SIATEC-C: COMPUTATIONALLY EFFICIENT REPEATED PATTERN DISCOVERY IN POLYPHONIC MUSIC

Otso Björklund
University of Helsinki

ABSTRACT

The use of point-set representations of music enable repeated pattern discovery to be performed on polyphonic music. The discovery of patterns containing polyphony is also enabled by the use of point-set representations. The SIA and SIATEC algorithms discover repeated patterns in point-sets by computing maximal translatable patterns and their translational equivalence classes. While the algorithms are relatively efficient, their application to larger pieces of music is not viable due to quadratic space complexity. This paper introcudes a novel algorithm, SIATEC-C, for repeated pattern discovery in point-set representations of music. The algorithm discovers repeated patterns and finds all of their occurrences, while running with subquadratic space complexity. The algorithm can also provide significant running time improvements over the comparable SIATEC algorithm. The computational performance of the algorithm is compared with SIATEC. The accuracy of the algorithm is also evaluated on the JKU-PDD data set.

1. INTRODUCTION

This paper presents a novel algorithm for computationally efficient repeated pattern discovery in symbolically represented polyphonic music. The main contribution of the algorithm is efficient production of candidate patterns for a post-processing phase where the patterns can be refined, and musically unimportant patterns can be filtered out.

The goal of repeated *pattern discovery* is to find musically important patterns and their occurrences in a piece of music (*intra-opus*) or in a corpus (*inter-opus*). In repeated pattern discovery no query is given by the user unlike in *pattern matching*, where the goal is to find occurrences of a query pattern [1]. Decomposing a piece of music into its constituent elements is a fundamental part of music analysis [2]. Repeated pattern discovery can thus be applied to multiple problems in computational music analysis, such as motivic analysis [3, 4], tune classification [5], segmentation [6], and style analysis [7]. A compressed representation of a piece formed by its repeated patterns can even be considered an analysis of the piece [8, 9].

Repeated pattern discovery is a challenging problem due to certain characteristics of music. Repetition is prevalent in music causing algorithms that focus just on finding repetitions to output large numbers of patterns even for short pieces of music [4, 10]. Often most of the repeated patterns discovered by an algorithm do not correspond to patterns that have musical significance [11]. Instead of relying on repetition alone for discovering musically significant pattern, methods for identifying which patterns are musically important are needed.

The SIATEC-C algorithm presented in this paper improves upon the running time and space complexity of previous point-set algorithms by avoiding the computation of patterns with large temporal gaps and small patterns that are already included in a larger pattern.

2. RELATED WORK

String representations of music have been employed for both pattern matching and discovery in monophonic music. Monophonic music can be represented as a string for pattern matching [12] or a set of strings for pattern discovery [13]. String representations have been often used for mining *closed* patterns, that is, patterns that cannot be extended without reducing the number of occurrences [4, 10]. Monophonic music can also be represented as a pitch signal which enables the use of signal processing methods. Wavelet analysis is used by [14] for repeated pattern discovery in monophonic music.

Polyphonic music can be split into monophonic voices in order to use monophonic pattern discovery methods (e.g., in [13, 15]. Voice separation can be challenging if the voice information is not included, and using monophonic pattern discovery on separate voices cannot be directly used to discover polyphonic patterns or patterns that move from one voice to another, such as *call-and-response* patterns in jazz.

Point-set representations of music enable pattern matching [16] and discovery in polyphonic music [11] with patterns that can be polyphonic. Time-shifts and transpositions of a pattern can be performed by translating the points of the pattern by a vector. The SIA-family of algorithms (see [17] for an overview) discover repeated patterns in music by computing *maximal translatable patterns* (MTP) and finding their occurrences by computing the *translational equivalence classes* (TEC) for the patterns ([11]). The definitions of MTPs and TECs are covered in section 3.

The SIA algorithm by [11] computes all MTPs in a k-dimensional point-set of n points in $O(kn^2 \log n)$ worst-case time and $O(kn^2)$ space, and the SIATEC algorithm computes the TECs of all MTPs in $O(kn^3)$ worst-case time and $O(kn^2)$ space. The worst-case space complexity of computing all MTPs has been improved upon by [18] to $O(kn)$. Another variant of SIA is the SIAR algorithm by [19], which aims to improve the computational performance of MTP computation by restricting the number of difference vectors by a sliding window.

A musical pattern may be *time-warped* and *time-scaled*. [20] present an algorithm that can discover transposed and time-warped repeating patterns in a two-dimensional point-set of n points in $O(n^2 \log n)$ time. The algorithm by [21] can discover transposed and time-scaled repeated patterns in a two-dimensional point-set in $O(n^4 \log n)$ time.

The number of patterns discovered by computing MTPs can often be very large even for small point-sets [11]. Various approaches based on the use of *compression ratio*, *compactness*, and other heuristics have been developed for selecting the musically most important patterns from the set of discovered MTPs. The COSIATEC and SIATEC-Compress algorithms [9, 22] compute a compressed representation of a point-set. The algorithm by [23] similarly aims to compute a representation formed by musically important patterns. [24] further develops the idea of computing a highly compressed representation of a point-set by recursively splitting the input point-set and refining the TECs by removing redundant translation vectors.

MTPs can contain large temporal gaps, such that, the MTP may consist of a musically important pattern and additional points called *isolated members* [19, 25]. [19] proposes a solution to the problem of isolated membership by using an additional processing stage called *compactness trawling* to split MTPs into smaller patterns without isolated members. Post-processing MTPs with compactness trawling in SIA has been found to improve precision and recall [26] over plain SIA. Compactness trawling combined with the symbolic fingerprinting method presented in [27] have been employed in the SIARCT-CFP algorithm to discover inexact occurrences of repeated patterns with high precision and recall [28].

3. BACKGROUND AND DEFINITIONS

In a point-set representation of music, note events are represented as points and a piece of music is represented as a point-set $D \subset \mathbb{R}^k$, where k is the dimensionality of the points. Often $k = 2$, where the first dimension represents the onset time of a note event, and the second dimension represents the pitch of the note event. The rest of this paper assumes that the points are two-dimensional, and $p.x$ denotes the onset time and $p.y$ the pitch of a point p. The number of points in a point-set D is denoted by $|D|$ or n. By using only the onset time of the note events, patterns can be matched based on the *inter-onset-intervals* (IOI) of the adjacent notes. This allows for variations in the durations of the notes. The IOI between consecutive note events p_1 and p_2 is defined as the difference $p_2.x - p_1.x$.

A point-set can be sorted by using a *lexicographical ordering* [11]. A two-dimensional point p_1 is considered to be less than a point p_2, if and only if, $p_1.x < p_2.x$ or $p_1.x = p_2.x$ and $p_1.y < p_2.y$. Lexicographical ordering can be extended to points of any dimensionality. A lexicographically sorted point-set is denoted D_s.

A pattern $P \subset \mathbb{R}^k$ is also a point-set, and P is said to occur in D if $P \subseteq D$. Shifting a pattern P in time and transposing it can be expressed as translation by a vector t, where $t.x$ is the time shift and $t.y$ is the transposition. Translating a pattern P by a vector t is denoted $P + t$ [11]. The points in patterns are assumed to be in ascending lexicographical order in this paper.

The maximal translatable pattern, ***MTP***, of a vector t in a point-set D is defined by [11] as

$$MTP(t, D) = \{d \mid d \in D \wedge d + t \in D\}. \tag{1}$$

The MTP of t in D is formed by the set of all points ind D that can be translated by t so that the translated points are also included in D. Negating the translation t produces the same MTP, that is, $MTP(t, D) = MTP(-t, D)$ [11], therefore only MTPs for translations that are greater than the zero vector are considered. All difference vectors between points in a point-set referred to in this paper are also assumed to be greater than the zero vector. All MTPs in a point-set can be computed with the SIA algorithm [11].

There are at most $n(n-1)/2$ MTPs in a point-set with n points. This occurs when differences between any pair of points are distinct. Conversely, there are at least $n-1$ MTPs in a point-set. The minimum number of MTPs occurs in a point-set where all points are placed on a line with a constant distance between adjacent points. These extreme cases are unlikely to occur in point-sets representing music, however, they are useful for analysis of algorithms that compute MTPs. A point-set with a minimum number of MTPs is denoted by D_{min} and a point-set with a maximum number of MTPs is denoted by D_{max}.

A pattern P may have multiple occurrences in a point-set D. The set of all occurrences of P in D is represented by the *translational equivalence class* (TEC) [11] of the pattern. Two patterns A and B are considered translationally equivalent if, and only if, there exists a translation t such that $A = B + t$. Translational equivalence of A and B is denoted by $A \equiv_{\mathrm{T}} B$. The translational equivalence of patterns can be compared using their *vectorized representations* [11]:

$$VEC(P) = \langle p_2 - p_1, p_3 - p_2, \ldots, p_l - p_{l-1} \rangle, \tag{2}$$

where p_i denotes the ith point of a lexicographically sorted pattern P of length l. Two patterns are translationally equivalent if and only if their vectorized representations are equal [11]. The TEC of a pattern P in a point-set is the set of all subsets that are translationally equivalent to P:

$$TEC(P, D) = \{Q \mid Q \equiv_{\mathrm{T}} P \wedge Q \subseteq D\}. \tag{3}$$

The SIATEC algorithm can be used to compute the TECs of all MTPs in a point-set [11]. The TEC of a pattern P

can be represented by storing one occurrence of the pattern and the translation vectors required to produce all other occurrences of P [11].

4. THE SIATEC-C ALGORITHM

The SIATEC-C algorithm computes TECs of patterns in a point-set D, such that, the IOI between adjacent points in a pattern is at most a given threshold δ. SIATEC-C also avoids producing small patterns when the points covered by the pattern are already covered by a larger discovered pattern. SIATEC-C thus avoids producing patterns with isolated members while reducing the running time and memory footprint. This approach aims at similar results as compactness trawling in [19]. Cutting patterns at large IOI gaps has also been suggested in [29].

The outline of SIATEC-C is described in algorithm 1. The pseudocode aims at a concise presentation of the algorithm. A more thorough pseudocode description and a reference implementation of the algorithm that covers all details is also made available [1]. The algorithm takes as its inputs a point-set D and the IOI thershold δ. The first components of the points are assumed to represent to onset times of note events. The algorithm outputs the discovered patterns and all of their occurrences represented as TECs.

The algorithm begins by sorting the input point-set in ascending lexicographical order to produce the point-set D_s. The variables T and W are used for tracking a sliding window for each point in D_s in the onset dimension. The sliding windows are used in computing MTPs in order to restrict the number of MTPs that need to be kept in memory simultaneously. The array T keeps track of the index from where to continue computation on each iteration, and the array W keeps track of the upper bounds of the windows. The indexing in the pseudocode starts at 1 and array access is denoted by brackets. For T the value at index i stores the index in D_s for where the next sliding window starts. The value at index i of W stores the upper bound of the sliding window for the ith point of D_s. The indices in T are initialized to the range from 1 to n (line 4) and on line 5 the upper bounds are initialized for each point $p \in D_s$ to be $p.x + \delta$. The values in the array C keep track of the size of the largest pattern occurrence that covers the corresponding point in D_s.

The *difference index* structure I is computed by the COMPUTEDIFFINDEX function. Difference vectors between all pairs of points, p_i and p_j, for which the IOI between the points does not exceed the theshold δ, are computed. The differences along with the index-pairs $\langle i, j\rangle$ are stored in the intermediate array I'. The array is sorted in ascending lexicographical order by the difference vectors and indices. The sorted array is partitioned by the difference vectors, so that an array of entries of the form $\langle v, [\langle s_1, t_1\rangle, \ldots, \langle s_i, t_i\rangle]\rangle$ is created. Each entry contains a difference vector v and the corresponding *source* and *target* indices. The source indices s_i are the indices of points in D_s that can be translated by v within D_s, and the corresponding target indices are of the points that are produced by translating the point at the source index by v. The index structure I is sorted in ascending order of difference vectors and all source and target indices for an entry are also in ascending order.

Algorithm 1 SIATEC-C Algorithm

```
1:  function SIATEC-C(D, δ)
2:      D_s ← SORT_Lex(D)
3:      n ← |D_s|
4:      T ← [1, 2, ..., n]
5:      W ← INITWINDOWBOUNDS(D_s, δ)
6:      C ← [0, 0, ... 0] of n zeros
7:      I ← COMPUTEDIFFINDEX(D_s, δ)
8:      while T[1] ≤ n do
9:          M ← COMPUTEMTPS(D_s, T, W)
10:         M′ ← CUTANDSORT(M, δ)
11:         for P ∈ M′ do
12:             if IMPROVESCOVER(P, C) then
13:                 FINDTRANSLATORS(P, I, D_s, C)
14:                 OUTPUTTEC
```

In the main loop of the SIATEC-C algorithm (lines 8–14 of 1), MTPs are computed for translation vectors within the sliding windows by the COMPUTEMTPS function. The MTPs are computed by first computing all translations between pairs of points where the target point is within the sliding window of the source point. The indices of the source points are stored in pairs with the translations. The array thus produced is sorted in ascending lexicographical order and partitioned by the translation vectors. The function is otherwise equal to the SIA algorithm [11], except that the difference vectors are limited by the sliding windows defined by the arrays T and W, and the indices of the MTP and its translated occurrence are also stored. The sliding windows are used to avoid keeping all $O(n^2)$ differences in memory at the same time. On each iteration the indices in T are updated to the point just outside the current window and then the sliding window upper bounds in W are incremented by δ.

The produced MTPs can have gaps in them that exceed the threshold δ. Thus the MTPs are cut on line 10 to produce the set of patterns M', where the IOI between no adjacent patterns points exceeds δ. The patterns are also sorted in descending order of size to ensure that larger patterns are handled first. The function IMPROVESCOVER checks if the pattern, or its translated version, is larger than any of the patterns that cover the same points. A pattern is considered to improve the cover only if it improves the cover value of at least one point. This step reduces the number of small and duplicate patterns that would be otherwise output by the algorithm. Small patterns may be output by the algorithm even if a larger pattern covering the same points is discovered. This occurs in the case that the small pattern is found on an earlier iteration of the main loop (lines 8–14).

[1] `https://github.com/otsob/siatec-c-code`

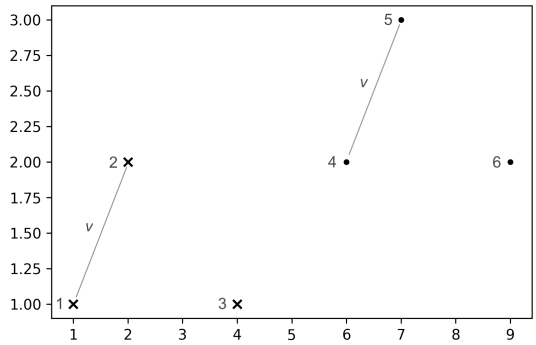
4.1 Finding translators

Finding the translators of a pattern P is achieved by traversing the index-pairs stored in I using the vectors of the vectorized representation $VEC(P)$. This is equivalent to finding translationally equivalent prefixes of P and extending the prefixes until they are equal in length to P.

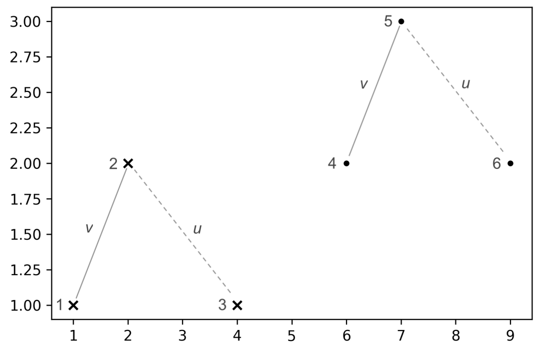
Algorithm 2 SIATEC-C: Find translators and update cover

```
function FindTranslators(P, I, D_s, C)
    V ← VEC(P)
    v ← V[1]
    A ← { t | ⟨s, t⟩ ∈ FindIndices(v, I) }
    for i ∈ [2, ..., |V|] do
        v ← V[i]
        A' ← FindIndices(v, I)
        A ← { t | ⟨s, t⟩ ∈ A' ∧ s ∈ A }
    l ← P[|P|]
    τ ← { D_s[i] − l | i ∈ A }
    C ← UpdateCover(P, A, C, I)
    return τ
```

Figure 1 illustrates the process of finding the translators of a pattern P, $VEC(P) = [v, u]$ with a very minimal point-set example. The crosses and points form two three-point patterns that are translationally equivalent. First binary search is used to find the index pairs for v from I, returning the index pairs $[\langle 1, 2\rangle, \langle 4, 5\rangle]$. The second elements of these pairs are the indices of points that can be translated with u to continue translationally equivalent prefixes of P. On the next iteration the index-pairs associated with u are retrieved producing the index-pairs $[\langle 2, 3\rangle, \langle 5, 6\rangle]$. The target indices 2 and 5 of the vector v are matched with the source indices of u to find that the translationally equivalent prefixes can be extended with the points at indices 3 and 6 to find the last points of translationally equivalent occurrences of P. The translators can be computed simply as the difference between the last points of the found occurrenc es and the last point of P.



(a) Step 1: Prefix of length 2



(b) Step 2: Prefix of length 3

Figure 1: FindTranslators example

4.2 Time and space complexity

The following theorems present the worst case time and space complexity of SIATEC-C.

Theorem 4.1. *Let D be a 2-dimensional point-set with n points. Let m be the largest number of points in any span of length δ in the onset dimension and let h be the number of points in the largest MTP in D. Then the worst case time complexity of* SIATEC-C *is $O(hn^2 \log nm)$.*

Proof. Computing the difference index I requires computing $O(nm)$ difference vectors, sorting them, and partitioning. Thus ComputeDiffIndex runs in $O(nm \log nm)$ time.

The number of iterations the main loop on lines 8–14 of algorithm 1 executes is approximately $\frac{n}{\delta} = O(n)$. Computing the MTPs requires also computing $O(nm)$ difference vectors, sorting and partitioning them into MTPs, and then sorting the MTPs by size, thus running in $O(nm \log nm)$ time. However, the total amount of computation required to compute MTPs in the loop performs the same number of difference vector computations and comparisons as computing all MTPs and sorting them by size, thus the total amount of work needed for MTP computation during the execution of the algorithm is $O(n^2 \log n)$ just as in SIA [11].

For a pattern P, the size of its vectorized represenation is $|P| - 1$. The FindTranslators function is thus run on $O(n^2)$ difference vectors in total. For each difference vector, the loop finds the index pairs from I in $O(\log nm)$ time using binary search and computes the intersection of A with the source indices in A'. The number of index pairs that can be found for a difference vector v in I is equal to the size of the largest MTP in D, denoted by h. Thus computing the intersection of sorted arrays is linear in h, resulting in time complexity of $O(h \log nm)$ for a single difference vector in $\vec{P}$. Overall, finding all translators for all produced patterns has a worst case time complexity of $O(hn^2 \log nm)$.

The overall worst case time complexity of the algorithm is thus dominated by computing the translators, resulting in a worst case time complexity of $O(hn^2 \log nm)$.

□

Theorem 4.2. *Let D be a 2-dimensional point-set with n points, and let m be the largest number of points in any span of length δ in the onset dimension. Then the worst case space complexity of* SIATEC-C *is $O(nm)$.*

Proof. Computing the difference index I requires storing $O(nm)$ difference vectors and corresponding index pairs, and after partitioning the number of difference vectors and index pairs does not increase. Therefore I takes $O(nm)$ space.

On each iteration of the main loop on lines 8–14 of algorithm 1 the MTP computation requires keeping $O(nm)$ difference vectors in memory.

In the FINDTRANSLATORS function the number of index pairs contained in A or A' is at most the equal to the size of the largest MTP in D. Therefore the space complexity of FINDTRANSLATORS is $O(n)$.

The space complexity of SIATEC-C is dominated by I and the MTP computation, therefore the worst case space complexity is $O(nm)$. □

5. RESULTS

This section contains the computational performance results of the SIATEC-C algorithm and its evaluation on the JKU-PDD dataset.

5.1 Computational Performance

The running time and memory usage of the SIA, SIAR ($r = 1$), SIATEC, and SIATEC-C algorithms was measured on two types of point-sets: D_{min} and random patterns. The point-set sizes ranged from 1000 to 10000 in increments of 1000. The D_{min} dataset was chosen as it produces the worst-case running time of SIATEC-C and can thus provide an estimate of the upper bound of running time and memory usage. Random patterns were used instead of concatenating short pieces of music together to control the sizes of MTPs in the benchmark data.

The algorithms were implemented using the Rust [2] programming language. The measurements were performed on a machine running Ubuntu 20.04 with an Intel i7-processor and 16GB of memory. The IOI threshold parameter δ of SIATEC-C was set to a 50.0 for the artificial point-sets, for music point-sets $\delta = 4$ was used throughout, corresponding to one measure in $\frac{4}{4}$ time.

The measurements for running times are plotted in figure 2 and the maximum heap usage measurements are plotted in figure 3. Log-scale is used on the y-axis as the measurements vary greatly in the range of values. The most significant running time improvements SIATEC-C can provide compared to SIATEC can be seen in the plot for the running time on the random pattern point-sets. Even with the largest point-sets, the running time of SIATEC-C is $22.2s$, while the running time of SIATEC exceeds $2500s$. On random pattern point-sets SIAR is the fastest algorithm.

In the case of the D_{min} point-sets the running time of SIATEC and SIATEC-C behaves relatively similarly. With SIAR the D_{min} produces worst-case performance leading the performance of SIAR to be comparable to that of SIA. This is explained by the worst-case time complexity of SIAR, which on a k-dimensional point-set of size n is $O(kn^3)$ [30].

The memory usage was measured using the Heaptrack software [3] that only measures heap memory. With the largest random patterns point-set SIATEC-C uses only $26.08MB$ and with the largest D_{min} point-set $78.20MB$.

On the random patterns point-sets SIAR runs with the smallest memory footprint. However, the D_{min} point-sets illustrates the quadratic space complexity of SIAR [30], with the memory footprint of SIAR exceeding that of SIATEC-C. Thus replacing SIA with SIAR in SIATEC will not guarantee a smaller memory footprint than can be obtained with SIATEC-C.

SIAR can be a very performant algorithm on many point-sets, however, its performance varies greatly depending on the size of the largest MTPs in the input point-set. In order to investigate the impact of the worst-case time and space complexities between SIATEC-C and SIAR, both algorithms were run on a point-set representation [4] of Beethoven's 9th symphony ($n = 107,355$). SIATEC-C ($\delta = 4$) ran in approximately 28 minutes with peak heap usage of $1.97GB$ while SIAR ($r = 1$) ran in approximately 1 hour 7 minutes with peak heap usage $5.64GB$. While SIAR can be the most performant algorithm on small point-sets, due to its worst-case time and space complexity there is no guarantee that it will be the most performant on large point-sets.

5.2 Evalution on JKU-PDD

The accuracy of the SIATEC-C algorithm was evaluated on the JKU-PDD data set [31]. A version of SIATEC-C without any post-processing was evaluated to investigate whether it is capable of achieving establishment precision and recall comparable to other point-set algorithms that have been shown to benefit from post-processing, e.g., compactness trawling [26, 28].

The COSIATEC and SIATECCompress compression algorithms [22] produce a compressed representation of the input point-set by selecting TECs produced by SIATEC. The algorithms COSIATEC-C and SIATEC-CCompress are otherwise equal to COSIATEC and SIATECCompress except they use SIATEC-C instead of SIATEC for producing TECs.

Table 1 displays the mean values of the MIREX metrics over the monophonic and polyphonic corpus of JKU-PDD. Compared to SIATEC and SIAR, SIATEC-C produces fewer patterns and achieves slighlty improved establishment precision and recall.

6. DISCUSSION AND CONCLUSION

In this paper we have presented a novel algorithm SIATEC-C for repeated pattern discovery in symbolic polyphonic

[2] https://www.rust-lang.org

[3] https://github.com/KDE/heaptrack

[4] Converted from `https://musescore.com/openscore/scores/5733014`.

Algorithm	Corpus	N_{points}	$N_{patterns}$	N_{gt}	P_{est}	R_{est}	$F1_{est}$	P_{3L}	R_{3L}	$F1_{3L}$	$P_{occ}(c = 0.75)$	$R_{occ}(c = 0.75)$	$F1_{occ}(c = 0.75)$	$P_{occ}(c = 0.5)$	$R_{occ}(c = 0.5)$	$F1_{occ}(c = 0.5)$
SIATEC	monophonic	677.2	30014.8	6.2	0.128	0.679	0.208	0.072	0.613	0.125	0.681	0.569	0.617	0.459	0.561	0.503
SIATEC-C ($\delta = 4$)	monophonic	677.2	970.0	6.2	**0.189**	**0.890**	**0.308**	**0.131**	**0.852**	**0.227**	**0.842**	**0.854**	**0.844**	**0.558**	**0.824**	**0.649**
SIAR ($r = 1$)	monophonic	677.2	5365.0	6.2	0.148	0.679	0.236	0.091	0.505	0.149	0.685	0.422	0.509	0.496	0.391	0.424
COSIATEC	monophonic	677.2	15.2	6.2	0.136	0.234	0.169	0.085	0.199	0.117	0.165	0.165	0.165	0.256	0.192	0.219
SIATECCompress	monophonic	677.2	10.6	6.2	0.124	0.116	0.114	0.068	0.091	0.075	0.000	0.000	0.000	0.000	0.000	0.000
COSIATEC-C	monophonic	677.2	28.8	6.2	0.090	0.214	0.124	0.088	0.217	0.122	0.000	0.000	0.000	0.120	0.038	0.058
SIATEC-CCompress	monophonic	677.2	21.4	6.2	0.087	0.148	0.109	0.068	0.130	0.088	0.200	0.110	0.142	0.200	0.110	0.142
SIATEC	polyphonic	1289.0	59081.8	5.4	0.105	0.690	0.178	0.066	0.595	0.117	0.677	0.543	0.593	0.499	0.530	0.501
SIATEC-C ($\delta = 4$)	polyphonic	1289.0	977.6	5.4	**0.131**	**0.775**	**0.217**	**0.097**	**0.675**	**0.164**	**0.868**	**0.708**	**0.759**	0.570	**0.645**	**0.577**
SIAR ($r = 1$)	polyphonic	1289.0	12721.4	5.4	0.116	0.635	0.195	0.089	0.483	0.147	0.683	0.476	0.544	**0.588**	0.419	0.477
COSIATEC	polyphonic	1289.0	19.6	5.4	0.091	0.196	0.122	0.056	0.172	0.083	0.157	0.157	0.157	0.290	0.224	0.253
SIATECCompress	polyphonic	1289.0	15.8	5.4	0.103	0.121	0.108	0.059	0.092	0.069	0.000	0.000	0.000	0.000	0.000	0.000
COSIATEC-C	polyphonic	1289.0	41.2	5.4	0.070	0.161	0.095	0.058	0.143	0.081	0.000	0.000	0.000	0.050	0.019	0.027
SIATEC-CCompress	polyphonic	1289.0	24.6	5.4	0.093	0.194	0.122	0.077	0.171	0.102	0.170	0.170	0.170	0.296	0.206	0.226

Table 1: Mean MIREX metrics on JKU-PDD (highest metric values in bold)

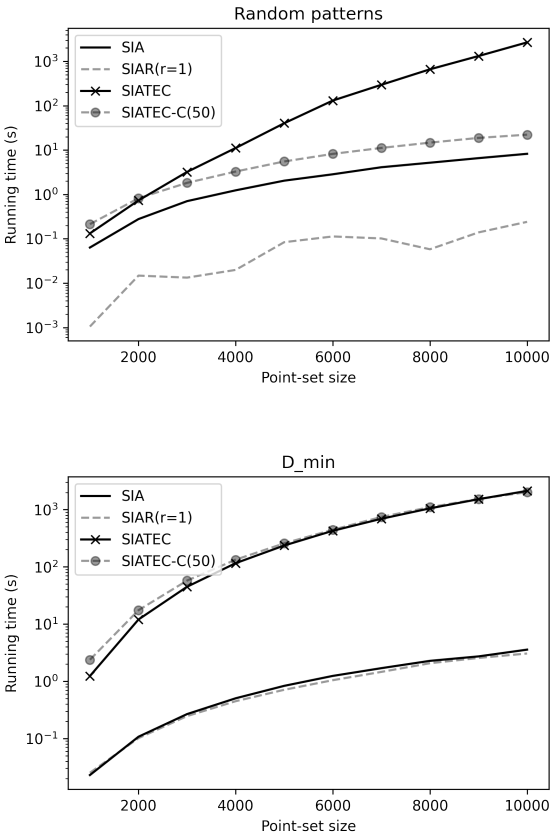


Figure 2: Running times

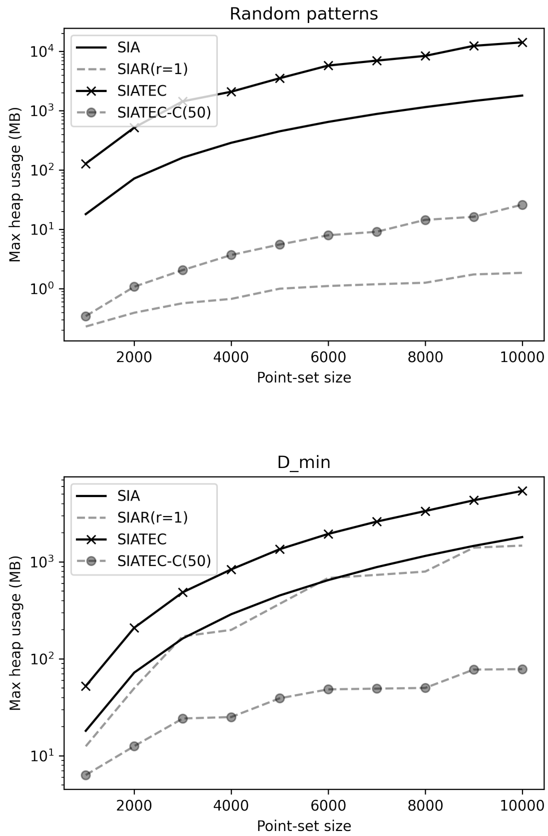


Figure 3: Maximum heap usages

music. The algorithm is based on previous research on repeated pattern discovery in polyphonic music using point-set representations of music [11].

The SIATEC-C algorithm can provide significant running time improvements over SIATEC in discovering patterns and their occurrences when the input consists of patterns that vary in size. In terms of worst-case performance, SIATEC-C does not provide improvements over SIATEC in running time. The most significant improvement SIATEC-C can provide in terms of computational efficiency is its small memory footprint, in which SIATEC-C can outperform SIAR. By keeping the memory usage small, repeated pattern discovery based on point-set representations can be applied to much longer pieces of music than previously.

The simple heuristic of cutting patterns at large IOI gaps in SIATEC-C was found to perform at least as well as the MTP-TEC computation performed by SIATEC in terms of precision and recall. Using SIATEC-C as the TEC algorithm for the compression algorithms COSIATEC and SIATECCompress did not improve their precision or recall. A different approach to filtering and refining the patterns produced by SIATEC-C is thus needed.

The version of SIATEC-C presented in this paper uses the size of patterns as a means of prefiltering. The cover array approach can also be used with other measures that can be computed for a point-set pattern, such as compactness [11, 22]. Evaluating the musical importance of a pattern is a challenging problem. As SIATEC-C also finds all occurrences of the patterns it discovers, the algorithm can be extended with various pattern filtering methods.

7. ACKNOWLEDGEMENTS

This paper has benefited from discussions with Kjell Lemström and Antti Laaksonen, and from email correspondence with Tom Collins. We also thank the anonymous reviewers for their insightful comments on the paper. This work was supported by the Eemil Aaltonen Foundation grant (grant number 220014K).

8. REFERENCES

[1] B. Janssen, W. B. de Haas, A. Volk, and P. van Kranenburg, "Finding repeated patterns in music: State of knowledge, challenges, perspectives," in *Sound, Music, and Motion. CMMR 2013. Lecture Notes in Computer Science, vol 8905*, M. Aramaki, O. Derrien, R. Kronland-Martinet, and S. Ystad, Eds. Springer, Cham, 2013, pp. 277–297.

[2] I. D. Bent and A. Pople, "Analysis," in *Grove Music Online*. Oxford University Press, 2001, accessed: 5 May 2022. [Online]. Available: https://www.oxfordmusiconline.com/grovemusic/view/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-0000041862

[3] O. Lartillot and P. Toiviainen, "Motivic matching strategies for automated pattern extraction," *Musicae Scientiae, Discussion Forum 4A*, pp. 281–314, 2007.

[4] O. Lartillot, "Automated motivic analysis: An exhaustive approach based on closed and cyclic pattern mining in multidimensional parametric spaces," in *Computational Music Analysis*, D. Meredith, Ed. Springer International Publishing, 2016, pp. 273–302.

[5] D. Meredith, "Using point-set compression to classify folk songs," in *The Fourth International Workshop on Folk Music Analysis (FMA 2014)*, Istanbul, Turkey, 2014.

[6] P. Allegraud, L. Bigo, L. Feisthauser, M. Giraud, R. Grould, E. Leguy, and F. Levé, "Learning sonata form structure on mozart's string quartets," *Transactions of the International Society for Music Information Retrieval*, vol. 2, no. 1, pp. 82–96, 2019.

[7] T. Collins, A. Arzt, H. Frostel, and G. Widmer, "Using geometric symbolic fingerprinting to discover distinctive patterns in polyphonic music corpora," in *Computational Music Analysis*, D. Meredith, Ed. Springer International Publishing, 2016, pp. 445–474.

[8] C. Louboutin and D. Meredith, "Using general-purpose compression algorithms for music analysis," *Journal of New Music Research*, vol. 45, no. 1, pp. 1–16, 2016.

[9] D. Meredith, "Analysis by compression: Automatic generation of compact geometric encodings of musical objects," in *The Music Encoding Conference*, Mainz, Germany, 2013.

[10] D. Conklin, "Mining contour sequences for significant closed patterns," *Journal of Mathematics and Music*, vol. 15, no. 2, pp. 112–124, 2021. [Online]. Available: https://doi.org/10.1080/17459737.2021.1903591

[11] D. Meredith, K. Lemström, and G. A. Wiggins, "Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music," *Journal of New Music Research*, vol. 31, no. 4, pp. 321–345, 2002.

[12] K. Lemström, *String Matching Techniques for Music Retrieval*. PhD thesis, University of Helsinki, 2000.

[13] D. Conklin and C. Anagnostopoulou, "Representation and discovery of multiple viewpoint patterns," in *Proceedings of the International Computer Music Conference (ICMC 2001)*, La Habana, Cuba, 2001.

[14] G. Velarde, D. Meredith, and T. Weyde, "A wavelet-based approach to pattern discovery in melodies," in *Computational Music Analysis*, D. Meredith, Ed. Springer International Publishing, 2016, pp. 303–333.

[15] D. Humphreys, K. Sidorov, A. Jones, and D. Marshall, "An investigation of music analysis by the application of grammar-based compressors," *Journal of New Music Research*, vol. 50, no. 4, pp. 312–341, 2021. [Online]. Available: https://doi.org/10.1080/09298215.2021.1978505

[16] E. Ukkonen, K. Lemström, and V. Mäkinen, "Geometric algorithms for transposition invariant content-based music retrieval," in *Proceedings of the 4th International Conference on Music Information Retrieval (ISMIR 2003)*, Baltimore, Maryland, USA, 2003.

[17] D. Meredith, "Analysing music with point-set compression algorithms," in *Computational Music Analysis*, D. Meredith, Ed. Springer International Publishing, 2016, pp. 335–366.

[18] A. Laaksonen and K. Lemström, "On the memory usage of the SIA algorithm family for symbolic music pattern discovery," in *Eighth International Conference on Mathematics and Computation in Music*, 2022, in press.

[19] T. Collins, *Improved methods for pattern discovery in music, with applications in automated stylistic composition*. PhD thesis, The Open University, 2011.

[20] A. Laaksonen and K. Lemström, "Transposition and time-warp invariant algorithm for detecting repeated patterns in polyphonic music," in *6th International Conference on Digital Libraries for Musicology*, 2019.

[21] A. Laaksonen, K. Lemström, and O. Björklund, "Transposition and time-scaling invariant algorithm for detecting repeated patterns in polyphonic music," in *Eighth International Conference on Mathematics and Computation in Music*, 2022, in press.

[22] D. Meredith, "COSIATEC and SIATECCompress: Pattern discovery by geometric compression," in *MIREX 2013. Competition on Discovery of Repeated Themes and Sections*, Curitiba, Brazil, 2013.

[23] J. C. Forth, *Cognitively-motivated geometric methods of pattern discovery and models of similarity in music*. PhD thesis, Department of Computing, Goldsmiths, University of London, 2012.

[24] D. Meredith, "RecurSIA-RRT: Recursive translatable point-set pattern discovery with removal of redundant translators," in *Machine Learning and Knowledge Discovery in Databases*, P. Cellier and K. Driessens, Eds. Cham: Springer International Publishing, 2019, pp. 485–493.

[25] T. Collins and D. Meredith, "Maximal translational equivalence classes of musical patterns in point-set representations," in *Mathematics and Computation in Music: 4th International Conference, MCM 2013, Proceedings. Lecture Notes in Computer Science, Vol. 7937*. Springer, Berlin, 2013, pp. 88–99.

[26] T. Collins, J. Thurlow, R. Laney, A. Willis, and P. H. Garthwaite, "A comparative evaluation of algorithms for discovering translational patterns in baroque keyboard works," in *Proceedings of the 11th International Society for Music Information Retrieval Conference (ISMIR 2010)*, Utrecht, Netherlands, 2010.

[27] A. Arzt, S. Böck, and G. Widmer, "Fast identification of piece and score position via symbolic fingerprinting," in *Proceedings of the 13th International Conference on Music Information Retrieval (ISMIR 2012)*, Porto, Portugal, 2012.

[28] T. Collins, A. Arzt, S. Flossmann, and G. Widmer, "SIARCT-CFP: Improving precision and the discovery of inexact musical patterns in point-set representations," in *Proceedings of the 14th International Society for Music Information Retrieval Conference (ISMIR 2013)*, Curitiba, Brazil, 2013.

[29] G. A. Wiggins, "Models of musical similarity," in *Musicae Scientiae, Discussion Forum 4A*, 2007, pp. 315–338.

[30] O. Björklund, *Improving the running time of repeated pattern discovery in multidimensional representations of music*. Master's Thesis, University of Helsinki, 2018. [Online]. Available: https://ethesis.helsinki.fi/repository/handle/123456789/21114

[31] T. Collins, "Mirex 2013 competition: Discovery of repeated themes and sections," 2013, accessed: 12 April 2022. [Online]. Available: https://www.music-ir.org/mirex/wiki/2013:Discovery_of_Repeated_Themes_\%26_Sections