Transposition and time-warp invariant algorithm for detecting repeated patterns in polyphonic music

Antti Laaksonen and Kjell Lemström

November 2019

Abstract

Finding repetitions in music is a fundamental music information retrieval problem that has several scientific and engineering applications. A popular algorithm for the problem is SIA, the structure induction algorithm developed by Meredith et. al. [10]. SIA is transposition invariant, allows gaps between the notes, and can process both monophonic and polyphonic music. However, the algorithm does not allow any distortion in the time dimension.

In this paper, we introduce a new algorithm that has all SIA's capabilities, but also respects *time-warp invariance*. Such invariance is highly needed, for instance, when there are rhythmic variations in the music, or the input data stems from a live performance. Like SIA, our algorithm works in $O(n^2 \log n)$ time, where n denotes the number of notes, and can efficiently process inputs of thousands of notes using current computers.

1 Introduction

In this paper we consider the problem of finding repetitions in Western, equal tempered, polyphonic music. Repetitions in Western music are frequent and finding them forms an important medium to understand the structure of the music at hand. This makes it easier for listeners to detect musical ideas and remember music [9]. Found repetitions can also be used for genre classification purposes [2].

There has been surprisingly little work on this interesting and important problem. A rather recent survey on symbolic algorithms can be found in [5], and methods based on the audio domain are described in [6, 14]. An apparent advantage gained when using symbolic algorithms is that there is no need to find a sufficient solution to the very challenging problem of fundamental frequency estimation for the polyphonic case. Moreover, as shown in [8], the border between symbolic and audio music and methods is not unbreakable, suggesting that in some cases techniques developed for one domain could be also used directly for the other.

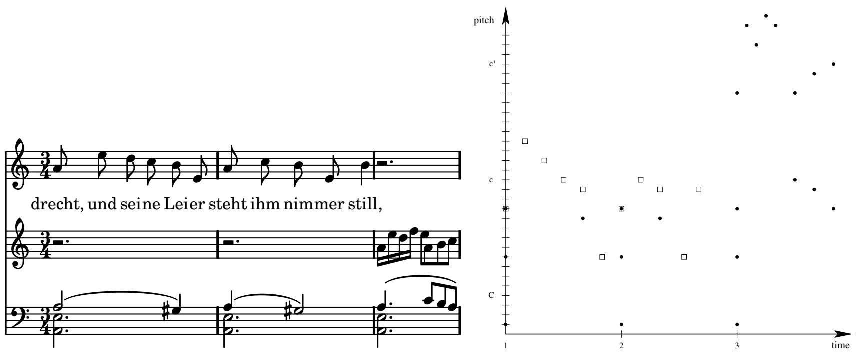


Figure 1: An excerpt from Schubert's *Der Leiermann* and the related geometric, point-set representation. The points associated with the vocal part are represented distinctly (by squares).

Traditionally methods developed for symbolically encoded music information retrieval (MIR) problems have been based on an approximate string matching framework using edit distance (see e.g. [1, 11, 4, 12, 13]). The framework has originally been developed for linear strings, such as natural language, and it is therefore rather straightforwardly applicable to handle melodies (monophonic music). Nevertheless, this approach is incapable to handle polyphonic music with a multitude of simultaneous notes and parallelly developing musical themes. Here, instead, we rely on the piano-roll-like, geometric representation of music, suggested by Meredith et. al. [10], where each note is represented by a point in the plane (Figure 1). For each point, the horizontal location (x-axis) gives its onset time and the vertical location (y-axis) its pitch level.

Given a set of n notes (a musical work), Meredith et. al. considered the problems of discovering all maximal repeated patterns and all occurrences of maximal repeated patterns. They presented the algorithms SIA and SIATEC for computing the solutions in $O(n^2 \log n)$ and $O(n^3)$ time, respectively.

Their algorithms, however, are tolerant to distortion in the time dimension only to a certain extent: if a note is out of time, it is simply discarded. This works rather nicely when only some sporadic notes are distorted. However, if a more systematic distortion has taken place (for instance, a theme has different rhythmic patterns, or the input is a transcription of a live performance), they probably omit the vast majority of the musically meaningful repetitions because they do not find a sufficient count of matching individual notes to form a repetition.

In this paper we introduce an algorithm that works like SIA but can also discover repeated patterns with differences in the time dimension. The algorithm finds for each input note pair the longest repeating pattern whose last note pair is that pair. We also show how the algorithm can be extended

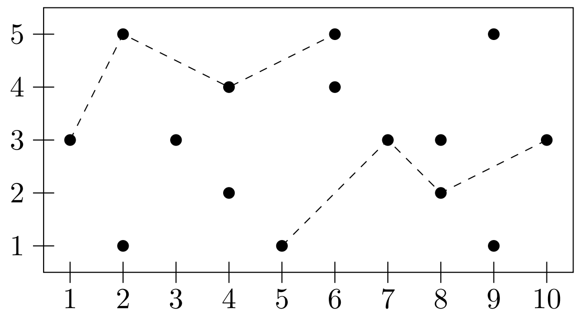


Figure 2: A repeating pattern of four notes. The first pattern consists of notes $(1, 3)$, $(2, 5)$, $(4, 4)$, and $(6, 5)$, and the second pattern consists of notes $(5, 1)$, $(7, 3)$, $(8, 2)$, and $(10, 3)$. The intervals between consecutive notes are the same, but the onset time differences are different.

so that it has an additional parameter w (window size) that restricts the onset differences between consecutive notes. Like SIA, the algorithm works in $O(n^2 \log n)$ time in both cases.

The structure of the rest of the paper is as follows: In Section 2, we define our problem and show how it differs from the previously studied problem. In Section 3, we present our transposition and time-warp invariant algorithm for finding repetitions in symbolic music. In Section 4, we study the actual performance of our new algorithm, and finally, in Section 5, we present our conclusions and outline some ideas for future work.

2 Problem statement

Let S be a set of n notes, numbered $1, 2, \ldots, n$. Each note i is associated two real numbers: an onset time $t(i)$ and a pitch $p(i)$. Geometrically each note can be considered as a two-dimensional point $(t(i), p(i))$.

A *repeating pattern* of length k consists of two note sequences $a_1, a_2, \ldots, a_k$ (first pattern) and $b_1, b_2, \ldots, b_k$ (second pattern) such that $p(a_i) - p(b_i) = c$ for each $i = 1, 2, \ldots, k$, where c is a constant, and $t(a_i) < t(a_{i+1})$ and $t(b_i) < t(b_{i+1})$ for each $i = 1, 2, \ldots, k - 1$. In other words, the interval of each corresponding note pair is c, and the onset times in both sequences are strictly increasing.

For example, Figure 2 shows a repeating pattern that consists of four notes. The intervals between consecutive notes in the patterns match each other, but the onset time differences do not match.

We want to find for each note pair $x, y \in S$ a maximum-length repeating pattern whose last note pair is (x, y). We also consider a variant of the problem with a window size w. In this variant, we require that $t(a_{i+1}) - t(a_i) \le w$ and $t(b_{i+1}) - t(b_i) \le w$ for each $i = 1, 2, \ldots, k - 1$, i.e., the onset time difference between two consecutive notes in a pattern has to be at most w.

Note that the total number of repeating patterns may be very large. An extreme case is a set of notes where each note has the same pitch. In this case, for each $k = 1, 2, \ldots, n$, there are $\binom{n}{k}$ similar patterns. However, by restricting ourselves to maximum-length repeating patterns we only have to find $O(n^2)$ pattern pairs, and it is possible to efficiently solve the problem.

The SIA algorithm [10] solves the problem in $O(n^2 \log n)$ time with an additional constraint $t(a_{i+1}) - t(a_i) = t(b_{i+1}) - t(b_i)$, i.e., no time-warping is allowed. We may use some of the ideas of SIA in our present time-warp invariant problem, but the overall situation is quite different. We will next present an algorithm that solves our problem in $O(n^2 \log n)$ time, both in the unrestricted case and with a window size w.

3 Algorithm description

In this section we describe an efficient algorithm for finding maximum-length time-warp invariant repeating patterns. The algorithm works in $O(n^2 \log n)$ time, and is based on dynamic programming and tree structures.

From now on, we will assume that all onset times are integers between $1 \ldots n$. If this is not the case, we can first sort and relabel them in $O(n \log n)$ time, give each note an onset time between $1 \ldots n$, and make some straightforward modifications to the rest of the algorithm.

The first step of the algorithm is to create a list of all note pairs of the form (x, y) where $x, y \in S$. After this, the list is sorted based on the intervals $p(x) - p(y)$, i.e., all note pairs with the same interval will be next to each other in the list. Since there are $O(n^2)$ note pairs, this phase of the algorithm works in $O(n^2 \log n)$ time.

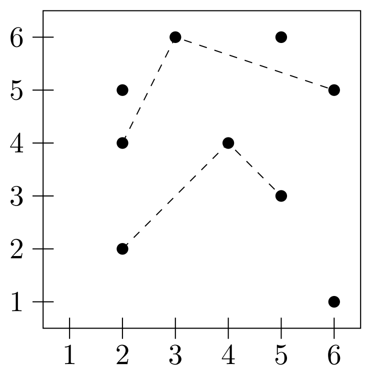
In a repeating pattern, each interval $p(a_i) - p(b_i)$ is a constant c. Thus, we can divide the note pairs into groups according to their intervals and separately process each group. We will next show how we can process each group in $O(m \log n)$ time where m is the number of note pairs in the group. Since the sum of the group sizes is $O(n^2)$, the algorithm works in $O(n^2 \log n)$ time.

3.1 Unrestricted algorithm

We first consider the unrestricted case where the onset time differences between consecutive pattern notes can be arbitrarily large. It turns out that we can reduce the problem to the longest increasing subsequence problem and efficiently solve it using a range query structure.

Since we know that each pair has the same interval, we do not need the pitches anymore and can focus on the onset times. More precisely, our input is a list of m note pairs (x_1, y_1), (x_2, y_2), $\ldots$, (x_m, y_m) where

(a)



(b)

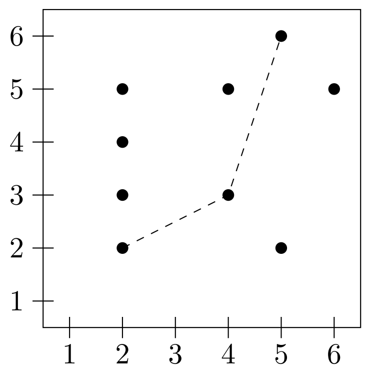


Figure 3: Reduction to the longest increasing subsequence problem. **(a)** The first pattern consists of notes $(2, 2)$, $(4, 4)$ and $(5, 3)$, and the second pattern consists of notes $(2, 4)$, $(3, 6)$ and $(6, 5)$. The interval of each note pair is 2. **(b)** The corresponding increasing sequence consists of onset time pairs $(2, 2)$, $(4, 3)$ and $(5, 6)$.

$p(x_i) - p(y_i) = c$ for each $i = 1, 2, \ldots, m$, and our task is to find for each pair (x_i, y_i) a maximum-length sequence of pairs whose last pair is (x_i, y_i) and both the $t(x)$ and $t(y)$ values are strictly increasing in the sequence.

To remove one dimension from the problem, we sort the pairs lexicographically based on their onset times $t(x)$ and $t(y)$. In such an ordering, the $t(x)$ values are already increasing, and our remaining task is to find for each pair the longest subsequence that ends at that pair and where also the $t(y)$ values are increasing. Geometrically, we have points of the form $(t(x), t(y))$, where both coordinates correspond to onset times of the notes. Figure 3 shows an example of the reduction.

Finding longest increasing subsequences is a classical algorithm design problem [7, 3] that can be efficiently solved using several strategies. To solve the problem so that we can also later support a window size w, we use a

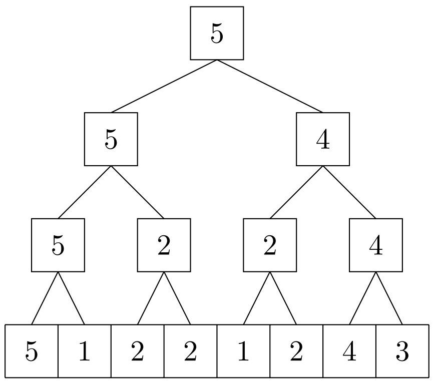


Figure 4: A range query structure that corresponds to the array $[5, 1, 2, 2, 1, 2, 4, 3]$. Each node has the maximum of its children, and the tree has $O(\log n)$ levels. We can both update an array value and determine the maximum value in a range in $O(\log n)$ time.

dynamic range query structure that maintains a sequence $q(1), q(2), \ldots, q(n)$ and has two operations:

1. `setValue`(k, x): set the value of $q(k)$ to x

2. `maxValue`(a, b): return the maximum $q(k)$ value, $k \in [a, b]$

Both the operations can be implemented in $O(\log n)$ time using a balanced binary tree (Figure 4).

We calculate for each $i = 1, 2, \ldots, m$ a value $d(i)$: the length of the longest repeating pattern whose last note pair is (x_i, y_i). We can calculate each such value in $O(\log n)$ time using the range query structure. The idea is that each value $q(k)$ in the structure corresponds to the maximum length of a repeating pattern discovered so far whose last $t(y)$ value is k.

Initially we set $q(k) = 0$ for each $k = 1, 2, \ldots, n$, because we have not discovered any patterns. Then, on each step, we set $d(i) = \texttt{maxValue}(1, t(y_i) - 1) + 1$, because we want to extend a maximum-length repeating pattern whose last $t(y)$ value is less than $t(y_i)$. Finally, if $d(i) > q(t(y_i))$, we also call `setValue`$(t(y_i), d(i))$ to update the range query structure. Since each operation takes $O(\log n)$ time, the total running time of the algorithm is $O(m \log n)$.

Note that if there are several pairs with the same $t(x)$ value, it is not possible to update the range query structure at the end of each iteration, because this could produce a repeating pattern where the $t(x)$ values are not strictly increasing. Instead, we store the updates in a buffer and process the updates always when the $t(x)$ value changes.

In practice, we would also like to trace the notes in the repeating patterns. This can be done by extending the algorithm so that it also maintains

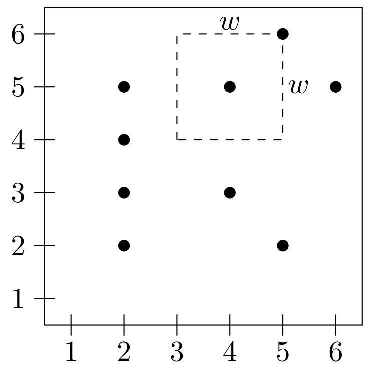


Figure 5: The longest increasing subsequence reduction in the windowed case. There is a $w \times w$ region that must contain the previous pair in the sequence.

for each $i = 1, 2, \ldots, m$ a value $e(i)$: the index of the previous note pair in a longest repeating pattern. We can update those values when calculating the values of $d(i)$.

3.2 Windowed algorithm

In many situations, the unrestricted algorithm can produce musically questionable results, because it allows arbitrarily large gaps between notes. Fortunately, we can modify the algorithm and create a windowed algorithm that only considers patterns where the maximum onset time difference between consecutive pattern notes is w.

Figure 5 shows how the parameter w changes the longest increasing subsequence problem. We no longer accept any previous pair whose $t(x)$ and $t(y)$ values are smaller, but only consider pairs in a $w \times w$ region. In fact, we can easily add this restriction to the $t(y)$ values, because we can just modify the range query. However, a more difficult task is to make sure that the $t(x)$ values also obey the restriction.

The idea is to assign each onset time $k = 1, 2, \ldots, n$ *two* tree structures: a range query structure, like in the unrestricted algorithm, but also an additional multiset structure $s(k)$ that contains all lengths of discovered patterns whose last $t(y)$ value is k and that are currently inside the window. Each multiset structure is implemented as a balanced binary-search tree, so we can both add and remove values in $O(\log n)$ time.

During the execution or the algorithm, the range query structure is maintained so that $q(k)$ always has the largest value in $s(k)$. After calculating a value of $d(i)$, we add it to $s(t(y_i))$ and update the value of $q(t(y_i))$. Furthermore, we maintain a value j that stores the left position of the window. Initially we set $j = 1$. Then, after each round, as long as $t(x_i) - t(x_j) > w$, we remove $d(j)$ from $s(t(y_j))$, update the value of $q(t(y_j))$, and increase the value of j by one.

Since we add and remove every $d(i)$ value at most once and every operation works in $O(\log n)$ time, the windowed algorithm processes m note pairs in $O(m \log n)$ time, like the unrestricted algorithm.

4 Experiment

In this section we study the actual performance of our algorithm. We implemented[1] the algorithm in C++ and carried out experiments using generated test sets of various sizes. We used the following three test sets:

- *Test Set 1*: There are n notes whose onset times are $1, 2, \ldots, n$ and pitches are random integers in the range $[1, 100]$.
- *Test Set 2*: There are n notes whose onset times are random integers in the range $[1, n/10]$ and pitches are random integers in the range $[1, 100]$.
- *Test Set 3*: There are n notes whose onset times are random integers in the range $[1, 10^6]$ and each pitch is constant.

The test sets focus on different aspects of the algorithms. In Test Set 1, each note has a distinct onset time, which simulates monophonic music. Then, in Test Set 2, each onset time is assigned ten notes on average, which simulates polyphonic music. Finally, Test Set 3 represents a special case where the interval of each note pair is the same.

We performed the experiments using a modern laptop computer with an Intel Core i5-7200U CPU. We generated test sets where n is an integer between $[1000, 10000]$, and tested both the unrestricted algorithm and the windowed algorithm using $w = 1000$. Table 1 shows the test results for the unrestricted algorithm, and Table 2 shows the results for the windowed algorithm.

The experiments show that our algorithm has rather small constant factors and can be used to process data sets of several thousands of notes in a few seconds. Even though the windowed algorithm uses more complex data structures, the differences in running times between the unrestricted and windowed algorithm are not substantial. The windowed algorithm, however, is much slower when Test Set 3 is used. A possible reason for this is that since every note pair has the same interval, all of them will be processed in a single pass and the additional set structures used in the windowed algorithm will slow down the algorithm.

[1]Our implementations are available on GitHub, and a link to our repository will be shown here in the final version.

n	Test Set 1	Test Set 2	Test Set 3
1,000	0.25 s	0.26 s	0.62 s
2,000	1.06 s	1.08 s	2.58 s
3,000	2.47 s	2.53 s	6.16 s
4,000	4.46 s	4.55 s	11.16 s
5,000	7.08 s	7.24 s	17.95 s
6,000	10.43 s	10.61 s	26.43 s
7,000	14.27 s	14.58 s	36.10 s
8,000	18.99 s	19.11 s	46.77 s
9,000	24.02 s	24.49 s	60.53 s
10,000	30.04 s	30.29 s	73.91 s

Table 1: Running times of the unrestricted algorithm.

n	Test Set 1	Test Set 2	Test Set 3
1,000	0.29 s	0.32 s	1.95 s
2,000	1.35 s	1.39 s	12.53 s
3,000	3.25 s	3.41 s	33.00 s
4,000	5.94 s	6.35 s	64.24 s
5,000	9.60 s	10.44 s	108.84 s
6,000	14.06 s	15.50 s	160.89 s
7,000	19.50 s	21.62 s	199.33 s
8,000	26.01 s	29.01 s	260.78 s
9,000	33.26 s	38.34 s	288.96 s
10,000	42.28 s	49.41 s	364.32 s

Table 2: Running times of the windowed algorithm.

Since the algorithms process lists that contain all possible note pairs, their memory usage is $O(n^2)$ which could be a problem when n is large. In typical cases, like in Test Sets 1 and 2, we could modify the algorithms so that they separately process each interval and create a list of pairs from scratch in $O(n^2)$ time. This would take more time but require less memory. However, for Test Set 3, such an optimization would not be possible.

Note that we could process Test Set 3 more efficiently using a dedicated algorithm that first greedily finds for each note the longest possible pattern that ends at the note, and then chooses for every note pair the minimum of their corresponding pattern lengths. Such an algorithm would work in $O(n^2)$ time and require only $O(n)$ memory. However, this algorithm only works if each note has the same pitch; we could easily create other difficult test sets where, for example, there are two possible pitches.

5 Conclusions

In this paper we have focused on a central MIR problem of finding repeating patterns in a given polyphonic musical work. In order to effectively deal with polyphonic music and transposition invariance, the geometric framework suggested by Meredith et. al. [10] appears superior over the others suggested in the literature. They presented the SIA algorithm that is transposition invariant and allows gaps. Considering the real-world cases, however, it has a rather major shortcoming: it does not effectively allow distortion in the time dimension.

We presented a time-warp invariant algorithm that works like SIA but also allows distortion in the time dimension. We first considered a case where the onset time differences in patterns are unrestricted, and then showed how the algorithm can support a window size. It turned out that the new problem can be solved without any addition to the computational complexity when compared to that of the SIA algorithm, that is, the algorithm runs in $O(n^2 \log n)$ time.

It is an interesting question whether the $O(n^2 \log n)$ bound could be improved. In both SIA and in our algorithm, the size of the output is $O(n^2)$, so it could be possible to get rid of the $\log n$ factor. In the time-warp invariant problem, we can at least do this in some special cases. As mentioned in Section 4.2, if all notes have the same pitch, the problem can be solved in $O(n^2)$ time, and also if the number of distinct pitches is constant, we can solve the unrestricted problem in $O(n^2)$ time using a greedy choice in dynamic programming. However, it seems difficult to generalize this approach to support arbitrary pitches or a window size.

Our experiments show that our new algorithm is quite efficient also in practice, and could quickly find all repeated patterns, for example, in a ten-minute long classical piano piece. However, processing a full concerto, symphony or opera would take a long time on current computers, and could also force the algorithm to run out of memory. To effectively use the algorithm for real-world repeated pattern detection cases would also require ingenious post-processing that decimates musically meaningless, irrelevant repetitions.

References

[1] R. Clifford, 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 2006)*, 150–155.

[2] A. Ferraro and K. Lemström. 2018. On large-scale genre classification in symbolically encoded music by automatic identification of repeating patterns. In *Proceedings of the 5th International Conference on Digital Libraries for Musicology (DLfM 2018)*, 34–37.

[3] M.L. Fredman. 1975. On computing the length of longest increasing subsequences. *Discrete Mathematics* 11, 1 (1975), 23–35.

[4] A. Ghias, J. Logan, D. Chamberlin, and B.C. Smith. 1995. Query by humming – musical information retrieval in an audio database. In *Proceedings of ACM Multimedia 1995*, 231–236.

[5] B. Janssen, W. Bas de Haas, A. Volk, and P. van Kranenburg. 2013. Finding repeated patterns in music: state of knowledge, challenges, perspectives. In *Proceedings of the 10th International Symposium on Computer Music Modeling and Retrieval (CMMR 2013)*, 277–297.

[6] A. Klapuri. 2010. Pattern induction and matching in music signals. In *Proceedings of the 7th International Symposium on Computer Music Modeling and Retrieval (CMMR 2010)*, 188–204.

[7] D.E. Knuth. 1973. *The Art of Computer Programming. Volume 3: Sorting and Searching.* Addison-Wesley.

[8] A. Laaksonen 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 2013)*, 47–52.

[9] F. Lerdahl and R. Jackendoff. 1983. *A Generative Theory of Tonal Music.* MIT Press, Cambridge.

[10] D. Meredith, K. Lemström, and G.A. Wiggins. 2002. Algorithms for discovering repeated patterns in multidimensional representations of

polyphonic music. *Journal of New Music Research*, 31, 4 (2002), 321–345.

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

[12] R. Typke. 2007. *Music Retrieval based on Melodic Similarity.* PhD thesis, Utrecht University, Netherlands.

[13] A. Uitdenbogerd and J. Zobel. 1998. Manipulation of music for melody matching. In *Proceedings of ACM Multimedia 1998*, 235–240.

[14] R.J. Weiss and J. Bello. 2010. Identifying repeated patterns in music using sparse convolutive non-negative matrix factorization. In *Proceedings of the 11th International Society for Music Information Retrieval Conference (ISMIR 2010)*, 123–128.