Sweepline the Music!* **

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. The problem of matching sets of points or sets of horizontal line segments in plane under translations is considered. For finding the exact occurrences of a point set of size m within another point set of size n we give an algorithm with running time $O(mn)$, and for finding partial occurrences an algorithm with running time $O(mn \log m)$. To find the largest overlap between two line segment patterns we develop an algorithm with running time $O(mn \log(mn))$. All algorithms are based on a simple sweepline traversal of one of the patterns in the lexicographic order. The motivation for the problems studied comes from music retrieval and analysis.

1 Introduction

Computer-aided retrieval and analysis of music offers fascinating challenges for pattern matching algorithms. The standard writing of music as exemplified in Fig. 1 and Fig. 2 represents the music as notes. Each note symbol gives the pitch and duration of a tone. As the pitch levels and durations are normally limited to a relatively small set of discrete values, the representation is in fact a sequence of discrete symbols. Such sequences are a natural application domain for combinatorial pattern matching.

The so-called query-by-humming systems [10, 18, 14] are a good example of music information retrieval (MIR) systems that use pattern matching methods. A query-by-humming system has a content-based query unit and a database of symbolically encoded music such as popular melodies. A user of the database remembers somewhat fuzzily a melody and wants to know if something similar is in the database. Then the user makes a query to the database by humming (or whistling or playing by an instrument) the melody, and the query system should then find from the database the melodies that match best with the given one. The search over the database can be done very fast using advanced algorithms for approximate string matching, based on the edit distance and discrete time-warping (e.g. [13]).

Problems get more complicated if instead of simple melodies, we have polyphonic music like symphony orchestra scores. Such a music may have very complex structure with several notes simultaneously on and several musical themes

* In *Comp. Sci. in Perspective*, LNCS 2598, pp. 330–342, 2003. © Springer-Verlag
** A work supported by the Academy of Finland.

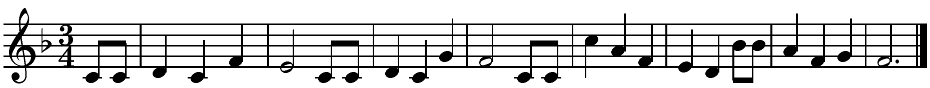

Fig. 1. A melody represented in common music notation.

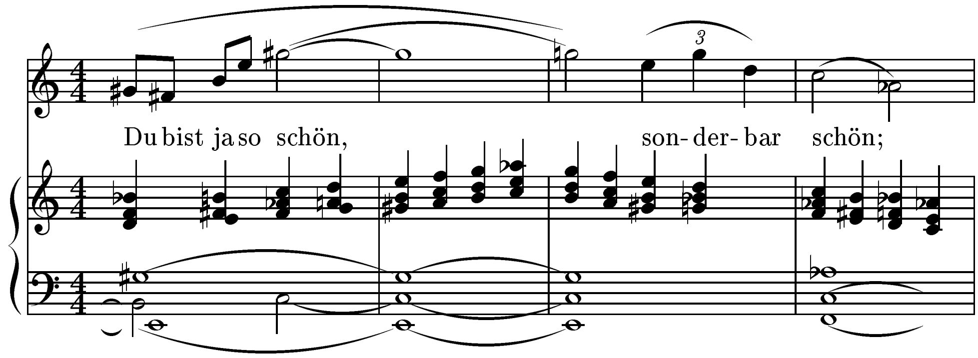

Fig. 2. An excerpt of Einojuhani Rautavaara's opera *Thomas* (1985). Printed with the permission of the publisher Warner/Chappell Music Finland Oy.

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.

Returning back to our examples, the melody of Fig. 1 is given in Fig. 3 using so-called piano-roll representation. The content is now already quite explicit: 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. Fig. 4 gives the piano-roll representation of the music of Fig. 2.

In western music, when comparing different pieces of music and melodies in particular, the absolute pitch levels of the notes are not of the primary interest. Rather, the pitch level differences between successive notes, the *intervals*, really matter. If a melody is obtained from another one by a transposition, which means adding a constant to the pitch levels of the notes, the transposed melody is still considered the same. Hence in music comparison it is customary to require invariance with respect to pitch level transpositions.

Similarly, for the durations of the notes, it is the duration ratio between successive notes that is important. Rewriting using shorter or longer notes does not basically change the melody as far as the duration ratios stay invariant. However, the number of possible rescalings is very small in practice. More common is, that the same musical theme occurs in otherwise varied forms, perhaps

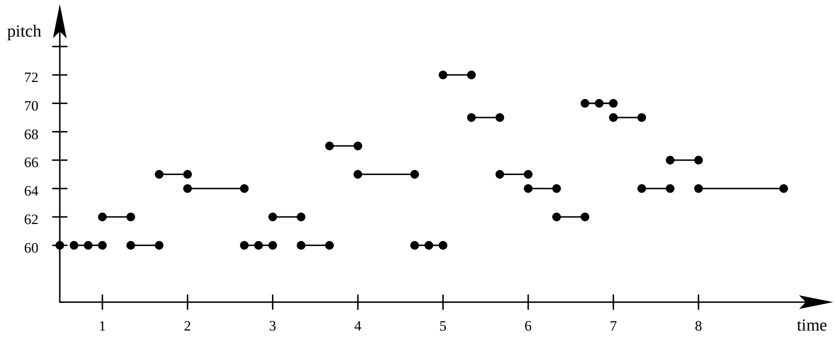

Fig. 3. The example of Fig. 1 in piano-roll representation.

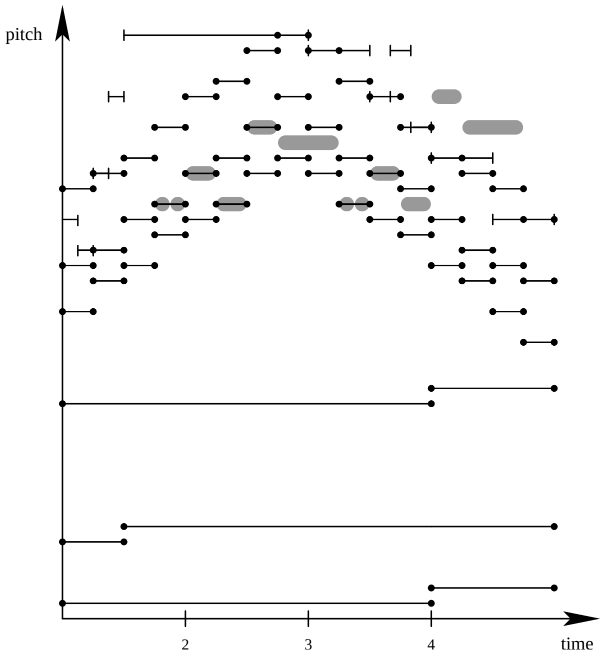

Fig. 4. The example of Fig. 2 in piano-roll representation. The notes belonging to the soloist's melody are represented distinctly. Moreover, the first twelve notes of Fig. 3 are shown by shading in a translated position such that the starting points of 6 notes match and the total length of the overlap time is 6 quarter notes.

with some notes added or deleted, or the intervals or the duration ratios slightly changed.

This suggests that comparing musical sequences would need an appropriate form of approximate pattern matching that is invariant with respect to pitch transpositions. This could be combined with invariance with respect to rescaling of the tempo of the music but here also repeated searches with different scales can be feasible.

In this paper we will use a simple two-dimensional geometric representation of music, abstracted from the piano-roll representation. In this representation, a piece of music is a collection of horizontal line segments in the Euclidean two-dimensional space $\mathbb{R}^2$. The horizontal axis refers to the time, the vertical to the pitch values. As we do not discretize the available pitch levels nor the onset times or durations of the notes, the representation is more powerful than the standard notation of music as notes. In some cases, however, we consider the effect of the discretization on the efficiency of the algorithms.

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. 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 typically assume that T represents a large database of music while P is a shorter query piece, but it is also possible that both P and T refer to the same 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 notes.

($P2$) Find all translations of P that give a partial match of the on-set times of the notes of P with the notes of T.

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

Fig. 4 illustrates all three problems. There is no solution of ($P1$), but the shading shows a solution to ($P2$) and ($P3$).

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 $O(mn \log(mn))$ and space $O(mn)$. When the number of possible pitch levels is a finite constant (as is the case with music), the running time and working space of the algorithm for ($P3$) become $O(mn \log m)$ and $O(m)$. All algorithms are based on a simple *sweepline*-type [4] scanning of T. We assume that T and P are given in the lexicographic order of the starting points of the line segments. Otherwise time $O(n \log n + m \log m)$ and space $O(n + m)$ for sorting should be added to the above bounds.

Our problems are basic pattern matching questions in music when transpositions are allowed and note additions and deletions are modeled as partial matches. Problem $(P3)$ also allows local changes in note durations. Tolerance to interval changes could be incorporated by representing the notes as narrow rectangles instead of lines.

Related work. Our results on problems $(P1)$ and $(P2)$ slightly improve the recent results of Meredith et al. [19, 21] who, using similar algorithms, gave for $(P1)$ a time bound $O(mn)$ and a space bound $O(n)$, and for $(P2)$ a time bound $O(mn \log(mn))$ and a space bound $O(mn)$.

A popular approach to MIR problems has been to use different variants of the edit distance evaluated with the well–known dynamic programming algorithm. This line of research was initiated by Mongeau and Sankoff [20] who presented a method for the comparison of monophonic music. Edit distances with transposition invariance have been studied in [16, 13, 17]. In [8], dynamic programming is used for finding polyphonic patterns with restricted gaps. Bit–parallelism is used in the algorithm of [15] for finding transposed monophonic patterns within polyphonic music, and in the algorithm of [11] for finding approximate occurrences of patterns with group symbols.

Most of the MIR studies so far consider only monophonic music, and from the rhythmic information only the order of the notes is taken into account. An example of a more general approach is the MIR system PROMS [6] which works with polyphonic music and some durational information.

Our work has connections also to computational geometry, where point pattern matching under translations (and under more general transformations) is a well-studied field; see e.g. the survey by Alt and Guibas [1]. A problem very close to $(P1)$ is to decide whether $A = \mathcal{T}(B)$, where $\mathcal{T}$ is an arbitrary rigid transformation, and A and B are point patterns (such as the sets of the starting points of the line segments in P and T), both of size n. This can be solved in $O(n \log n)$ time [3] by using a reduction to exact string matching. Our problem is more difficult since we are trying to match one point set with a subset of the other. A general technique, called alignment method in [12], can be used to solve problems $(P1)$ and $(P2)$ in time $O(mn \log n)$; we will sketch this solution in the beginning of Sect. 3. Allowing approximate point matching in problems $(P1)$ and $(P2)$ will make them much harder; an $O(n^6)$ algorithm was given in [2] for the case $|A| = |B| = n$, and only an improvement to $O(n^5 \log n)$ [9] has been found since. However, a relaxed problem in which a point is allowed to match several points in the other pattern, leading to Hausdorff distance minimization under translations, can be solved in $O(n^2 \log^2 n)$ time [5], when distances are measured by L_∞ norm (see citations in [1] for other work related to Hausdorff distance). Recently, a special case of $(P1)$ in which points are required to be in integer coordinates was solved in $O(n \log n \log N)$ time [7] with a Las Vegas algorithm, where N is the maximum coordinate value.

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 $1 \leq j \leq n$. Checking for some such translation $f = t_j - p_1$, that also the other points $p_2 + f, \ldots, p_m + f$ of $\overline{P} + f$ match, can be performed in time $O(m \log n)$ using some geometric data structure to query $\overline{T}$ in logarithmic time. This leads to total running time of $O(mn \log n)$.

We can do better by utilizing 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 translation is compared to the current f_j, and the pointer q_i will be 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 procedure is given below in Fig. 5.

(1) **for** $i \leftarrow 1, \ldots, m$ **do** $q_i \leftarrow -\infty$
(2) $q_{m+1} \leftarrow \infty$
(3) **for** $j \leftarrow 1, \ldots, n-m$ **do**
(4) $\quad f \leftarrow t_j - p_1$
(5) $\quad i \leftarrow 1$
(6) $\quad$ **do**
(7) $\quad\quad i \leftarrow i + 1$
(8) $\quad\quad q_i \leftarrow \max(q_i, t_j)$
(9) $\quad\quad$ **while** $q_i < p_i + f$ **do** $q_i \leftarrow next(q_i)$
(10) $\quad$ **until** $q_i > p_i + f$
(11) $\quad$ **if** $i = m + 1$ **then** $output(f)$
(12) **end for**.

Fig. 5. Algorithm 1.

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 $(\overline{P}+f) \cap \overline{T}$ 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 $(\overline{P}+f) \cap \overline{T}$ is largest possible.

There are $O(mn)$ translations f such that $(\overline{P}+f) \cap \overline{T}$ is nonempty, namely the translations $t_j - p_i$ for $1 \leq j \leq n$, $1 \leq i \leq n$. Checking the size of $(\overline{P}+f) \cap \overline{T}$ 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 $(\overline{P}+f) \cap \overline{T}$ 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 $(\overline{P}+f) \cap \overline{T}$ 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 $1 \leq i \leq m$. 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, $1 \leq i \leq m$. 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) $f \leftarrow -\infty$; $c \leftarrow 0$;
do
(2) $\quad f' \leftarrow min(F)$; $update(F)$
(3) $\quad$ **if** $f' = f$ **then** $c \leftarrow c + 1$
(4) $\quad$ **else** $\{output(f, c);\ f \leftarrow f';\ c \leftarrow 1\}$
(5) **until** $f = \infty$

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, $1 \leq h \leq k + 1$,*

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 $\overline{P}_h + f$. $\square$

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 $r \leq (k + 1)n$ 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 as $\pi_i = [p_i, p'_i]$ for $1 \leq i \leq m$, and the line segments of T as $\tau_j = [t_j, t'_j]$ for $1 \leq i \leq n$.

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

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

Our problem is to maximize this function. The value of $C(f)$ 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, that is, 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 here 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

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

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 $c_{ij}(f_x)$. 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

$f_x = (t_j)_x - (p'_i)_x$	slope 1 starts
$f_x = \min\big((t_j)_x - (p_i)_x, (t'_j)_x - (p'_i)_x\big)$	slope 0 starts
$f_x = \max\big((t_j)_x - (p_i)_x, (t'_j)_x - (p'_i)_x\big)$	slope -1 starts
$f_x = (t'_j)_x - (p_i)_x$	slope 0 starts.

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 $g_1 \le g_2 \le \cdots \le g_K$ be the turning points in increasing order, each point listed according to its multiplicity; note that different functions c_{ij} may have the same turning point. So, for each i, j, the four values

$(t_j)_x - (p'_i)_x$	(type 1)
$(t_j)_x - (p_i)_x$	(type 2)
$(t'_j)_x - (p'_i)_x$	(type 3)
$(t'_j)_x - (p_i)_x$	(type 4)

are in the lists of the g:s, 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. The evaluation is given below.

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

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

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

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 π_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 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 $\overline{T}'$, $O(mn \log m)$ 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. Hence we need in general time $O(\log |H|) = O(\log(mn))$. This gives a total time requirement $O(mn \log(mn))$ and space requirement $O(mn)$.

When P and T represent music, the size of H is limited independently of m and n. Certainly $|H|$ is less than 300 and often much smaller. We can use bucketing to manage the slope and value counters. The resource bounds become $O(n \log n + mn \log m)$ for time and, more importantly, $O(m + n)$ for space.

6 Conclusion

We presented efficient algorithms for three pattern matching problems. Their motivation comes from music but as computational problems they have a clear geometric nature. Also our algorithms are geometric, based on simple sweepline techniques. The algorithms adapt themselves easily to different variations of the problems such as to weighted matching or to patterns that consist of rectangles instead of line segments.

Experimentation with the algorithms on real music data would be interesting, to see whether these techniques can compete in accuracy, flexibility and speed with the dynamic programming based methods.

References

1. H. Alt and L. Guibas. Discrete geometric shapes: Matching, interpolation, and approximation. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 121 – 153. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 1999.
2. H. Alt, K. Mehlhorn, H. Wagener, and E. Welzl. Congruence, similarity and symmetries of geometric objects. *Discrete Comput. Geom.*, 3:237–256, 1988.
3. M.D. Atkinson. An optimal algorithm for geometric congruence. *J. Algorithms*, 8:159– 172, 1997.
4. J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28:643–647, September 1979.
5. L.P. Chew and K. Kedem. Improvements on geometric pattern matching problems. In *Proceedings of the Scandinavian Workshop Algomthm Theory (SWAT)*, pages 318–325, 1992.
6. M. Clausen, R. Engelbrecht, D. Meyer, and J. Schmitz. Proms: A web-based tool for searching in polyphonic music. In *Proceedings of the International Symposium on Music Information Retrieval (ISMIR'2000)*, 2000.

7. R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 592–601. ACM Press, 2002.
8. M.J. Dovey. 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, 2001.
9. A. Efrat and A. Itai. Improvements on bottleneck matching and related problems using geometry. In *Proceedings of the twelfth annual symposium on Computational geometry*, pages 301–310. ACM Press, 1996.
10. A. Ghias, J. Logan, D. Chamberlin, and B.C. Smith. Query by humming - musical information retrieval in an audio database. In *ACM Multimedia 95 Proceedings*, pages 231–236, 1995. Electronic Proceedings: http://www.cs.cornell.edu/Info/Faculty/bsmith/query-by-humming.
11. J. Holub, C.S. Iliopoulos, and L. Mouchard. Distributed string matching using finite automata. *Journal of Automata, Languages and Combinatorics*, 6(2):191–204, 2001.
12. D. Huttenlocher and S. Ullman. Recognizing solid objects by alignment with an image. *Intern. J. Computer Vision*, 5:195–212, 1990.
13. K. Lemström. *String Matching Techniques for Music Retrieval.* PhD thesis, University of Helsinki, Department of Computer Science, 2000. Report A-2000-4.
14. K. Lemström and S. Perttu. SEMEX - an efficient music retrieval prototype. In *Proceedings of the International Symposium on Music Information Retrieval (ISMIR'2000)*, 2000.
15. K. Lemström and J. Tarhio. Detecting monophonic patterns within polyphonic sources. In *Content-Based Multimedia Information Access Conference Proceedings (RIAO'2000)*, pages 1261–1279, 2000.
16. K. Lemström and E. Ukkonen. Including interval encoding into edit distance based music comparison and retrieval. In *Proceedings of the AISB'2000 Symposium on Creative & Cultural Aspects and Applications of AI & Cognitive Science*, pages 53–60, 2000.
17. V. Mäkinen, G. Navarro, and E. Ukkonen. Algorithms for transposition invariant string matching. Technical Report TR/DCC-2002-5, Department of Computer Science, University of Chile, 2002.
18. R.J. McNab, L.A. Smith, D. Bainbridge, and I.H. Witten. The New Zealand digital library MELody inDEX. *D-Lib Magazine*, 1997. http://www.nzdl.org/musiclib.
19. D. Meredith, G.A. Wiggins, and K. Lemström. Pattern induction and matching in polyphonic music and other multi-dimensional data. In *the 5th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI'2001)*, volume X, pages 61–66, 2001.
20. M. Mongeau and D. Sankoff. Comparison of musical sequences. *Computers and the Humanities*, 24:161–175, 1990.
21. G.A. Wiggins, K. Lemström, and D. Meredith. SIA(M) — a family of efficient algorithms for translation invariant pattern matching in multidimensional datasets. Manuscript (submitted), September 2002.