Fig. 3 gives the running times of our experiments. First, consider the problem ($P1$). The first plot (to the left) confirms that P1 can be expected to run independently of m. Therefore, P1 is a useful algorithm that should be included in an content-based music retrieval query engine for the cases when exact occurrences need to be found in a large database, although P2 can be used for the task, as well.

[1] We used a hashing table of size $2 \times m \times n$, as suggested by Wiggins et al. (2003).

[2] http://www.mutopiaproject.org/

The second plot illustrates, that there is no significant differences in the behaviour of the three algorithms capable of solving ($P2$) [3]. However, out of those three, P2 seems to slightly outperform SIA(M)$_{E_1}$ and SIA(M)$_{E_2}$ in this experiment. Recall that P2 needs only $O(m)$ space in contrast to the $O(nm)$ space for the hashing table [4] required by SIA(M)$_{E_1}$. Moreover, if the size of the hashing table used by SIA(M)$_{E_1}$ needs to be diminished, the quadratic time behaviour $O((mn)^2)$ becomes prevailing.

P3 is rather clearly outperformed by the three algorithms solving ($P2$). However, there is no doubt about its usefulness. To our knowledge it is the only algorithm capable of solving problem ($P3$).

7 Conclusion

We presented efficient algorithms for transposition invariant content-based retrieval on polyphonic music. The algorithms adapt themselves to different variations of the problems such as to weighted matching or to patterns that consist of rectangles instead of line segments. The presented algorithms have been implemented and embedded in our C-BRAHMS engine available at http://www.cs.helsinki.fi/group/cbrahms/demoengine/.

Acknowledgement

We are grateful to Mika Turkia for the implementations.

References

Bentley, J. and Ottmann, T. (1979). Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28:643–647.

Clausen, M., Engelbrecht, R., Meyer, D., and Schmitz, J. (2000). Proms: A web-based tool for searching in polyphonic music. In *Proceedings of the International Symposium on Music Information Retrieval (ISMIR'2000)*, Plymouth, MA.

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

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

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

Holub, J., Iliopoulos, C., and Mouchard, L. (2001). Distributed string matching using finite automata. *Journal of Automata, Languages and Combinatorics*, 6(2):191–204.

Lemström, K. (2000). *String Matching Techniques for Music Retrieval*. PhD thesis, University of Helsinki, Department of Computer Science. Report A-2000-4.

Lemström, K. and Navarro, G. (2003). Flexible and efficient bit-parallel techniques for transposition invariant approximate matching in music retrieval. In *Proceedings of the 10th International Symposium on String Processing and Information Retrieval (SPIRE 2003)*, Manaus, Brazil. (To appear).

Lemström, K. and Tarhio, J. (2003). Transposition invariant pattern matching for multi-track strings. *Nordic Journal of Computing*. (To appear).

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

Meredith, D., Wiggins, G., and Lemström, K. (2001). Pattern induction and matching in polyphonic music and other multidimensional datasets. In *the 5th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI'2001)*, volume X, pages 61–66, Orlando, FLO.

Mongeau, M. and Sankoff, D. (1990). Comparison of musical sequences. *Computers and the Humanities*, 24:161–175.

Ukkonen, E., Lemström, K., and Mäkinen, V. (2003). Sweepline the music! In *Computer Science in Perspective — Essays Dedicated to Thomas Ottmann*, volume 2598 of *Lecture Notes in Computer Science*, pages 330–342. Springer-Verlag.

Wiggins, G., Lemström, K., and Meredith, D. (2003). SIA(M): A family of efficient algorithms for translation-invariant pattern matching in multidimensional datasets. (Submitted).

Zhu, Y. and Shasha, D. (2003). Warping indexes with envelope transforms for query by humming. In *Proceedings of the ACM SIGMOD 2003 International Conference on Management of Data*, San Diego.

[3] Interestingly enough, in the same surroundings except that the compiler was changed to RedHat's gcc compiler 2.96, SIA(M)$_{E_2}$ was clearly outperformed by P2 and SIA(M)$_{E_1}$.

[4] In our implementation, the required size for the hashing table is actually $O(n'm)$, where n' is the maximum length of the pieces of music in the database.