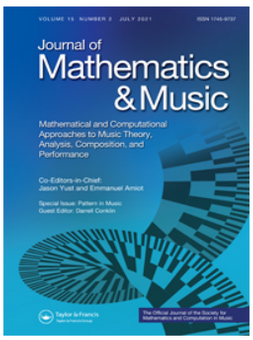

Taylor & Francis
Taylor & Francis Group

Journal of Mathematics and Music

Mathematical and Computational Approaches to Music Theory, Analysis, Composition and Performance

ISSN: (Print) (Online) Journal homepage: https://www.tandfonline.com/loi/tmam20

Discovering distorted repeating patterns in polyphonic music through longest increasing subsequences

Antti Laaksonen & Kjell Lemström

To cite this article: Antti Laaksonen & Kjell Lemström (2021) Discovering distorted repeating patterns in polyphonic music through longest increasing subsequences, Journal of Mathematics and Music, 15:2, 99-111, DOI: 10.1080/17459737.2021.1896811

To link to this article: https://doi.org/10.1080/17459737.2021.1896811

Published online: 05 Apr 2021.

Journal of Mathematics and Music, 2021
Vol. 15, No. 2, 99–111, https://doi.org/10.1080/17459737.2021.1896811

Discovering distorted repeating patterns in polyphonic music through longest increasing subsequences

Antti Laaksonen* and Kjell Lemström

Department of Computer Science, University of Helsinki, Helsinki, Finland

(*Received 1 November 2020; accepted 19 February 2021*)

We study the problem of identifying repetitions under transposition and time-warp invariances in polyphonic symbolic music. Using a novel onset-time-pair representation, we reduce the repeating pattern discovery problem to instances of the classical problem of finding the longest increasing subsequences. The resulting algorithm works in $O(n^2 \log n)$ time where n is the number of notes in a musical work. We also study windowed variants of the problem where onset-time differences between notes are restricted, and show that they can also be solved in $O(n^2 \log n)$ time using the algorithm.

Keywords: repeating pattern discovery; longest increasing subsequences; symbolic music processing

2012 Computing Classification Scheme: music retrieval; pattern matching

1. Introduction

Being able to identify repetitions in music is important for gathering a rich understanding of music (see e.g. Schenker 1954; Lerdahl and Jackendoff 1983; Bent and Drabkin 1987; Temperley 2001) and, also, an elementary task in solving many music-related computational problems. The problem has been widely studied for decades in text and other linear structures. As real music is almost inevitably polyphonic, meaning that a multitude of parallel tones may sound at any time, the corresponding musical task is much more complex. An exhaustive search through all possible sequences of tones would shortly lead to a combinatorial explosion.

In this paper, we consider the problem of finding repeating patterns in Western equal tempered polyphonic music. Our algorithms work on the symbolic music domain using a geometric point set representation of musical notes. By using a symbolic representation we avoid the very challenging problem of the fundamental frequency estimation of a general polyphonic audio music case. However, with imaginative modifications, algorithms developed to one musical domain may become applicable to the other (see e.g. Laaksonen and Lemström 2013).

It is not only polyphony that makes the music pattern discovery task a hard one. There are many possible sources for distortions that should be taken into account. Distortions may be systematic, for instance, identical patterns may appear in different musical key and/or in different written tempo. Such distortions are easy to handle by looking at relative values instead of absolute values: in pitch at the pitch difference, in duration at the quotient between corresponding durations. Such algorithms are called transposition and time-scale *invariant*, respectively,

*Corresponding author. Email: ahslaaks@cs.helsinki.fi

and there are various systematic distortions that may be overcome by applying an appropriate invariance. Unfortunately, this comes with a price: the more the algorithms allow distortions, the more they also generate false-positive repetitions. Therefore, a post-processing phase is often needed to discard extraneous ones.

Lemström and Wiggings (2009) formalized and gave a sparse invariance taxonomy for music information retrieval (MIR) tasks. The taxonomy considers the most common and relevant MIR features: pitch and onset time. Invariances for a feature can be ordered based on their strength. For instance, invariances for onset information in an increasing order are time-position (allows linear shifts in time), time-scale (allows constant scaling in time) and time-warp (allows order preserving local scaling in time).

It would be useful to be able to pick up a suitable method for the task just by analyzing the nature of the underlying dataset (what kind of systematic distortions there are) for understanding the required invariance combination. For instance, assuming that the transposition invariance is indispensable for a musical pattern matching task, depending on how strong invariance is required, there are already efficient algorithms for the transposition and time-shifting invariance-combination (Ukkonen, Lemström, and Mäkinen 2003; Romming and Selfridge–field 2007), transposition and time-scaling combination (Romming and Selfridge-field 2007; Lemström 2010) and transposition and time-warp combination (Lemström and Laitinen 2011). For the repeating pattern discovery problem, there is a method for the transposition and time-shifting invariance-combination (Meredith, Lemström, and Wiggins 2002) but no methods for the transposition and time-scaling or for the transposition and time-warp combination. In this paper, we present an efficient algorithm for the combination of transposition and time-warp invariance for the repeating pattern discovery problem.

In a typical repeating pattern discovery case, the data set is generated by a conversion from either sheet music or a live performance played by some instruments. The latter setting generates much more distortions, thus a stronger invariance combination is needed to deal with the inherent, systematic distortions. For the pitch feature the transposition invariance is strong enough assuming that the pitches are mostly perfectly played. However, in live performances, the tempo varies and onset times are rarely accurate making the time-warp invariance desirable for the onset times. Moreover, the data sets are expected to be dense, that is, there are many musical events irrelevant to the repetition that occur within the time frame of a repetition. Therefore, we should be able to allow *gaps* in repetitions.

Traditional methods for symbolic music information retrieval (MIR) problems have been based on an approximate string matching framework using edit distance (see e.g. Mongeau and Sankoff 1990; Ghias et al. 1995; Uitdenbogerd and Zobel 2002; Clifford et al. 2006; Typke 2007). As the framework has originally been developed for linear strings, it is straightforwardly applicable to handle monophonic music. In order to be able to cope with polyphonic music with a multitude of simultaneous notes and parallelly developing musical themes, Meredith, Lemström, and Wiggins (2002) suggested a piano-roll-like geometric representation of music where each note is represented by a point in the plane. In this representation, the horizontal location (x-axis) gives the onset time for a musical event and the vertical location (y-axis) its pitch level.

In Figure 1, we both illustrate the geometric representation and give an example of a repeating pattern in polyphonic music. In this example, there is a musical motif played in two different keys and tempos. Note that in this example, the recurrent motif is a time-scaled repetition of the original motif, which also means that it is a time-warped repetition. Interestingly, it seems that finding only time-scaled repetitions is much harder than finding time-warped repetitions, as discussed in Section 4.

Given a set of n notes (a musical work), Meredith et al. considered the problems of discovering all maximal repeating patterns and all their occurrences under transposition and time-shifting invariances (here *maximal* means that we cannot add any notes to the repeating pattern). They presented the algorithms SIA and SIATEC for computing the solutions in $O(n^2 \log n)$ and $O(n^3)$

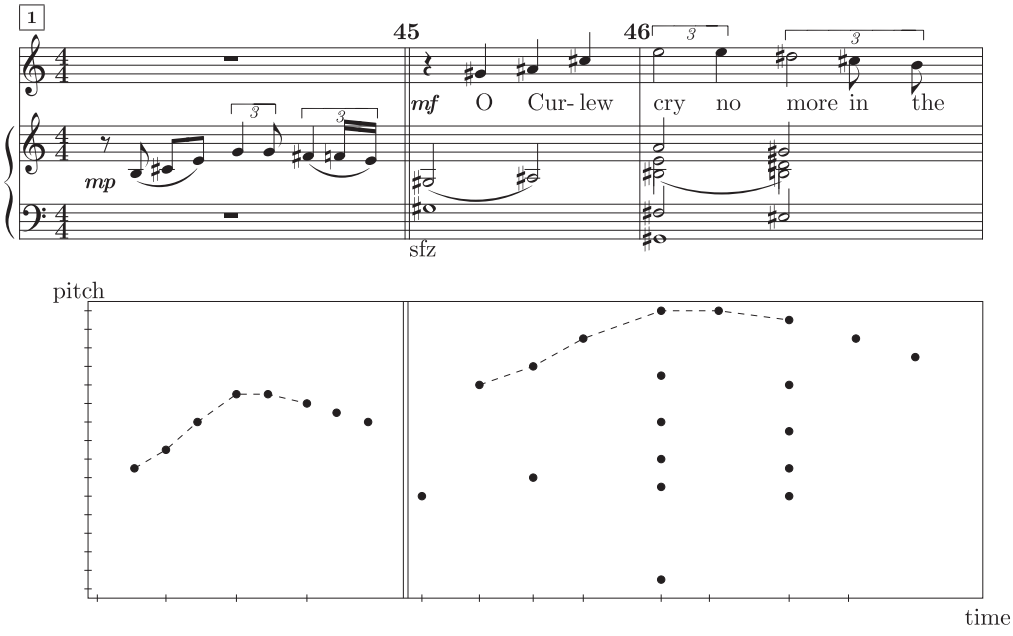

Figure 1. Two excerpts from the piano reduction of Peter Warlock's *The Curlew* (measures 1 and 45–46) in common music notation (above) and the corresponding geometric representation (below) where each note is represented as a point whose x coordinate is the onset time and y coordinate is the pitch. The two representations are synchronized, i.e. the points below are matched with the corresponding notes above, leading to a time-warped point set-representation (note the uneven division in the x axis.) There are several maximal repeating patterns in the example, the most notable being the pattern of six notes represented with dashed lines.

time, respectively. More precisely, the algorithms can be used to process k-dimensional data sets in $O(kn^2 \log n)$ and $O(kn^3)$ time, respectively. In this paper, we only focus on the usual two-dimensional case and ignore the additional k parameter.

The SIA and SIATEC algorithms are tolerant to distortion in the time dimension only to a some extent: if a note is out of time, it is simply discarded. This works rather nicely when only some sporadic notes are distorted. However, when the input is a transcription of a live performance, for instance, the method omits totally the vast majority of the musically meaningful repetitions because a sufficient count of matching individual notes to form a repetition will not be found.

Meredith et al.'s representation gives us a good starting point: it inherently supports point translations (giving us invariances under transposition and pattern location) and sporadic noise omitting. We have to, however, somehow modify it to support arbitrary, local stretching in the time dimension. We will do this by using an onset-time-pair representation where we can solve the original problem of identifying musical repetitions under transposition and time-warp invariances by reducing the problem to instances of the longest increasing subsequence problem.

The rest of the paper is organized as follows: Section 2 discusses the longest increasing subsequence problem with two-dimensional point sets. In Section 3, we describe our repeating pattern discovery algorithm and show variants of the algorithm that can be used to exact and windowed pattern search. All variants of the algorithm work in $O(n^2 \log n)$ time. Finally, Section 4 concludes the paper.

2. Longest increasing subsequences

In this section, we discuss the longest increasing subsequence problem in the context of two-dimensional point sets. It turns out that we can reduce the repeating pattern discovery problem

to instances of the longest increasing subsequence problem, and we use the techniques presented in this section in our pattern discovery algorithm.

Consider a set S of n points in the two-dimensional plane. Each point p is represented as a pair $(x[p], y[p])$ where $x[p]$ and $y[p]$ are real numbers. A *subsequence* of length k is a sequence of k points $p_1, p_2, \ldots, p_k$ where $x[p_i] < x[p_{i+1}]$ for $i = 1, 2, \ldots, k-1$. A subsequence is *increasing* if $y[p_i] < y[p_{i+1}]$ for $i = 1, 2, \ldots, k-1$. In the *longest increasing subsequence* problem, we want to calculate for each point $p \in S$ a value $\texttt{lis}(p)$: the maximum length of an increasing subsequence whose last point is p.

For example, let us consider the set

$$S = \{(1,2), (3,5), (4,8), (6,7), (7,2)\}.$$

In this case, $\texttt{lis}((1,2)) = 1$, $\texttt{lis}((3,5)) = 2$, $\texttt{lis}((4,8)) = 3$, $\texttt{lis}((6,7)) = 3$, and $\texttt{lis}((7,2)) = 1$. For example, $\texttt{lis}((6,7)) = 3$, because there is an increasing subsequence $(1,2), (3,5), (6,7)$ whose length is 3.

The longest increasing subsequence problem is a classical algorithm design problem (see e.g. Knuth 1973, 5.1.4; Fredman 1975), and there are several ways to solve it efficiently. Next, we discuss a well-known dynamic programming algorithm for the problem, which can be implemented in $O(n \log n)$ time using a range query data structure. This algorithm will be used as a building block in our pattern discovery algorithm.

2.1. *Dynamic programming algorithm*

We can efficiently solve the longest increasing subsequence problem using a dynamic programming algorithm. To simplify the presentation of the algorithm, we assume that each point in S has a distinct x coordinate and each y coordinate is an integer between 1 and m where $m = O(n)$. However, we will later describe how the algorithm can be modified so that there are no such restrictions.

The idea of the algorithm is to go through the points from left to right (in increasing x coordinate order) and calculate the $\texttt{lis}$ value for each point using the previously calculated values. More precisely, the algorithm can be implemented as follows, assuming that the points are sorted by x coordinate:

for $i \leftarrow 1$ to n **do**
  $\texttt{lis}[p_i] \leftarrow 1$
  for $j \leftarrow 1$ to $i-1$ **do**
    if $y[p_j] < y[p_i]$ **then**
      $\texttt{lis}[p_i] \leftarrow \max(\texttt{lis}[p_i], \texttt{lis}[p_j] + 1)$
    end if
  end for
end for

When processing a point p_i, the algorithm initially assumes that $\texttt{lis}[p_i] = 1$. After that, the algorithm goes through all previously processed points $p_1, p_2, \ldots, p_{i-1}$ whose $\texttt{lis}$ values are already known. For each such point, the algorithm checks if the corresponding subsequence can be extended by adding the current point p_i to the subsequence. The final $\texttt{lis}$ value is the maximum length of a subsequence that ends at p_i.

For example, when calculating the value $\texttt{lis}((6,7))$ for the input set $\{(1,2), (3,5), (4,8), (6,7), (7,2)\}$, we have already calculated the values $\texttt{lis}((1,2)) = 1$, $\texttt{lis}((3,5)) = 2$ and $\texttt{lis}((4,8)) = 3$. In this case, we can extend the previous subsequences that end at $(1,2)$ and $(3,5)$ because their y values are below 7. The best solution is to extend the subsequence that ends at $(3,5)$, which yields the value $\texttt{lis}((6,7)) = \texttt{lis}((3,5)) + 1 = 3$.

The running time of the above algorithm is $O(n^2)$ because it consists of two loops that go through the input points. Next, we improve the running time of the algorithm by removing the inner loop using a range query data structure.

2.2. *Improving the algorithm using range queries*

To improve the running time of the dynamic programming algorithm, we use a range query data structure that maintains an array of m numbers (where m is the maximum y coordinate) and can efficiently process the following operations:

- `setVal`(k,x): the array value at position k becomes x
- `getMax`(a,b): find the maximum value between array positions a and b

We assume that the array elements are indexed $1,2,\ldots,m$ and every array value is initially 0. Each array position corresponds to a y value in the point set. Using such a data structure, we can implement the algorithm as follows:

for $i \leftarrow 1$ to n **do**
  `lis`$[p_i] \leftarrow$ `getMax`$(1, y[p_i]-1)+1$
  `setVal`$(y[p_i],$ `lis`$[p_i])$
end for

The range query structure contains for each possible y value the maximum length of an increasing subsequence whose last point has that y value. To calculate a value `lis`$[p_i]$, the algorithm efficiently finds the length of a previous subsequence whose last point has a y value between 1 and $y[p_i]-1$, and adds one to that value (if $y[p_i]=1$, we assume that `getMax`$(1,0)=0$). After that, the algorithm updates the range query structure so that it can be used in future searches.

Let us consider again our previous example where the input set consists of points $\{(1,2),(3,5),(4,8),(6,7),(7,2)\}$. In this case $m=8$ and the initial range query array is $[0,0,0,0,0,0,0,0]$. When the algorithm reaches the point $(6,7)$, it has already calculated the values `lis`$((1,2))=1$, `lis`$((3,5))=2$ and `lis`$((4,8))=3$ and the range query array is $[0,1,0,0,2,0,0,3]$. Then, to calculate the value of `lis`$((6,7))$, the algorithm performs a range query `getMax`$(1,6)=2$ and correctly detects an increasing subsequence of length 3. After that, the range query array becomes $[0,1,0,0,2,0,3,3]$.

The efficiency of the algorithm depends on the operations of the range query structure. It turns out that we can implement both `setVal` and `getMax` in $O(\log m)$ time. Since we have assumed that $m=O(n)$, this yields an $O(n \log n)$ time algorithm. More precisely, we can use a segment tree data structure where each leaf has an array value and each internal node has the maximum value in its subtree (de Berg et al. 2010, Chapter 10; Laaksonen 2017, Chapter 9). For example, Figure 2 shows the segment tree that corresponds to the array $[0,1,0,0,2,0,3,3]$. Since each array value belongs to $O(\log n)$ subtrees and any range can be divided into $O(\log n)$ distinct subranges that correspond to segment tree nodes, both the operations can be implemented in $O(\log n)$ time.

2.3. *Generalizing the algorithm*

So far we have assumed that every point has a distinct x coordinate and every y coordinate is an integer between 1 and m where $m=O(n)$. However, it is possible to implement the algorithm without such restrictions.

First, to support multiple points with the same x coordinate, we can defer the `setVal` operations using a *buffer* of updates. After calculating a `lis` value, we add the point p_i to the buffer instead of directly performing a `setVal` operation. Then, always when the x coordinate of a

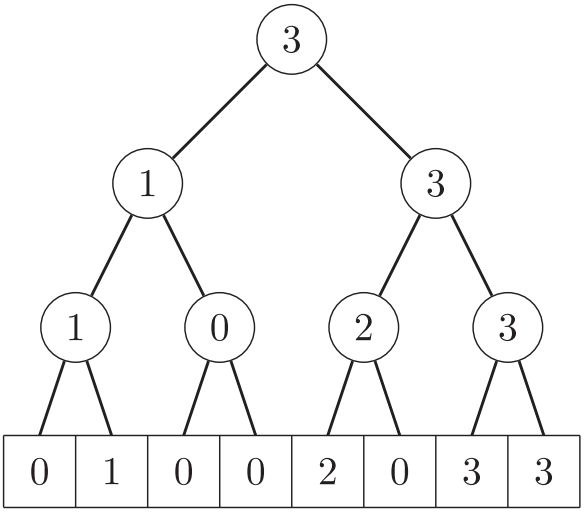

Figure 2. A segment tree that corresponds to the array [0, 1, 0, 0, 2, 0, 3, 3]. Using this data structure, we can efficiently calculate the maximum value in an array range and update an array value. Both the operations can be implemented in $O(\log n)$ time.

point is greater than the x coordinate of the previous point, we perform `setVal` operations for all points in the buffer and then clear the buffer. This ensures that there will be no subsequences where two points have the same x coordinate. We can implement the buffer using a stack, and each point will be added to the buffer and removed from the buffer at most once during the algorithm.

Then, to allow arbitrary y coordinates, we can first *compress* the y coordinates so that they become consecutive integers. This can be done by creating a list that contains all y coordinates of the input points and then sorting the list. After that, we can modify the y coordinates so that the smallest coordinate becomes 1, the second smallest coordinate becomes 2, and so on. For example, if the input point set is $\{(1, 500), (4, -5), (5, 8)\}$, it becomes $\{(1, 3), (4, 1), (5, 2)\}$. Since there are at most n distinct coordinates, each coordinate will be an integer between 1 and n after the compression. This works because the algorithm only uses the order of the coordinates and not their exact values.

We can efficiently implement both the generalizations so that the resulting algorithm still works in $O(n \log n)$ time.

3. Finding repeating patterns

In this section, we present our algorithm for finding repeating patterns in polyphonic music. The algorithm solves the pattern discovery problem by reducing it to instances of the longest increasing subsequence problem. Given a musical work represented as a set S of two-dimensional points, the algorithm calculates for each note pair the maximum length of a repetition that ends at those notes.

Each point $p \in S$ corresponds to a musical note. The x coordinate $x[p]$ denotes the onset time of the note, and the y coordinate $y[p]$ denotes the pitch of the note. A repetition of length k consists of two patterns $a_1, a_2, \ldots, a_k \in S$ and $b_1, b_2, \ldots, b_k \in S$ where $x[a_i] < x[a_{i+1}]$, $x[b_i] < x[b_{i+1}]$ and $y[a_{i+1}] - y[a_i] = y[b_{i+1}] - y[b_i]$ for $i = 1, 2, \ldots, k-1$, i.e. the onset times in both patterns are increasing and each corresponding note pair in the patterns has the same interval.

We first describe the idea of the algorithm in a general setting where the onset-time differences in the patterns are not restricted. After that, we show how the algorithm can be used to find exact repetitions where $x[a_{i+1}] - x[a_i] = x[b_{i+1}] - x[b_i]$ for $i = 1, 2, \ldots, k-1$, creating an alternative to the traditional SIA algorithm. Finally, we present more advanced windowed variants of the algorithm.

3.1. *Reduction to increasing subsequences*

The algorithm is based on two main ideas. First, the algorithm divides the repeating pattern discovery problem into subproblems, each of which has a fixed interval between the two patterns. Second, the algorithm uses an *onset-time-pair representation* to solve each subproblem using the longest increasing subsequence algorithm.

First the algorithm creates a collection of sets $P_0, P_1, P_2, \ldots$ where $P_i = \{(a, b) \mid a \in S, b \in S, y[b] - y[a] = i\}$, i.e. P_i contains all note pairs (a, b) whose interval $y[b] - y[a]$ is constant i. The algorithm processes each set P_i separately and finds all repetitions among each set. When processing a set P_i, the algorithm creates an onset-time-pair representation $C_i = \{(x[a], x[b]) \mid (a, b) \in P_i\}$. Each point in C_i consists of x coordinates of a note pair in P_i. The idea of this representation is that each increasing subsequence in C_i corresponds to a repetition (with interval i) in P_i. The algorithm determines for each point $(x[a], x[b])$ the length of the longest increasing subsequence that ends at that point. This increasing subsequence corresponds to a maximum length repetition in S whose last note pair is (a, b).

As an example, consider a point set

$$S = \{(2,2), (2,4), (2,5), (3,6), (4,4), (5,3), (5,6), (6,1), (6,5)\}.$$

Figure 3(a) shows a repetition of length 3 that consists of patterns $[(2,2), (4,4), (5,3)]$ and $[(2,4), (3,6), (6,5)]$. In this repetition the interval between the patterns is 2. The algorithm detects this repetition when it first creates the set

$$\begin{aligned} P_2 = \{&((2,2),(2,4)), ((2,2),(4,4)), ((2,4),(3,6)), ((2,4),(5,6)), ((4,4),(3,6)), \\ &((4,4),(5,6)), ((5,3),(2,5)), ((5,3),(6,5)), ((6,1),(5,3))\}, \end{aligned}$$

which consists of all note pairs whose interval is 2, and then the set

$$C_2 = \{(2,2), (2,3), (2,4), (2,5), (4,3), (4,5), (5,2), (5,6), (6,5)\},$$

which contains the onset-time-pair representation of the notes. Figure 3(b) shows the increasing subsequence $[(2,2), (4,3), (5,6)]$ in C_2, which corresponds to the repetition in Figure 3(a). This is a longest increasing subsequence that ends at point $(5,6)$ in C_2, which means that the corresponding repetition is a maximum length repetition with interval 2 that ends at notes $(5,3)$ and $(6,5)$ in S. Note that there are also other longest increasing subsequences that end at point $(5,6)$ in C_2; Table 1 shows all such subsequences and the corresponding repeating patterns.

The total running time of the algorithm is $O(n^2 \log n)$. First, the algorithm can generate the P_i sets in $O(n^2 \log n)$ time, because the total number of note pairs in S is $O(n^2)$ and it is possible to sort the pairs and create the sets in $O(n^2 \log n)$ time. After that, using the longest increasing subsequence techniques discussed in Section 2, the algorithm can process each set P_i in $O(k \log k)$ time, where k denotes the number of note pairs in P_i. Since the sum of all k values is $O(n^2)$, the total time required to process all sets is $O(n^2 \log n)$, regardless of the number of note pairs in each set.

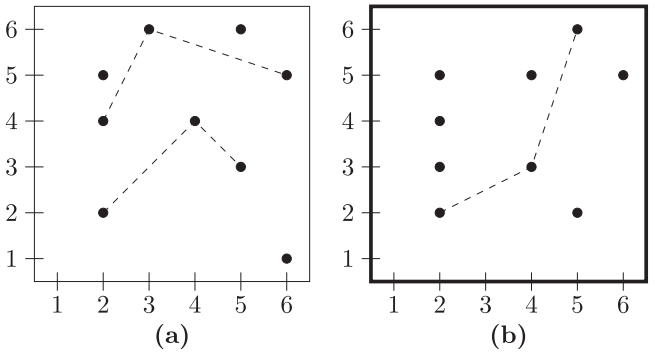

Figure 3. Reduction to the longest increasing subsequence problem for intervals of size two. (a) A repeating pattern that consists of note sequences $[(2,2),(4,4),(5,3)]$ and $[(2,4),(3,6),(6,5)]$. (b) The corresponding longest increasing subsequence $[(2,2),(4,3),(5,6)]$ in the onset-time-pair space.

Table 1. In the example of Figure 3, there are a total of four longest increasing subsequences of length 3 whose last point is $(5,6)$ in the onset-time-pair representation C_2.

Longest increasing subsequence	Repeating pattern occurrence
$[(2,2),(4,3),(5,6)]$	$[(2,2),(4,4),(5,3)]$ and $[(2,4),(3,6),(6,5)]$
$[(2,2),(4,5),(5,6)]$	$[(2,2),(4,4),(5,3)]$ and $[(2,4),(5,6),(6,5)]$
$[(2,3),(4,5),(5,6)]$	$[(2,4),(4,4),(5,3)]$ and $[(3,6),(5,6),(6,5)]$
$[(2,4),(4,5),(5,6)]$	$[(2,2),(4,4),(5,3)]$ and $[(4,4),(5,6),(6,5)]$

Note: This table shows all such subsequences and for each subsequence the corresponding repeating patterns. The first entry in the table corresponds to the repeating pattern shown in Figure 3.

3.2. *Finding exact repetitions*

In the exact version of the problem it is required that $x[a_{i+1}] - x[a_i] = x[b_{i+1}] - x[b_i]$ for $i = 1, 2, \ldots, k-1$. The traditional way to solve the problem is to use the SIA algorithm (Meredith, Lemström, and Wiggins 2002) which first generates and sorts a list of translation vectors between all note pairs in S and then detects repeating patterns as blocks of equivalent vectors in the list. In other words, each block of equivalent vectors corresponds to a maximal translatable pattern of notes. The SIA algorithms works in $O(n^2 \log n)$ time because the list consists of $O(n^2)$ vectors.

We can also approach the exact problem using the longest increasing subsequence technique. In this case, we want to find subsequences whose slope is 1, i.e. the points are located on the same diagonal. For example, Figure 4(a) shows an exact repetition that consists of patterns $[(2,4),(4,4),(5,3)]$ and $[(3,6),(5,6),(6,5)]$, and Figure 4(b) shows the corresponding increasing subsequence $[(2,3),(4,5),(5,6)]$ in C_2. All the points in C_2 are located on the same diagonal.

This variant of the longest increasing subsequence problem is easier to solve than the general problem: we can divide the points in C_i into diagonals where each diagonal has all points (a,b) that have a constant $x[a] - x[b]$ value. For example, in Figure 4(b), all points in the increasing subsequence are located on a diagonal where $x[a] - x[b] = -1$. To find a maximum length repetition, we can simply generate for each diagonal an increasing subsequence that consists of *all* points on the diagonal.

We can go through the diagonals by sorting the pairs by the value of $x[a] - x[b]$, so the algorithm works in $O(n^2 \log n)$ time. In fact, the algorithm works almost like the SIA algorithm: each diagonal in the onset-time-pair representation consists of points where both $x[a] - x[b]$ and $y[a] - y[b]$ are equivalent, which holds exactly when the translation vectors are equivalent.

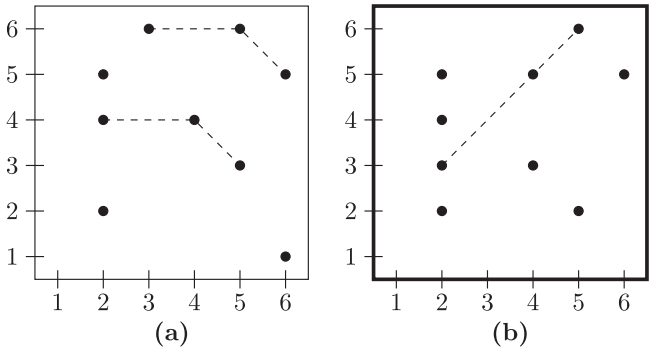

Figure 4. Finding exact repetitions through longest increasing subsequences. (a) A repeating pattern that consists of note sequences $[(2,4),(4,4),(5,3)]$ and $[(3,6),(5,6),(6,5)]$. (b) The corresponding longest increasing subsequence $[(2,3),(4,5),(5,6)]$ in C_2. The repetition is exact, so the slope of the subsequence is 1.

Thus, each diagonal in our algorithm corresponds to a maximal translatable pattern in the SIA algorithm, and the main difference between the algorithms is that our algorithm separately solves a subproblem for each vertical translation (constant interval) while SIA simply processes a sorted list of all vectors.

Note that this exact version of our algorithm finds all maximal translatable patterns, like the SIA algorithm. In other versions of the algorithm, repeating patterns cannot be described using translation vectors because onset-time differences are allowed.

3.3. *Windowed algorithms*

In practice, we often want to discover repeating patterns that have some restrictions in the onset time differences. For example, a pattern where the onset-time difference between two consecutive notes is one minute is probably not musically interesting. One approach for that is to define a window length w and only consider patterns where $x[a_{i+1}] - x[a_i] \leq w$ and $x[b_{i+1}] - x[b_i] \leq w$ for $i = 1, 2, \ldots, k-1$, i.e. the onset-time difference between any two consecutive notes in both patterns is at most w.

It turns out that we can extend our general pattern discovery algorithm so that it supports a window length and still works in $O(n^2 \log n)$ time. To do that, we need a restricted longest increasing subsequence algorithm which ensures that the x and y difference between two consecutive points is at most w. Figure 5 shows the difference between the general and windowed algorithm in the onset-time-pair representation. When we process the point $(6,5)$ and a window length $w = 2$ is used, we can only extend subsequences where x is between 4 and 6 and y is between 3 and 5.

To support the window length in the y dimension, a small modification to the general algorithm is needed. The general algorithm uses the formula $\texttt{getMax}(1, y[p_i] - 1) + 1$ to calculate the length of an increasing subsequence that ends at point p_i. Since the first parameter in the range query is 1, any y value below $y[p_i]$ is allowed. Thus, to restrict the y difference, we can modify the query so that the first parameter is $\max(1, y[p_i] - w)$ instead of 1. After this change, the algorithm only considers subsequences where the y difference between two consecutive points is at most w.

The remaining task is to also support the window length in the x dimension. This can be done by removing values from the range query structure after the x coordinate of the corresponding point is no longer inside the window. We can create for each y value an additional data structure that contains all `lis` values of points that are inside the current window and have that y coordinate. The data structure must allow efficient insertion and removal of elements and finding the

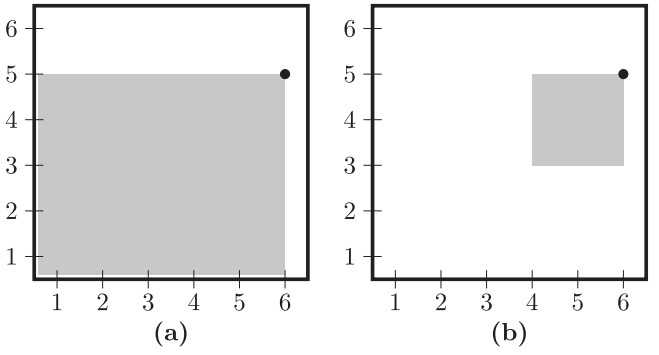

Figure 5. The gray area shows the possible locations of the previous point in a longest increasing subsequence. (a) In the general algorithm, we can choose any point whose x and y coordinates are smaller. (b) In the windowed algorithm, we can only choose points where the x and y difference is at most w. In this example, $w = 2$.

maximum element; we can use a balanced binary search tree that supports all those operations in $O(\log n)$ time. When a point moves outside the window, we first remove the point from the data structure and then update the corresponding value in the range query structure by finding the maximum value for the y coordinate. Since each point is added and removed at most once, the resulting algorithm works in $O(n^2 \log n)$ time like the original algorithm.

Another way to restrict the patterns is to require that the onset-time differences in corresponding pattern positions do not differ too much. This can be done using an additional parameter d and require that $|(x[a_{i+1}] - x[a_i]) - (x[b_{i+1}] - x[b_i])| \leq d$ for $i = 1, 2, \ldots, k-1$, i.e. the onset-time difference is always at most d. For example, if $d = 1$ and the onset-time difference between two notes in the first pattern is 5, then the onset-time difference between the corresponding notes in the second pattern must be between 4 and 6. If $d = 0$, only exact repetitions are allowed.

We can support the d parameter by using two range queries: one in the usual onset-time-pair point set and the other in a *rotated* point set. Figure 6(a) shows the possible point locations using the d parameter, and Figure 6(b,c) represents this area as two rectangles. In both cases, we can use the windowed longest increasing subsequence algorithm to find the repetitions (in the second case we rotate the point set by 45 degrees). Both queries can be processed in $O(\log n)$ time using two range query structures, so the resulting algorithm works in $O(n^2 \log n)$ time.

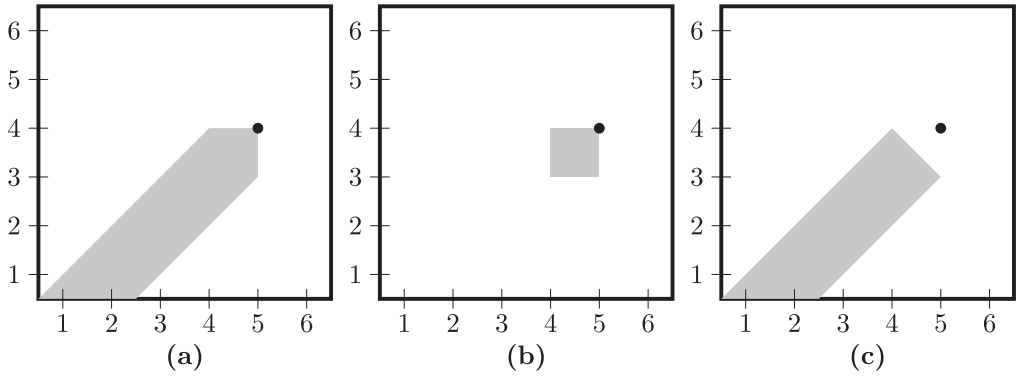

Figure 6. (a) The gray area corresponds to repetitions where the onset-time difference in corresponding pattern positions is at most d (here $d = 1$). (b,c) We can find the repetitions using two windowed queries, one in the original point set and one in a rotated point set.

3.4. *Algorithm implementation*

We have implemented our algorithm in C++, and the source code is available in our GitHub repository (https://github.com/c-brahms/lis-algorithms). The implemented algorithm supports general search without restrictions, exact search and windowed search. The implementation shows that the algorithm works in practice, and it can efficiently process data sets of thousands of notes.

4. Conclusions

In this paper, we have described a new repeating pattern discovery algorithm that can be used to find both exact and time-warped repetitions in $O(n^2 \log n)$ time where n is the number of notes in a musical work. The algorithm is based on an onset-time-pair representation which can be used to reduce the pattern discovery problem to instances of the longest increasing subsequence problem.

In the exact pattern discovery problem, our algorithm can be seen as an alternative (in the two-dimensional case) to the traditional SIA algorithm that also works in $O(n^2 \log n)$ time. The main contribution of our algorithm is that it also solves the more difficult time-warped variant of the problem in $O(n^2 \log n)$ time and supports windows that make the algorithm useful when analyzing real musical inputs.

A general problem in repeating pattern discovery is that many of the detected patterns are usually not musically interesting. This problem can be even more serious in time-warped search because the number of patterns can be large even when the window parameters are well chosen. Thus, in practice, after finding potential repeating patterns, the next step would be to filter musically interesting patterns. In addition, there could be a need for an algorithm like SIATEC that finds all translated occurrences of each maximal translatable pattern. In time-warped search, there are no translation vectors so there cannot be a direct equivalent for SIATEC, but it would still be possible to use a time-warped pattern matching algorithm (Lemström and Laitinen 2011) to find occurrences of each pattern found by the main algorithm.

A future challenge would be to extend the algorithm so that it also supports *time-scaled* repeating pattern discovery. In this variant of the problem, we require that $x[a_{i+1}] - x[a_i] = \alpha|x[b_{i+1}] - x[b_i]|$ for $i = 1, 2, \ldots, k-1$ where α is some constant (each repetition can have an arbitrary α value) which equals the slope of an increasing subsequence (Figure 7 shows

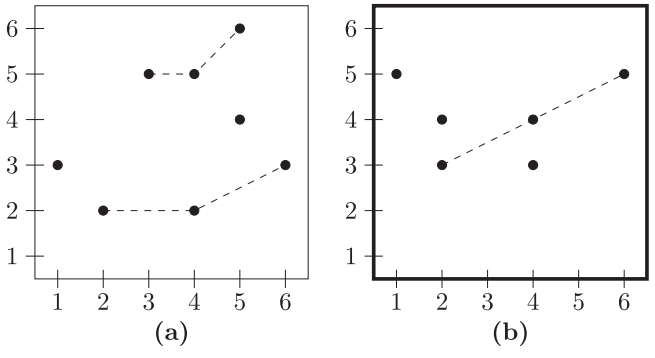

Figure 7. Time-scaled repeating pattern discovery problem. (a) A repeating pattern that consists of note sequences $[(3,5),(4,5),(5,6)]$ and $[(2,2),(4,2),(6,3)]$. (b) The corresponding longest increasing subsequence $[(2,3),(4,5),(5,6)]$ in C_3. The slope of the subsequence equals the scaling parameter $\alpha = 1/2$.

an example where $\alpha = 1/2$). Note that time-scaled and time-warped problems are very different problems from the viewpoint of algorithm design. Time-warped algorithms are usually based on dynamic programming, but this approach is not possible in time-scaled problems and they seem to be more difficult: the best known algorithms for the easier time-scaled pattern matching problem (Lemström 2010) already require quadratic time. Using the longest increasing subsequence technique, we can solve the time-scaled repeating pattern discovery problem in $O(n^4 \log n)$ time by going through all $O(n^2)$ possible α values and performing an individual $O(n^2 \log n)$ search for each of them. However, such an algorithm would not be efficient enough to be used with real musical inputs, so a better approach would be needed to support this invariance.

Disclosure statement

No potential conflict of interest was reported by the author(s).

References

Bent, I., and W. Drabkin. 1987. *Analysis. New Grove Handbooks in Music*. London, UK: Macmillan Press.

Clifford, R., M. Christodoulakis, T. Crawford, D. Meredith, and G. Wiggins. 2006. A Fast, Randomised, Maximal Subset Matching Algorithm for Document-Level Music Retrieval. In *Proceedings of the 7th International Conference on Music Information Retrieval (ISMIR'06)*, 150–155. Victoria, BC, Canada.

de Berg, M., O. Cheong, M. van Kreveld, and M. Overmars. 2010. *Computational Geometry: Algorithms and Applications*. 3rd ed. Berlin, Germany: Springer.

Fredman, M. 1975. "On Computing the Length of Longest Increasing Subsequences." *Discrete Mathematics* 11 (1): 29–35.

Ghias, A., J. Logan, D. Chamberlin, and B. C. Smith. 1995. Query by Humming – Musical Information Retrieval in an Audio Database. In *ACM Multimedia*, 231–236. San Francisco, CA, USA.

Knuth, D. 1973. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Boston, MA, USA: Addison Wesley.

Laaksonen, A. 2017. *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*. Cham, Switzerland: Springer.

Laaksonen, A., and K. Lemström. 2013. On Finding Symbolic Themes Directly From Audio Using Dynamic Programming. In *Proceedings of the 14th International Society for Music Information Retrieval Conference (ISMIR'13)*, 47–52. Curitiba, Brazil.

Lemström, K. 2010. Towards More Robust Geometric Content-Based Music Retrieval. In *Proceedings of the 11th International Society for Music Information Retrieval Conference (ISMIR'10)*, 577–582. Utrecht, Netherlands.

Lemström, K., and M. Laitinen. 2011. Transposition and Time-Warp Invariant Geometric Music Retrieval Algorithms. In *Proceedings of the 2011 International Conference on Multimedia and Expo (ICME'11)*, 1–6. Barcelona, Spain.

Lemström, K., and G. Wiggins. 2009. Formalizing Invariances for Content-Based Music Retrieval. In *Proceedings of the 10th International Society for Music Information Retrieval Conference (ISMIR'09)*, 591–596. Kobe, Japan.

Lerdahl, F., and R. Jackendoff. 1983. *A Generative Theory of Tonal Music*. Cambridge, MA: MIT Press.

Meredith, D., K. Lemström, and G. Wiggins. 2002. "Algorithms for Discovering Repeated Patterns in Multidimensional Representations of Polyphonic Music." *Journal of New Music Research* 31 (4): 321–345.

Mongeau, M., and D. Sankoff. 1990. "Comparison of Musical Sequences." *Computers and the Humanities* 24(3): 161–175.

Romming, C. A., and E. Selfridge-field. 2007. Algorithms for Polyphonic Music Retrieval: The Hausdorff Metric and Geometric Hashing. In *Proceedings of the 8th International Conference on Music Information Retrieval (ISMIR'07)*, 457–462. Vienna, Austria.

Schenker, H. 1954. *Harmony*. London: University of Chicago Press.

Temperley, D. 2001. *The Cognition of Basic Musical Structures*. Cambridge, MA: MIT Press.

Typke, R. 2007. "Music Retrieval based on Melodic Similarity." PhD thesis, Utrecht University, The Netherlands.

Uitdenbogerd, A., and J. Zobel. 2002. Manipulation of Music for Melody Matching. In *Proceedings of the 6th ACM International Conference on Multimedia*, 235–240. Santa Barbara, CA, USA.

Ukkonen, E., K. Lemström, and V. Mäkinen. 2003. Geometric Algorithms for Transposition Invariant Content-Based Music Retrieval. In *Proceedings of the 4th International Conference on Music Information Retrieval (ISMIR'03)*, 193–199. Baltimore, MD, USA.