# Transposition and time-scale invariant geometric music retrieval

Lemström, Kjell

Heidelberg, Berlin, Springer Verlag,
2010

Lemström , K 2010 , ' Transposition and time-scale invariant geometric music retrieval ' , in T Elomaa , H Mannila & P Orponen (eds) , Algorithms and applications . Essays dedicated to Esko Ukkonen on the occasion of his 60th birthday . , Lecture Notes in Computer Science 6060 , Heidelberg, Berlin, Springer Verlag, .

http://hdl.handle.net/10138/23644

# Transposition and Time-Scale Invariant Geometric Music Retrieval

Kjell Lemström

University of Helsinki
Department of Computer Science
klemstro@cs.helsinki.fi

**Abstract.** This paper considers how to adapt geometric algorithms, developed for content-based music retrieval of symbolically encoded music, to be robust against time deformations required by real-world applications. In this setting, music is represented by sets of points in plane. A matching, pertinent to the application, involves two such sets of points and invariances under translations and time scalings. We give an algorithm for finding exact occurrences, under such a setting, of a given query point set, of size $m$, within a database point set, of size $n$, with running time $O(mn^2 \log n)$; partial occurrences are found in $O(m^2n^2 \log n)$ time. The algorithms resemble the sweepline algorithm introduced in [1].

## 1 Introduction

Query-by-humming is a problem that has fascinated researchers working in the music-retrieval area for over fifteen years. First, the music under investigation was assumed to be monophonic (see Fig. 1) [2], later the term has been used with a wider meaning addressing problems where the task is to search for excerpts of music, resembling a given query pattern, in a large database. Moreover, both the query pattern and the database may be polyphonic, and the query pattern constitutes only a subset of instruments appearing in the database representing possibly a full orchestration of a musical work. Although current audio-based methods can be applied to rudimentary cases where queries are directed to clearly separable melodies, the general setting requires methods based on symbolic representation that are truly capable of dealing with polyphonic subset matching.

To this end, several authors have recently used geometric-based modeling of music [1,3,4,5]. Geometric representations usually also take into account another feature intrinsic to the problem: the matching process ignores extra intervening notes in the database that do not appear in the query pattern. Such extra notes are always present because of the polyphony, various noise sources and musical decorations. There is, however, a notable downside of the current geometric methods: they do not allow distortions in tempo (except for individual outliers that are not even discovered) that are inevitable in the application. Even if the query could be given exactly on tempo, the occurrences in the database would be time-scaled versions of the query (requiring *time-scale invariance*). If the query

T. Elomaa et al. (Eds.): Ukkonen Festschrift 2010, LNCS 6060, pp. 170–181, 2010.

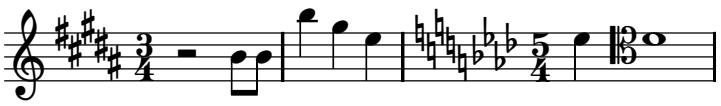

**Fig. 1.** An excerpt of a well-known melody in common music notation. Let us have a closer look at the last bar with a change in key and time signature: The first note is associated with pitch value "Es" (or E flat). It is followed by a c-clef, which looks like a letter "k" to this author. Note also the resemblance of the last note to the letter "o".

is to be given in a live performance, more or less local jittering will inevitably take place and a stronger invariance, namely the *time-warping invariance* [6], would be a desired property for the matching process.

In this paper, new time-scale invariant geometric algorithms that deal with symbolically encoded, polyphonic music will be introduced. We use the pitch-against-time representation of note-on information, as suggested in [5] (see Fig 2). The musical works in a database are concatenated in a single geometrically represented file, denoted by $T$; $T = t_0, t_1, \ldots, t_{n-1}$, where $t_j \in \mathbb{R}^2$ for $0 \leq j \leq n-1$. In a typical retrieval case the query pattern $P$, $P = p_0, p_1, \ldots, p_{m-1}$; $p_i \in \mathbb{R}^2$ for $0 \leq i \leq m-1$, to be searched for is often monophonic and much shorter than the database $T$ to be searched, that is $m \ll n$. It is assumed that $P$ and $T$ are given in the lexicographic order. If this is not the case, the sets can be sorted in $m \log m$ and $n \log n$ times, respectively.

The problems under consideration are modified versions of two problems originally represented in [1]. The following list gives both the original problems and the modifications under consideration; for the partial matches in P2 and S2, one may either use a threshold $\alpha$ to limit the minimum size of an accepted match, or to search for maximally sized matches only.

(P1) Find translations of $P$ such that each point in $P$ matches with a point in $T$.
(P2) Find translations of $P$ that give a partial match of the points in $P$ with the points in $T$.
(S1) Find time-scaled translations of $P$ such that each point in $P$ matches with a point in $T$.
(S2) Find time-scaled translations of $P$ that give a partial match of the points in $P$ with the points in $T$.

Ukkonen et al introduced online algorithms PI and PII solving the original problems P1 and P2 in $O(mn)$ and $O(mn \log m)$ worst case times, respectively, in $O(m)$ space [1]. Lemström et al [7] showed that the practical performance can be improved at least by an order of magnitude by combining sparse indexing and filtering. P2 is known to belong to a problem family for which $o(mn)$ solutions are conjectured not to exist, there is, however, an online approximation algorithm for it running in time $O(n \log n)$. [8]. Romming and Selfridge-Field [9]

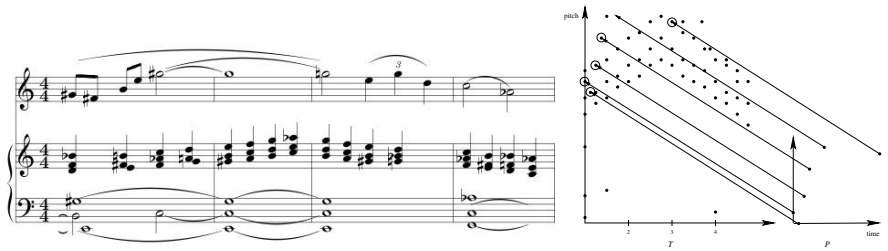


**Fig. 2.** A polyphonic music score, to the left, is represented by a pointset $T$, in the middle, in the geometric representation. Pointset $P$, to the right, corresponds to the first two and a half bars of the melody line (the highest staff of the score) but the fifth point has been delayed somewhat. The depicted trans-set vectors correspond to the translation $f$ that gives the largest partial match of $P$ within $T$.

gave a geometric hashing-based algorithm for S2, which works in $O(n^3)$ space and $O(n^2m^3)$ time.

This paper studies another way to solve S1 and S2. As stated above, in this case, the timing (rhythm) of the music is distorted by a uniform scaling factor; methods ignoring such distortions are called time-scale invariant [6]. The novel time-scale invariant algorithms to be introduced resemble Ukkonen et al's PI and PII algorithms. The new algorithm for S1 runs in time $O(mn^2 \log n)$; the algorithm for S2 in $O(m^2n^2logn)$ time.

## 2 Related Work

Let us denote by $\alpha$ a similarity threshold for P2, and let $p_0, p_1, \ldots, p_{m-1}$ and $t_0, t_1, \ldots, t_{n-1}$ be the pattern and database points, respectively, *lexicographically sorted* according to their co-ordinate values: $p_i < p_{i+1}$ iff $p_i.x < p_{i+1}.x$ or $(p_i.x = p_{i+1}.x$ and $p_i.y < p_{i+1}.y)$, and $t_j < t_{j+1}$ iff $t_j.x < t_{j+1}.x$ or $(t_j.x = t_{j+1}.x$ and $t_j.y < t_{j+1}.y)$. In our application the elapsing time runs along the horisontal axes, represented by $x$, the perceived height, the *pitch*, is represented by $y$. A translation of $P$ by vector $f$ is denoted as $P + f$: $P + f = p_0 + f, \ldots, p_{m-1} + f$. Using this notation, problem P1 is expressible as the search for a subset $I$ of $T$ and some $f$ such that $P + f = I$. Note that decomposing translation $f$ into horisontal and vertical components $f.x$ and $f.y$, respectively, captures two musically distinct phenomena: $f.x$ corresponds to aligning the pattern time-wise, $f.y$ to *transposing* the musical excerpt to a lower or higher key (see Fig. 2). Note also that a musical time-scaling $\sigma$, $\sigma \in \mathbb{R}^+$, has an effect only on the horisontal translation, the vertical translation stays intact.

*Example 1.* Let $p = (1, 1)$, $f = (2, 2)$ and $\sigma = 3$. Then $p + \sigma f = (7, 3)$.

A straight-forward algorithm solves P1 and P2 in $O(mn \log(mn))$ time. The algorithm first collects exhaustively all the translations mapping a point in $P$ to

another point in $T$. The set of the collected translation vectors are then sorted in lexicographic order. In the case of P1, a translation $f$ that has been used $m$ times corresponds to an occurrence; in the case of P2, any translation $f$ that has been used at least $\alpha$ times would account for an occurrence. Thoughtful implementations of the involved scanning (sorting) of the translation vectors, will yield an $O(mn)$ ($O(mn \log m)$) time algorithm for P1 (P2) [1].

Indeed, the above $O(mn \log m)$ time algorithm is the fastest online algorithm known for P2. Moreover, any significant improvement in the asymptotic running time, exceeding the removal of the logarithmic factor, cannot be seen to exist, for P2 is known to be a `3SUM`-hard problem [8]. It is still possible that P2 is also a `Sorting X+Y` -hard problem, in which case Ukkonen et al's PII algorithm would already be an optimal solution. In [8], Clifford et al introduced an $O(n \log n)$ time approximation algorithm for P2.

To make the queries more efficient, several indexing schemes have been suggested. The first indexing method using geometric music representation was suggested by Clausen et al. [3]. Their sublinear query times were achieved by using inverted files, adopted from textual information retrieval. The performance was achieved with a lossy feature extraction process, which makes the approach non-applicable to problems P1 and P2. Typke et al. [4] proposed the use of metric indexes that works under robust geometric similarity measures. The approach lacks flexibility on features pertinent to the application: it is very difficult to adopt it to support translations and partial matching at the same time. Lemström et al's approach [7] combines sparse indexing and (practically) lossless filtering. Their index is used to speed up a filtering phase that charts all the promising areas in the database where real occurrences could reside. Once a query has been received, the filtering phase works in time $O(g_f(m) \log n + n)$ where function $g_f(m)$ is specific to the applied filter $f$. The last phase involves checking the found $c_f$ ($c_f \leq n$) candidate positions using Ukkonen et al's PI or PII algorithm executable in worst-case time $O(c_f m)$ or $O(c_f m \log m)$, respectively.

A brute-force solution for S2 would work in time $O(m^3 n^3)$ and space $O(mn)$: First all translation vectors are calculated, exhaustively, in lexicographic order. This gives $m$ increasing sequences of vectors (pairs of real values) each of length $n$. Then, each possible time-scaling value is selected by choosing two vectors from two distinct sequences; there are $O(m^2 n^2)$ possibilities in this choice. For each time-scaling value, the maximum co-occurrence between pattern and database needs to be determined. This can be done by checking whether each of the remaining $m-2$ sequences (each containing $n$ vectors) includes a vector that accords with the chosen scaling vector. This is feasible in time $O(mn)$. Candidates thus found are to be verified by checking that the pitch intervals also match. The only non-brute-force method for S2 is by Romming and Selfridge-Field [9]. It is based on geometric hashing and works in $O(n^3)$ space and $O(n^2 m^3)$ time. By applying a window on the database such that $w$ is the maximum number

of events that occur in any window, the above complexities can be restated as $O(w^2n)$ and $O(wnm^3)$, respectively.

## 3 Matching Algorithms

Our matching algorithms for the time-scale invariant problems S1 and S2 resemble somewhat Ukkonen et al's original PI and PII algorithms in that they all use a priority queue as the focal structure. Ukkonen et al's PI and PII work on trans-set translations, or *trans-set vectors*, $f = t - p$ (see Fig. 2), where $p$ and $t$ are points in a given query pattern, of length $m$, and in the underlying database, of length $n$, respectively. Let us assume (without loss of generality) that all the points, both in the pattern and in the database, are unique. The number of trans-set vectors is within the range $[n + m - 1, nm]$. In order to be able to build an index on the database in an offline phase, Lemström et al's method [7] is based on *intra-set vectors*. For instance, translation vector $f$ is an *intra-pattern vector*, if there are two points $p$ and $p'$ $(p, p' \in P)$ such that $p + f = p'$. *Intra-database vectors* are defined accordingly. Naturally, the number of intra-pattern and intra-database vectors are $O(m^2)$ and $O(n^2)$, respectively. Lemström et al study several heuristics, relevant to the application, to keep the index structure of a linear $O(n)$ size.

The set of *positive intra-pattern vectors* include translations $p_{i'} - p_i$ where in the case of S1: $0 \leq i < m$ and $i' = i + 1$, and in the case of S2: $0 \leq i < i' \leq m$. The set of *positive intra-database vectors* include translations $t_{k'} - t_k$ where, independently of the case, $0 \leq k < k' \leq n$. For the convenience of the algorithms, we pretend that there are an extra element $p_m$ in the pattern and another extra element $t_n$ in the database. The matching algorithms take as input intra-set vectors, stored in tables $K[i], 0 \leq i < m$. Table $K[i]$ stores intra-database translations that may match [1] the positive intra-pattern vectors $p_{i'} - p_i$, i.e., translation vectors starting at point $p_i$. See Fig. 3 for an illustration on tables $K[i]$.

The entries in our main data structures will be sorted in a lexicographic order. We will specify the underlying order by an ordered set $\aleph$. $\aleph$ is formed by members of $\{a, b, s\}$, where $a, b$ and $s$ correspond to the accordingly named columns in tables $K[i]$. For instance, lexicographic order $\langle a, s \rangle$ is firstly based on the values on column $a$ (the starting point of the associated intra-database vector), secondly on the values on column $s$ (the associated scaling value). A main loop that goes exhaustively through all the possibilities of positive intra-pattern and positive intra-database vectors to initialise the tables $K[i]$ is needed. To this end, let a positive intra-database vector $g = t_{k'} - t_k$ be such that there is a positive intra-pattern vector $f = p_{i'} - p_i$ for which $g.y = f.y$ (ie. the pitch intervals of the two vectors match). Because $g$ may be part of an occurrence,

[1] Please note the distinction between an occurrence and a match. An *occurrence* involves as many *matching* pairs of intra-database and intra-pattern vectors as is required by the problem specification.

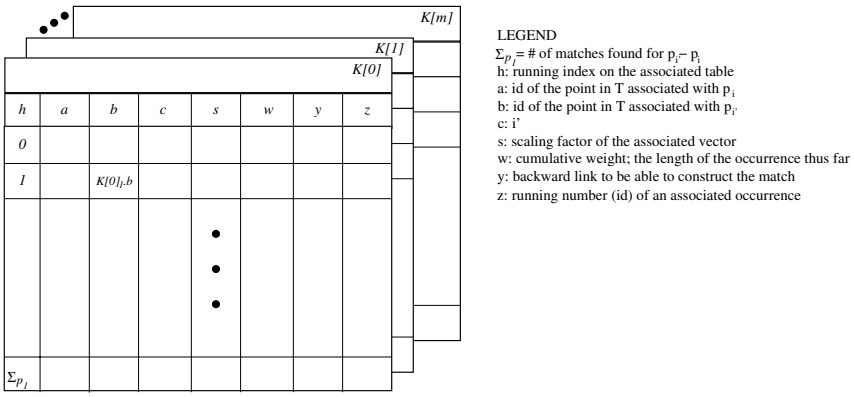


**Fig. 3.** Illustration of the main data structure. Each $K[i]$ stores intra-database vectors $t_{k'} - t_k,\ 0 \leq k < k' \leq m-1$ that matches with an intra-patter vector $p_{i'} - p_i$ (where, in the case of S1: $0 \leq i < m-1$ and $i' = i+1$, and in the case of S2 $0 \leq i < i' \leq m-1$) with any positive time-scaling $\sigma \in \mathbb{R}^+$.

a new row, let it be the $h$th, in $K[i]$ is allocated and the following updates are conducted:

$$K[i]_h.a \leftarrow k; \quad K[i]_h.b \leftarrow k'; \tag{1}$$

$$K[i]_h.s \leftarrow \frac{t_{k'}.x - t_k.x}{p_{i'}.x - p_i.x}; \tag{2}$$

$$K[i]_h.y \leftarrow \mathbf{nil}; \quad K[i]_h.w \leftarrow 1; \tag{3}$$

$$K[i]_h.c \leftarrow i'; \quad K[i]_h.z \leftarrow 0. \tag{4}$$

Above, in (1), the associated starting and ending points of the matching intra-database vector are stored in $K[i]_j.a$ and $K[i]_j.b$, respectively. The required time scaling for the intra-vectors to match is stored in $K[i]_j.s$ (2); here extra carefulness is needed in order to avoid dividing by zero: If both the numerator and the denominator equal zero, we set $K[i]_h.s = 1$. If only one of them equals zero, the whole row is deleted from the table altogether (it would represent for an impossible time scaling). The columns $y$ and $w$, initialised in (3), are used for backtracking a found occurrence and storing the length of a candidate occurrence, respectively. The last columns, initialised in (4), will be needed when searching for partial occurrences (in Section 3.2): column $c$ stores the ending point of the associated intra-pattern vector, $z$ is used for identifying an occurrence.

Denoting by $\Sigma_{p_i}$ the number of rows generated above for table $K[i]$, $0 \leq i < m$, for the aforementioned extra elements (for the convenience of the algorithms) we set:

$$K[i]_{\Sigma_{p_i}}.a \leftarrow K[i]_{\Sigma_{p_i}}.b \leftarrow \infty;\ K[i]_{\Sigma_{p_i}}.s \leftarrow K[i]_{\Sigma_{p_i}}.w \leftarrow 0;\ K[i]_{\Sigma_{p_i}}.c \leftarrow i+1$$

As each iteration of the main loop takes a constant time, this exhaustive initialisation process runs in $O(n^2m^2)$ time. Finally, the columns in $K[i]$ are

sorted in lexicographic order $\langle a, s\rangle$. The matching algorithms have an associated priority queue $Q_i$ for each table $K[i]$, $0 < i \leq m$ [2]. For $Q_i$, a lexicographic order $\langle b, s\rangle$ is used. As a reminder, the order is given in the superscript of a priority queue (e.g. $Q_i^{\langle b,s\rangle}$).

### 3.1 S1: Quest for Time Scaled Exact Occurrences

Once the tables $K[i]$ have been initialised and their columns have been sorted in lexicographic order $\langle a, s\rangle$, the transposition-invariant time-scaled exact occurrences can be found using the matching algorithm given in Fig. 4. The algorithm works by observing *piecewise matches* between positive intra-database and intra-pattern vectors

$$t_{k_{i'}} - t_{k_i} = \sigma_i(p_{i+1} - p_i) \tag{5}$$

that are stored in the associated $K[i]$. Above $\sigma_i \in \mathbb{R}^+$ is the time-scaling factor (recall Example 1). The piecewise matches may form a chain $T_{\tau_0 \ldots \tau_{m-1}} = t_{\tau_0}, t_{\tau_1}, \ldots, t_{\tau_{m-1}}$, where $\tau_0, \tau_1, \ldots, \tau_{m-1}$ is an increasing sequence of indices in $T$; $t_{\tau_{i+1}} - t_{\tau_i} = \sigma(p_{i+1} - p_i)$ for $0 \leq i < m-1$ and $\sigma \in \mathbb{R}^+$ is a time-scaling factor common to all the piecewise matches in the chain. Naturally, such a chain would constitute a transposition-invariant, time-scaled exact occurrence. A chain $T_{\tau_0 \ldots \tau_{m'}}$, $m' < m-1$, is called a *prefix occurrence* (of length $m'$); $T_{\tau_{m'-1}, \tau_{m'}}$ is the *final suffix* of the prefix occurrence $T_{\tau_0 \ldots \tau_{m'}}$.

Let $t_{\tau_{i+1}} - t_{\tau_i}$ (that, by definition, equals $\sigma(p_{i+1} - p_i)$) be the final suffix of a prefix occurrence $T_{\tau_1 \ldots \tau_{m'}}$. The prefix occurrence is *extensible* if there is a piecewise match $t_{k'_{i+1}} - t_{k_{i+1}} = \sigma(p_{i+2} - p_{i+1})$ such that

$$t_{\tau_{i+1}} = t_{k_{i+1}} \tag{6}$$

and scaling factor $\sigma$ is the one that was used in forming $T_{\tau_1 \ldots \tau_{m'}}$. The binding in Equation 6 is called the *binding of extension*, $t_{\tau_{i+1}} - t_{\tau_i}$ the *antecedent* and $t_{k'_{i+1}} - t_{k_{i+1}}$ the *postcedent of the binding*.

**Lemma 1.** *If a prefix occurrence is extensible, its final suffix is also extensible.*

*Proof.* Immediate. □

To be more efficient, at point $i+1$, the algorithm actually considers any piecewise match $t_{k'_i} - t_{k_i} = \sigma_i(p_{i+1} - p_i)$ as an antecedent to the binding and tries to extend it. Because in this case the piecewise matches in an occurrence chain have to be consecutive in $P$, the antecedents of the binding are all found in $K[i]$ and their possible extensions, postcedents, in $K[i+1]$. To process all the bindings of extension at point $i+1$, therefore, involves going through all the entries both in $K[i]$ and in $K[i+1]$. To make this process efficient, no entry of either of the tables should be observed more than once for one iteration. In order for this to be possible, both sides of the binding of extension (associated with antecedents

[2] A single priority queue would suffice, but the algorithm would become more complicated.

```
TimeScaledExactOccurrence(K[i])
(1)  for j ← 0, ..., Σ_{p_0} do
(2)      Q_1^{⟨b,s⟩} ← push(&K[0]_j)
(3)  for i ← 1, ..., m − 1 do
(4)      q ← pop(Q_i^{⟨b,s⟩})
(5)      for j ← 0, ..., Σ_{p_i − 1} do
(6)          while [q.b, q.s] < [K[i]_j.a, K[i]_j.s] do
(7)              q ← pop(Q_i^{⟨b,s⟩})
(8)          if [q.b, q.s] = [K[i]_j.a, K[i]_j.s] then
(9)              K[i]_j.w ← q.w + 1
(10)             K[i]_j.y ← q
(11)             Q_{i+1}^{⟨b,s⟩} ← push(&K[i]_j)
(12)             q ← pop(Q_i^{⟨b,s⟩})
(13)     K[i]_{Σ_{p_i}}.s ← ∞
(14)     Q_{i+1}^{⟨b,s⟩} ← push(&K[i]_{Σ_{p_i}})
(15) if K[m − 1]_j.w = m for some 0 ≤ j ≤ Σ_{p_{m−1}} then report an occurrence
```

**Fig. 4.** Online algorithm for finding transposition-invariant time-scaled exact occurrences

and postcedents) should be enumerated in the same (increasing) order. However, the lefthand side of the binding involves end points of the intra-database vectors in $K[i]$ and the righthand side the start points of the intra-database vectors in $K[i+1]$. Therefore, we use a priority queue $Q_{i+1}^{\langle b,s\rangle}$ whose entries are addresses to rows associated with the antecedents of the binding at $i+1$. In this way, the binding of extension at $i+1$ can be done efficiently by enumerating the entries in $Q_{i+1}$ and $K[i+1]$. Note that the set of piecewise matches extended this way also includes all the final suffixes, and therefore, according to Lemma 1, also all the prefix occurrences.

The binding of extension takes place in line (8) of the algorithm. If a piecewise match is extensible, its length is updated (line 9) and a backtracking link is stored (line 10). The latter becomes useful if any of the extended piecewise matches extends into a proper occurrence, and the whole occurrence is to be revealed (instead of just reporting it).

*Correctness.* Let there be an occurrence $t_{\tau_0}, t_{\tau_1}, \ldots, t_{\tau_{m-1}}$, such that $t_{\tau_{i+1}} - t_{\tau_i} = \sigma(p_{i+1} - p_i)$ for $0 \le i < m-1$ with some $\sigma \in \mathbb{R}^+$. It is obvious from the way the tables are constructed that every element $t_{\tau_{i+1}} - t_{\tau_i}$, associated with a piecewise match, is stored in the corresponding table $K[i]$. Clearly the first antecedent $t_{\tau_1} - t_{\tau_0}$ is treated correctly: It is inserted in the priority queue $Q_1$ in line (2); being part of a proper occurrence, the binding of extension can be done with a correct time-scaling factor and, therefore, the equation condition in line (8) is satisfied. The length of this prefix occurrence is set to 2 (line (9)), and the address of this newly found final suffix is stored in $Q_{i+1}$ (line (11)) to be considered as an antecedent for a binding of extension at iteration 2. Let us now assume that

everything is done correctly up to point $i$, $2 < i < m-1$ and we are dealing with the element $t_{\tau_{i+1}} - t_{\tau_i}$. It is pushed in the priority queue $Q_{i+1}$ in line (12) just before the iteration $i+1$ of the loop in line (3) starts. Then again, being part of a proper occurrence, the antecedent $t_{\tau_{i+1}} - t_{\tau_i}$, found in $Q_{i+1}$, and the postcedent $t_{\tau_{i+2}} - t_{\tau_{i+1}}$, found in $K[i+1]$, can be bound: the extension condition is met and the prefix occurrence is increased to $i+2$, and the address of the new final suffix is passed to further iterations. So, by induction, the algorithm finds the existing occurrences.

That the algorithm does not wrongly increase the length of a piecewise match (causing reports of spurious occurrences or occurrences of lengths exceeding $m$) is down to the binding of extension, i.e., the condition in line (8). Remember that we assume points in $T$ and in $P$ to be unique. Let us make a counter assumption that a piecewise match may be unintentionally extended. To that end, at some iteration $j$ of the inner loop (starting at line (5)) there has to be at least two antecedents in $Q_i$ that can be bound with a postcedent stored in $K[i]_j$ (line (8)). Let $q$ and $q'$ be entries in $Q_i$ that are antecedents for a postcedent in $K[i]_j$, where $K[i]_j.a = t_l$, and let $q$ correspond to a piecewise match $t_l - t_k = \sigma(p_i - p_{i-1})$ and $q'$ to $t_l - t_{k'} = \sigma(p_i - p_{i-1})$. But this is impossible unless $t_k = t_{k'}$ which contradicts our assumption of points being unique in $T$. □

*Analysis.* Let us denote by $|Q_i|$ and $|K[j]|$ the number of entries in $Q_i$ and $K[j]$, respectively. Clearly, in this case, $|Q_i| \leq |K[i-1]|$ for $1 \leq i \leq m$. Moreover, let $\Sigma = max_{i=1}^{m}(|Q_i|, |K[i-1]|)$. The outer loop (line (3)) is iterated $m$ times. Within the inner loop (line (5)), all the entries in $Q_i$ and in $K[i]$ are processed exactly once, resulting in $O(\Sigma)$ entry processing steps. The only operation taking more than a constant time is the updating of the priority queue; it takes at most $O(\log \Sigma)$ time. Thus, the algorithm runs in time $O(m\Sigma \log \Sigma)$. Moreover, the tables $K[i]$ and priority queues $Q_i$ require $O(m\Sigma)$ space.

In this case $\Sigma = O(n^2)$, because each table $K[i]$ contains the piecewise matches for the positive intra-pattern vector $p_{i+1} - p_i$, and there are $O(n^2)$ possibilities to this end. □

### 3.2 S2: Quest for Time Scaled Partial Occurrences

In order to be able to find transposition-invariant time-scaled partial occurrences, we need the two extra columns $c$ and $z$, that were initialised in Equation 4, for tables $K[i]$. Recall that $K[i]_h.c$ stores the ending point $i'$ for an intra-pattern vector $p_{i'} - p_i$ that is found to match an intra-database vector $t_{k'} - t_k$ with some scaling factor $\sigma_i$. Column $z$ is used for storing a running number that is used as an id, for a found partial occurrence. Furthermore, we use an extra table, denoted by $\kappa$, for storing all the found occurrences.

The structure of the algorithm, given in Fig. 5, is similar to the previous algorithm. Again, at point $i$, the antecedents in $Q_i$ are to be extended by postcedents found in $K[i]$. However, as we are looking for partial occurrences this time, we

TimeScaledPartialOccurrence($K[i]$)
(0) $\ell \leftarrow 0$
(1) **for** $j \leftarrow 0, \ldots, \Sigma_{p_0}$
(2) $\quad Q^{\langle b,s\rangle}_{K[0]_j.c} \leftarrow$ **push**$(\&K[0]_j)$
(3) **for** $i \leftarrow 1, \ldots, m-1$ **do**
(4) $\quad q \leftarrow$ **pop**$(Q_i^{\langle b,s\rangle})$
(5) $\quad$ **for** $j \leftarrow 0, \ldots, \Sigma_{p_i-1}$ **do**
(6) $\quad\quad$ **while** $[q.b, q.s] < [K[i]_j.a, K[i]_j.s]$ **do**
(7) $\quad\quad\quad q \leftarrow$ **pop**$(Q_i^{\langle b,s\rangle})$
(8) $\quad\quad$ **if** $[q.b, q.s] = [K[i]_j.a, K[i]_j.s]$ **then**
(9) $\quad\quad\quad$ **while min**$(Q_i^{\langle b,s\rangle}) = [q.b, q.s]$ **do**
(10) $\quad\quad\quad\quad r \leftarrow$ **pop**$(Q_i^{\langle b,s\rangle})$
(11) $\quad\quad\quad\quad$ **if** $r.w > q.w$ **then** $q \leftarrow r$
(12) $\quad\quad\quad K[i]_j.w \leftarrow q.w + 1$
(13) $\quad\quad\quad K[i]_j.y \leftarrow q$
(14) $\quad\quad\quad$ **if** $K[i]_j.w = \alpha$ **then**
(15) $\quad\quad\quad\quad \ell \leftarrow \ell + 1$
(16) $\quad\quad\quad\quad K[i]_j.z = \ell$
(17) $\quad\quad\quad\quad \kappa[\ell] \leftarrow \&K[i]_j$
(18) $\quad\quad\quad$ **if** $K[i]_j.w > \alpha$ **then**
(19) $\quad\quad\quad\quad K[i]_j.z = q.z$
(20) $\quad\quad\quad\quad \kappa[q.z] \leftarrow \&K[i]_j$
(21) $\quad\quad\quad Q^{\langle b,s\rangle}_{K[i]_j.c} \leftarrow$ **push**$(\&K[i]_j)$
(22) $\quad\quad\quad q \leftarrow$ **pop**$(Q_i^{\langle b,s\rangle})$
(23) $\quad K[i]_{\Sigma_{p_i}}.s \leftarrow \infty$
(24) $\quad Q^{\langle b,s\rangle}_{i+1} \leftarrow$ **push**$(\&K[i]_{\Sigma_{p_i}})$
(25) ReportOccurrences($\kappa$)

**Fig. 5.** Online algorithm for finding transposition-invariant time-scaled partial occurrences

cannot rely on piecewise matches that are consecutive in $P$ but any piecewise match associated with a positive intra-pattern vector

$$t_{k_{i'}} - t_{k_i} = \sigma_i(p_{i'} - p_i) \tag{7}$$

has to be considered. Here $0 \leq k_i < k_{i'} \leq n-1$; $0 \leq i < i' \leq m-1$ and $\sigma_i \in \mathbb{R}^+$. Given a threshold $\alpha$, a chain $T_{\tau_0 \ldots \tau_{\beta-1}}$, such that $t_{\tau_j} - t_{\tau_{j-1}} = \sigma(p_{\pi_j} - p_{\pi_{j-1}})$, for $0 < j \leq \beta$, $\beta \geq \alpha$, where $\tau_0 \ldots \tau_{\beta-1}$ and $\pi_0 \ldots \pi_{\beta-1}$ are increasing sequences of indices in $T$ and $P$, respectively, would constitute for a transposition-invariant time-scaled partial occurrence (of length $\beta$).

That piecewise matches can now be between any two points in the pattern makes the problem somewhat more challenging. This has the effect that, at point $i$, pushing a reference to a priority queue (lines (2) and (21) of the algorithm) may involve any future priority queue, from $Q_{i+1}$ to $Q_m$, not just the successive one as in the previous case; the correct priority queue is the one that is stored in

$K[i]_j.c$ (recall that it stores the end point of the intra-pattern vector associated with the piecewise match). Conversely, the antecedents at point $i$ (stored in $Q_i$) may include references to any past tables, from $K[0]$ to $K[i-1]$, expanding the size of the priority queue $Q_i$.

The two remaining differences to the algorithm above, are in lines (11) and (14-20). In line (11), the algorithm chooses to extend the piecewise match that is associated with the longest prefix occurrence. This is a necessary step, once again, because we are no more dealing with piecewise matches that are consecutive in $P$. In lines (14-20) the algorithm deals with a found occurrence. Lines (14-17) deal with a new occurrence: generate a new running number, $\ell$, for it (that is used as its id) and store a link to the found occurrence to the table of occurrences $\kappa$. Lines (18-20) deal with extending a previously found occurrence.

*Correctness and Analysis.* Denoting by $\Sigma = max_{i=1}^{m}(|Q_i|, |K[i-1]|)$, with an analogous reasoning to that of the previous analysis, we arrive at similar complexities: the algorithm runs in $O(m\Sigma \log \Sigma)$ time and $O(m\Sigma)$ space. Let us now analyse the order of $\Sigma$ in this case. Still it holds that for a positive intra-pattern vector, $p_{i'} - p_i$, there are $O(n^2)$ possible piecewise matches. However, the table $K[i]$ may contain entries associated with piecewise matches with any positive intra-pattern vector ending at point $i'$. Thus, $max_{i=1}^{m}(|K[i-1]|) = O(mn^2)$. As $|Q_i| \leq |K[i-1]|$ for $0 < i \leq m$ and $m = O(n)$, we conclude that the algorithm has an $O(m^2n^2 \log n)$ running time and works in a space $O(m^2n^2)$.

The proof of the correctness of this algorithm is left for an interested reader. □

## 4 Conclusions

In this paper we suggested novel content-based music retrieval algorithms for polyphonic, geometrically represented music. The algorithms are both transposition and time-scale invariant. Given a (polyphonic) query pattern $P = p_0, \ldots, p_{m-1}$ to be searched for in a polyphonic music database $T = t_0, \ldots, t_{n-1}$, the algorithms run in $O(m\Sigma \log \Sigma)$ time and $O(m\Sigma)$ space, where $\Sigma = O(n^2)$ when searching for exact occurrences under such a setting, and $\Sigma = O(n^2m)$ when searching for partial occurrences. Whether this is an improvement in practice over the existing algorithm by Romming and Selfridge-Field [9], working in space $O(n^3)$ and time $O(n^2m^3)$, is left for future experiments on real data.

However, the new approach seems to be very flexible: at the present the author is adopting the approach to a more complex case, where an uneven time deformation is known just to preserve the order of the notes; there are no known solutions for this time-warping invariant problem [6]. Moreover, it seems that with slight modifications to the data structures and ideas presented by Lemström, Mikkilä and Mäkinen in [7], it would be possible to adopt the idea of using a three-phase searching process (indexing, filtering and checking) with a smaller search space and a better running time to those presented here.

## References

1. Ukkonen, E., Lemström, K., Mäkinen, V.: Sweepline the music! In: Klein, R., Six, H.-W., Wegner, L. (eds.) Computer Science in Perspective. LNCS, vol. 2598, pp. 330–342. Springer, Heidelberg (2003)
2. Ghias, A., Logan, J., Chamberlin, D., Smith, B.: Query by humming - musical information retrieval in an audio database. In: ACM Multimedia 1995 Proceedings, San Francisco, CA, pp. 231–236 (1995)
3. Clausen, M., Engelbrecht, R., Meyer, D., Schmitz, J.: Proms: A web-based tool for searching in polyphonic music. In: Proceedings of the International Symposium on Music Information Retrieval (ISMIR 2000), Plymouth, MA (October 2000)
4. Typke, R.: Music Retrieval based on Melodic Similarity. PhD thesis, Utrecht University, Netherlands (2007)
5. Wiggins, G., Lemström, K., Meredith, D.: SIA(M)ESE: An algorithm for transposition invariant, polyphonic content-based music retrieval. In: Proceedings of the International Conference on Music Information Retrieval (ISMIR 2002), Paris, France, October 2002, pp. 283–284 (2002)
6. Lemström, K., Wiggins, G.: Formalizing invariances for content-based music retrieval. In: Proceedings of the 10th International Conference on Music Information Retrieval (ISMIR 2009), Kobe, October 2009, pp. 591–596 (2009)
7. Lemström, K., Mikkilä, N., Mäkinen, V.: Filtering methods for content-based retrieval on indexed symbolic music databases. Journal of Information Retrieval 13(1), pp. 1–21 (2010)
8. Clifford, R., Christodoulakis, M., Crawford, T., Meredith, D., Wiggins, G.: A fast, randomised, maximal subset matching algorithm for document-level music retrieval. In: Proceedings of the 7th International Conference on Music Information Retrieval, Victoria, BC, Canada, October 2006, pp. 150–155 (2006)
9. Romming, C., Selfridge-Field, E.: Algorithms for polyphonic music retrieval: The hausdorff metric and geometric hashing. In: Proceedings of the 8th International Conference on Music Information Retrieval (ISMIR 2007), Vienna, Austria, September 2007, pp. 457–462 (2007)