Improving the running time of repeated pattern discovery in multidimensional representations of music

Otso Björklund

Master's Thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, April 21, 2018

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Otso Björklund

Työn nimi — Arbetets titel — Title

Improving the running time of repeated pattern discovery in multidimensional representations of music

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's Thesis	April 21, 2018	67

Tiivistelmä — Referat — Abstract

Methods for discovering repeated patterns in music are important tools in computational music analysis. Repeated pattern discovery can be used in applications such as song classification and music generation in computational creativity. Multiple approaches to repeated pattern discovery have been developed, but many of the approaches do not work well with polyphonic music, that is, music where multiple notes occur at the same time. Music can be represented as a multidimensional dataset, where notes are represented as multidimensional points. Moving patterns in time and transposing their pitch can be expressed as translation. Multidimensional representations of music enable the use of algorithms that can effectively find repeated patterns in polyphonic music.

The research on methods for repeated pattern discovery in multidimensional representations of music is largely based on the SIA and SIATEC algorithms. Multiple variants of both algorithms have been developed. Most of the variants use SIA or SIATEC directly and then use heuristic functions to identify the musically most important patterns. The variants do not thus typically provide improvements in running time. However, the running time of SIA and SIATEC can be impractical on large inputs.

This thesis focuses on improving the running time of pattern discovery in multidimensional representations of music. The algorithms that are developed in this thesis are based on SIA and SIATEC. Two approaches to improving running time are investigated. The first approach involves the use of hashing, and the second approach is based on using filtering to avoid the computation of unimportant patterns altogether.

Three novel algorithms are presented: SIAH, SIATECH, and SIATECHF. The SIAH and SIATECH algorithms, which use hashing, were found to provide great improvements in running time over the corresponding SIA and SIATEC algorithms. The use of filtering in SIATECHF was not found to significantly improve the running time of repeated pattern discovery.

ACM Computing Classification System (CCS):

- **Theory of computation~Design and analysis of algorithms**
- **Applied computing~Sound and music computing**

Avainsanat — Nyckelord — Keywords
music information retrieval, computational music analysis, repeated pattern discovery

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction **1**

2 Background **2**
2.1 Music terminology . . . 2
2.2 String representations of music . . . 4
2.3 Multidimensional representations of music . . . 4
2.4 Pattern discovery in music . . . 7
2.5 Alternative methods of pattern discovery . . . 8

3 The SIA family of algorithms **11**
3.1 Maximal translatable patterns and translational equivalence classes . . . 11
3.2 The Structure Induction Algorithm SIA . . . 17
3.3 Variants of SIA . . . 19
3.3.1 The SIACT algorithm . . . 19
3.3.2 The SIAR algorithm . . . 20
3.4 The SIATEC algorithm . . . 25
3.5 Variants of SIATEC . . . 28
3.5.1 Heuristic functions . . . 29
3.5.2 The COSIATEC algorithm . . . 30
3.5.3 The SIATECCompress algorithm . . . 32
3.5.4 Forth's algorithm . . . 33

4 Improving the running time of MTP and TEC computation by hashing **35**
4.1 The SIAH algorithm . . . 35
4.1.1 Experiments . . . 38
4.2 The SIATECH algorithm . . . 41
4.2.1 Experiments . . . 48

5 Improving the running time of TEC computation by filtering **51**
5.1 An upper bound on compression ratio . . . 51
5.2 SIATECHF: SIATECH with filtering . . . 53
5.2.1 Experiments . . . 56

6 Conclusions **61**

References **63**

1 Introduction

Identifying important structures in music is a crucial part of music analysis [4]. Important and characteristic structures in a piece of music are often patterns that are repeated throughout the piece. Efficient methods for finding repeated patterns in music can thus be useful for a variety of tasks, such as song classification [37] and computational music generation [10].

The algorithms covered in this thesis operate on symbolic representations of music. Although there are standard ways of representing Western music symbolically, such as music notation [3] and MIDI [51], specialized representations can make pattern finding more efficient. Pattern finding in music is often performed by representing music as strings and using string algorithms. One limitation of the string-based approach is that certain types of musical patterns cannot be effectively found by string algorithms when multiple notes occur simultaneously. Multidimensional representations of music and related algorithms have been developed to overcome this limitation of string representations [41].

When music is represented as a multidimensional dataset, finding repeated patterns becomes a geometric problem. The translation operation can used to express moving a pattern in time and transposing the pitches in the pattern. What sets finding patterns in multidimensional music datasets apart from many problems in computational geometry, is that only the translation operation is of interest. Often in computational geometry, patterns are also rotated, scaled, and mirrored (e.g., in [1, 5, 45, 23]). For example, in [5], the problem of finding maximal repeatable patterns is very similar to the computational problems solved by the algorithms described in Section 3. However, in [5], rotations, scalings, and mirrorings of the pattern are also considered repetitions. In computational geometry the problems concerning patterns often also involve checking if a pattern can be contained within a shape (e.g., in [2]). The problems in computational geometry are typically more complicated than finding translations of patterns. As a result, the algorithms for geometric problems are typically more complicated than is necessary for repeated pattern discovery in music. Developing specialized solutions can therefore lead to faster algorithms.

In previous research on repeated pattern discovery in multidimensional representations of music, the SIA and SIATEC algorithms by Meredith et al. [41] have a central role. Many repeated pattern discovery algorithms that use multidimensional representations are based on them. In [41], it is noted that the running time of the SIA and SIATEC algorithms could be improved, and that heuristics should be developed for identifying the musically most important patterns in the outputs of SIA and SIATEC. The development of algorithms based on SIA and SIATEC has mostly concentrated on the use of heuristic functions to filter out musically unimportant patterns without any improvements in running time being gained. In this thesis the focus is

on improving the running time of repeated pattern discovery by developing algorithms based on SIA and SIATEC. Two research questions are explored in this thesis:

1. Is it possible to develop algorithms that provide the same outputs as SIA and SIATEC but require less time?
2. Is it possible to use filtering by heuristic functions to improve the running time of pattern discovery?

The goal related to the first research question is improving the running time of repeated pattern discovery in general. With the second research question, the aim is to provide an algorithm that outputs musically important patterns and also improves running time by avoiding the computation of unimportant patterns. The algorithms that are developed in this thesis are evaluated by comparing their running times to those of the previously developed algorithms. Comparisons are based on time complexity analyses and empirical running time measurements.

This thesis consists of four main sections. The background on related music terminology, music representations, and pattern discovery in music is covered in Section 2. Previously developed algorithms for repeated pattern discovery in multidimensional representations of music are described in Section 3. Novel algorithms that are based on SIA and SIATEC, and that use hashing to improve the running time of repeated pattern discovery are presented in Section 4. In Section 5, a filtering method is presented that does not require finding all occurrences of a pattern to compute its musical importance. A novel algorithm employing the method is also described in the section.

2 Background

The algorithms covered in this thesis are intended for repeated pattern discovery in music. A brief overview of relevant music terminology is therefore provided. The reader is assumed to be familiar with Western music theory and notation.

2.1 Music terminology

The symbolic representations of music and algorithms covered in this thesis are designed for pattern discovery in music that can be represented using Western music notation (see [3]) or MIDI [51]. The term *score* is used to refer to the notated representation of a piece of music. Music is considered to consist of notes, and each note has at least a pitch and a duration. Notations used with pitchless percussion instruments are not considered in the representations covered in this thesis. The durations of notes are

expressed as fractions of a whole note. For example, instead of using the term *crotchet* for a duration that is one fourth of a whole note the equivalent name *quarter note* is used. The beginning of a note is called its *onset* and the ending of a note its *offset*. The term *pattern* is used to refer to a sequence of notes. Notes in a pattern can also occur at the same time. The formal definition of pattern in the context of pattern discovery in multidimensional representations of music is given in Section 3.1.

It is assumed that the tuning system in use is twelve-tone equal temperament (12TET) (see [47]), in which there are twelve pitches in each octave, and pitch-classes such as $A\flat$ and $G\sharp$ are considered equal. In 12TET pitches can be easily mapped into integers such as MIDI note numbers (see [51]). The integer representation of the pitch of a note in 12TET is called the *chromatic pitch number* of the note. Pitch can also be represented using the *morphetic pitch* number of a note. The morphetic pitch of a note is defined by its position on the staff [41]. For example the pitches $A_4, A\flat_4$, and $A\sharp_4$ have the same value when represented using morphetic pitch because they are all placed in the same position on a staff [34]. If the chromatic pitch number is used in comparing patterns, only chromatic transpositions of a pattern can be found. Using morphetic pitch numbers makes finding certain diatonic transpositions of a pattern easier. If the pitch names are not known, for example, if the input is a MIDI-file, they can be computed using a *pitch spelling algorithm*, such as the one presented in [34]. Transpositions of a note can be expressed by adding a constant to the pitch number of the note. For example, transposing a note chromatically by a major fourth would be expressed by adding 4 to its chromatic pitch number. For the interval between pitches, the term *pitch interval* will be used in this thesis to differentiate it from the general use of the term interval. Pitch intervals can also be expressed as integers, and the pitch interval number between two notes is the difference of the pitch numbers of the notes.

Music can be represented as a sequence of *note events* such as note onsets. Two musical patterns may have the same rhythm even if the notes in the patterns do not have the same durations. This can occur when there are rests between the notes. If the durations of the notes were compared, these rhythms would not be considered the same. However, if the onset times of the notes were compared, the rhythms would be found equal. Using note onsets in comparing rhythms thus makes it easier to match patterns with the same rhythm.

Western music is often *polyphonic*. In the context of this thesis, polyphony means that multiple notes can sound at the same time. Conversely *monophonic* music is such where notes do not occur simultaneously. In this thesis the term *voice* refers to a monophonic sequence that occurs as a part of polyphonic music. For example, a monophonic violin part in orchestral music is a single voice.

2.2 String representations of music

The notes or note events in a score can be represented as a string or a set of strings. String representations allow the use of string algorithms in finding occurrences of musical patterns in a score. The use of string representations will only be briefly explained as the focus of this thesis is on algorithms which take as their input a multidimensional representation of music. A brief overview of string representations of music is given because string representations have been widely used for repetition discovery in music [41], and the purpose of multidimensional representations of music (see Section 2.3) is to overcome some of the limitations of string representations.

In [41], string representations of music are divided into two categories: *event strings* and *interval strings*. In an event string symbols can represent the pitches of note events. The durations of note events can also be represented in an event string. One limitation of event strings is that it is difficult to find transposed occurrences of a pattern in an event string consisting of note event pitches. Using interval strings makes finding such pattern occurrences easier. In an interval string the symbols represent the transformations between consecutive note events [41]. For example, an interval string can represent the pitch intervals between consecutive note events. Finding transposed occurrences of a pattern in an interval string can thus be easier than finding them in an event string.

Polyphony causes many challenges in representing music using strings. Representing simultaneous note events as strings in such a way that string algorithms can still be used to find occurrences of musical patterns is not straightforward. Often in string representations of polyphonic music, the note events are associated with monophonic voices. The string representation of a score then becomes a set of strings with a string for each voice (e.g., in [27], [26], [15]). Finding occurrences of patterns that move from one voice to another in such a representation is not simple. Alternatively, music can be represented as a sequence of simultaneous events in a string representation. In [29], a sequence of polyphonic music was represented as a sequence of sets of integers, i.e., characters in an integer alphabet. A similar approach was also used in the viewpoints representation of music presented in [14] (see also Section 2.5). In addition to the difficulties related to handling polyphony, it is also difficult to find occurrences of patterns that have gaps in them using string algorithms [41].

2.3 Multidimensional representations of music

This section discusses how a piece of music can be represented as a *multidimensional dataset* [41]. The formal definitions related to vectors and point-sets are found in Section 3.1. One of the reasons for using multidimensional representations is to simplify the processing of polyphonic music.

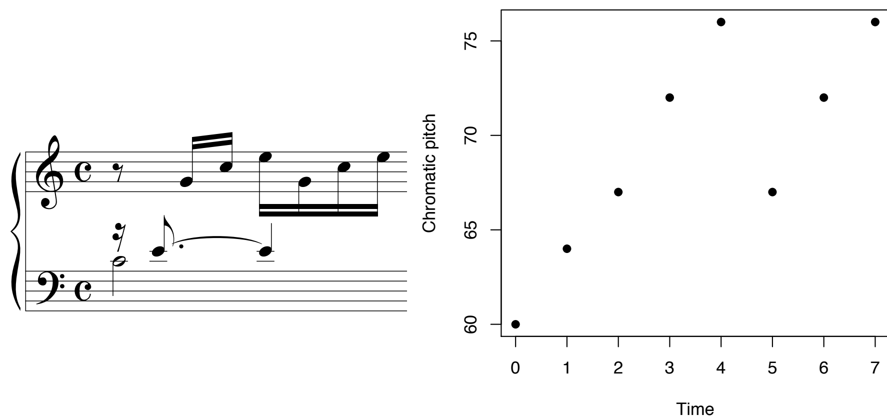
In multidimensional representation of music, note events are represented as k-dimensional vectors (i.e., points), and the score of a piece is represented as a finite subset of $\mathbb{R}^k$. Often the note events being represented are the onsets of notes, and their properties are represented by the components of a vector. The terms vector and point are used interchangeably in this thesis to refer to the elements of a vector space. Note events can also be represented as line-segments represented by a pair of points where the first element in the pair represents the onset of the note and the second point represents the offset (e.g., in [48], [32]).

The first component of a note event vector typically represents the onset time of the note measured from the beginning of the score. Time can be measured by counting sixteenth notes [41] or *tatums*. The tatum for a dataset is the largest common divisor of note duration and onset in the dataset [39]. When the first component of vectors represents time, the vectors can easily be sorted so that their order reflects the temporal order of note events in the music being represented.

The second component of note event vectors is often the pitch number. This can be the chromatic or morphetic pitch number. Both the chromatic and morphetic pitch numbers can also be used as separate components of the vector.

Onset time and pitch are the two basic properties of note events that are represented by the components of a vector. Other properties can be added by increasing the dimensionality of the vectors. For example, in polyphonic music the voice to which a note event belongs can be expressed as an integer and added as a dimension. In a similar way any property of a note event that can be represented as a real number can be added to the vector representation of a note event.

Figure 1 below shows the first two beats of the C-major prelude from J.S. Bach's *Das Wohltemperierte Klavier* (BWV 846). In Figure 1a, the excerpt is shown in standard notation. Figure 1b shows the excerpt as a plot of a 2-dimensional point-set. In Figure 1c, the excerpt is shown as a list of three-dimensional points where the third component represents the staff of the note (0 for top staff and 1 for bottom staff).



(a) Standard notation

(b) 2-d point-set

$$\{(0,60,1),(1,64,1),\\ (2,67,0),(3,72,0),\\ (4,76,0),(5,67,0),\\ (6,72,0),(7,76,0)\}$$

(c) 3-d point-set

Figure 1: Example of a multidimensional representation (BWV 846).

Finding patterns in a multidimensional representation of a score is achieved by translating, i.e. moving, points. If a pattern is repeated in a piece, then there is a vector by which the points in the pattern can be translated to obtain the second occurrence of the pattern. Simultaneously occurring note events simply have the same the same onset time and handling polyphonic music does not require any special considerations. With multidimensional representations there is no need to handle simultaneous events as sets or to divide the music into a set of monophonic voices as is the case with string representations [41].

Multidimensional representations of music can be used for a variety of tasks related to searching musical scores. In [50] and [48], multidimensional representations were used for *content-based music information retrieval*, in which the goal is to find occurrences of a given query pattern in a database of music. This is also called *pattern matching* (see Section 2.4). In [48], an algorithm that uses multidimensional representations for finding both exact and approximate occurrences of a query pattern is presented. In the same study, an algorithm that uses line-segments to find those occurrences that have the most overlap with the query pattern is also presented. Line-segments were also used by Lubiw and Tanur [32] in their approximate pattern matching algorithm. In [6], a randomized algorithm is presented

for approximate matching, in which a subset of the pattern needs to match some points in the dataset.

2.4 Pattern discovery in music

In [25], approaches to pattern finding in music are divided into two categories: *pattern matching* and *pattern discovery*. The goal in pattern matching is to find occurrences of a given pattern in a corpus of music whereas in pattern discovery no input patterns are typically given to the algorithms. The goal of pattern discovery is to find important musical structures in a piece of music or between multiple pieces of music. Pattern discovery within a single piece of music is known as *intra-opus* discovery, and the discovery of common patterns in a corpus is termed *inter-opus* pattern discovery. The algorithms covered in this thesis are mainly intended for intra-opus pattern discovery. Matching and discovery can aim to find exact or approximate occurrences of patterns [25].

Repetition is considered to be an important part of many aspects of music. Meyer [44] considers repeated patterns to play an important role in creating musical style. Many concepts of musical form are based on the repetition of sections. For example, the *sonata form*, which has been used widely in Western classical music since the nineteenth century, is based on repetition of the first part of a movement [46].

The quality of the patterns discovered by a pattern discovery algorithm can be evaluated by comparing the output to reference data, which is often an analysis of important patterns by domain experts [25]. This reference data is used as the *ground truth*.

Repeated pattern discovery is useful for a variety of tasks in computational music analysis. One part of music analysis is dividing the music into smaller elements and investigating the role of those elements [4]. Finding repeated patterns is therefore an important part of music analysis even though it is not necessarily a complete analysis of a piece of music [7].

Lartillot and Toiviainen [27] have applied repeated pattern discovery to the motivic analysis of music. They define motives as "musical structures that constitute one of the most characteristic descriptions of music." [27, p. 281] Repeated pattern discovery can be used for compressing the representation of music. In [31], the compressed representation of a piece of music is considered to be an explanation or an analysis of the piece. The quality of analysis is evaluated by how effectively the analysis compresses the piece. In [35], the idea of music analysis as compression was applied to fugues from the first book of J.S. Bach's *Das Wohltemperierte Klavier*. Using compression by finding repeated patterns as a method for indentifying themes is discussed in [33] and [40]. The repeated patterns discovered in a piece of music can be used to compare it to other pieces for similarity. For example, in [37], repeated pattern discovery and compression were used for classifying folk

tunes. Inter-opus pattern discovery can be used for analyzing the style of a composer and finding occurrences of material re-use by composers (e.g., in [9]).

The set of discovered patterns can be used to model musical style in computational creativity. In [10], using discovered patterns in evaluating the output of a creative musical system is discussed. Repeated pattern discovery is also useful as a preprocessing step for pattern matching. Finding repeated structures is used as a part of the pattern matching algorithm SIA(M) [43, 50].

2.5 Alternative methods of pattern discovery

This section provides a brief overview of selected repeated pattern discovery methods that do not handle music as a multidimensional point-set. These methods have been selected because they are either intended for polyphonic music or they have been presented only recently.

Conklin [14] presents a method that extends the *multiple viewpoint system* to polyphonic music. In the multiple viewpoint system there are multiple independent *views* for each event in a score [16]. Views describe the properties of an event. For a note event these views can represent the pitch, duration, and articulation of the note. Viewpoints can also represent differences between events, such as pitch intervals between notes. In the system there are also events for key and time signature changes. Onset times of events are measured in sixteenth notes and pitch is represented using chromatic pitch numbers and scale degrees. Music is essentially represented as a set of sequences that contains a sequence for each view, such as pitch or duration.

Using the multiple viewpoint system for pattern discovery in monophonic music is discussed in [15]. In the polyphonic extension presented in [14], music is represented as musical objects that can be notes, *simultaneities*, or *sequences*. Simultaneities consist of objects that have the same onset time and sequences consist of objects that do not have any temporal overlap. In order to have simultaneously occuring notes that do not have the same onset time in the same simultaneity, a *full expansion* can be performed. In the fully expanded score, all pitches are duplicated for each unique onset time in the original score. A viewpoint pattern is defined as a fragment that occurs in multiple pieces in a corpus. The significance of a pattern is evaluated by computing its p-value in the corpus. In [14], the p-value of a pattern is defined as the probability that the pattern occurs at least as many times in a randomly selected corpus of the same size. How exactly viewpoint patterns are computed is not described in detail in [14]. It is simply stated that the algorithm uses a suffix tree data structure. No analysis of the time complexity of the algorithm and no empirical measurements of running times are provided. The method was applied to a set of Bach chorales, but no

evaluation of the quality of the discovered patterns is given in [14].

Lartillot's [26] method for repeated pattern discovery uses a representation that is similar to the multiple viewpoints system. Music is represented using parameters that represent properties of note events and parameters that represent differences between consecutive note events (e.g., pitch intervals). Patterns are discovered by using a *pattern prefix tree*. The tree is built in a single pass of the score. New patterns can be detected by using an *associative table* for each parameter. In the associative table of a parameter p, there is an entry for each value v, containing all notes in the score with value v for the parameter p. At the end of the pass, all the discovered patterns are found in the pattern prefix tree.

One challenge in Lartillot's method is that it produces a very large number of patterns [26]. The number of discovered patterns can be reduced by applying a *closure operation* to each possible set of pattern repetitions. The set of patterns is thus assigned a description called its *closed pattern*, which is the least frequently occurring pattern that contains all the other patterns in the set as a subsequence. Another way the number of patterns in the output can be limited, is by detecting *cyclic patterns*. Cyclic patterns occur when a pattern is repeated multiple times successively. Consecutive repetitions of a pattern lead to a large number of redundant patterns. By detecting cyclic patterns it is possible to limit the number of redundant patterns in the output. Lartillot's method was submitted to the 2014 MIREX competition on discovering repeated themes and sections. The method was generally able to discover more of the transformed occurrences of patterns than the other algorithms submitted to the competition [26]. No analysis of the time complexity or empirical measurements of the running time of the method are provided in [26].

The pattern discovery method presented in [49] uses *wavelet analysis* to find repeated patterns in music. Wavelet analysis is a signal analysis method that is used in audio signal processing. In wavelet analysis a *wavelet* is compared with a time-series or signal to calculate the similarity between the wavelet and the different positions of the time-series or signal. In [49], continuous wavelet analysis with the Haar wavelet was used for segmenting music for pattern discovery.

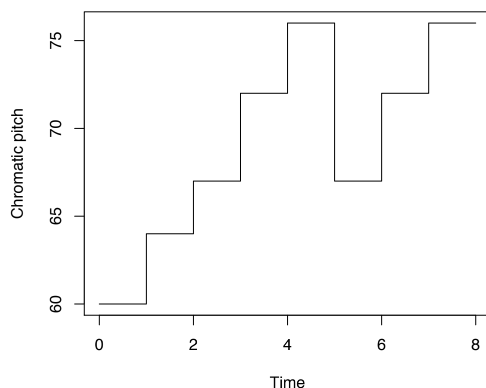


Figure 2: Pitch signal representation of the BWV 846 excerpt.

In order to use signal processing methods with a symbolic representation of music, the chromatic pitch of the music is sampled at regular time intervals to produce a *pitch signal*. Figure 2 above depicts a pitch signal representation of the BWV 846 excerpt. In the figure, pitch has been sampled at a rate of one sample per sixteenth-note. The pitch signal is divided into segments using either constant-duration segmentation or using wavelet analysis. Constant-duration segmentation is performed by dividing the signal into segments of constant size. In addition, the segments are normalized by subtracting the average pitch of the segment from each pitch value in the segment. In the wavelet based segmentation approaches, the wavelet transform of the signal is computed, and then the signal produced by the wavelet transform is segmented at either zero crossings or absolute maxima to produce pattern occurrences.

After segmentation the segments are compared to each other for similarity by using Euclidean distance, city block distance, or dynamic time warping. In addition, similar consecutive segments are concatenated to form *units*. These units are then clustered using their similarity ratings and ranked so that the clusters that have the highest compression ratio are first in the output. The clusters correspond to repeated patterns in the music.

In [49], the quality of the discovered patterns was evaluated using the JKU PDD dataset. Different segmentation and similarity measures were compared and the results were also compared with other pattern discovery methods. No time complexity analysis of the method is given in [49], but running times were measured and compared with the other pattern discovery approaches that were submitted to the 2014 MIREX competition. The methods presented in [49] were found to be the fastest. The second fastest method in the comparison was SIATECCompress (see [36] and Section 3.5.3).

The methods presented in [26] and [49] work with polyphonic music only when the music is partitioned into monophonic voices. Patterns that move

from one voice to another cannot be discovered using these methods. The multiple viewpoints system presented in [14] is similarly unable to discover patterns that move between voices. Even though the methods presented in this section do not directly use string representations of music, they have similar limitations when applied to polyphonic music.

3 The SIA family of algorithms

The SIA family of algorithms consists of SIA, SIATEC, and their variants. The goal of these algorithms is to find occurrences of *maximal translatable patterns* (MTPs) in multidimensional representations of music (see Section 2.3). The term point-set is also used for referring to a multidimensional dataset [39]. Section 3.1 presents the relevant definitions and notations needed to describe the algorithms. The Structure Induction Algorithm (SIA) is presented in Section 3.2, and its variants are presented in Section 3.3. Section 3.4 presents SIATEC and Section 3.5 its variants.

3.1 Maximal translatable patterns and translational equivalence classes

This section provides the formal definitions and mathematical notations used to describe the functions computed by SIA and SIATEC in [41, 39]. Results on the upper and lower bounds for the number of MTPs in a dataset are also presented.

Music is represented as a multidimensional dataset D, which is a finite subset of $\mathbb{R}^k$. A dataset D is a proper set, and all elements in it are thus distinct. The dimensionality k depends on the number of properties that are represented for each note event (see Section 2.3). The number of elements in D is denoted by $|D|$. Unless specified otherwise, n is used to denote the size and k the dimensionality of D. If D is ordered, then the $(i+1)$th element of D is denoted by $D[i]$, where $0 \leq i < |D|$. The elements of a dataset D are k-dimensional vectors, i.e., points. Moreover, array notation with zero-based indexing is used for referring to the elements of sorted tuples and components of vectors. For a vector $v \in \mathbb{R}^k$ and integer i such that $0 \leq i < k$, $v[i]$ is the $(i+1)$th component of v. When an ordered set V consists of tuples or vectors, the array notation can be chained such that $V[i][j]$ denotes the $(j+1)$th element or component of the $(i+1)$th element of V.

A vector $v \in \mathbb{R}^k$ can be *translated* by a vector $t \in \mathbb{R}^k$. This is denoted by $v + t$ and corresponds to vector addition. The *difference vector* d from vector u to v is the vector by which u needs to be translated to obtain v. This is the vector difference $d = v - u$. The worst-case time complexity of addition, subtraction, and equality comparison for k-dimensional vectors is assumed to be $\mathcal{O}(k)$.

Angle brackets "$\langle\cdot\rangle$" are used when an ordered set or tuple is written out as a list of elements. Parentheses "$(\cdot)$" are used when the components of a vector are written out. This notation differs from the notations used in [41] and [39]. The rationale for this distinction is that addition and subtraction are assumed to be defined only for vectors. The " $\oplus$ " operator denotes concatenation of ordered sets.

Sorting datasets and patterns is an integral part of SIA and SIATEC. The ordering that is used for vectors and tuples is *lexicographical ordering*.

Definition 1. Lexicographical ordering [41].
For k-dimensional vectors (or k-tuples) u and v, $u < v$ if and only if there exists an integer i such that $0 \leq i < k$ and $u[i] < v[i]$ and $u[j] = v[j]$ for $0 \leq j < i$. If $u < v$ (respectively $u > v$), then u is said to be *less than* (*greater than*) v.

Vectors that are lexicographically greater than the *zero vector* are called *positive vectors*. The zero vector is denoted by $\bar{0}$. The worst-case time complexity of the lexicographical comparison of k-dimensional vectors is assumed to be $\mathcal{O}(k)$. For k-tuples where comparisons between elements can be performed in constant time, the worst-case time complexity of lexicographical comparison is also assumed to be $\mathcal{O}(k)$. Any ordered set of vectors or tuples that is sorted in ascending lexicographical order is denoted by a subscript s. For example, when the set A is sorted, it is denoted A_s.

A *pattern* P is also a finite subset of $\mathbb{R}^k$. All patterns are assumed to be ordered sets in this thesis. Pattern P can be translated by a vector v, denoted by $P + v$ [39]. This is the set of all points in P translated by v:

$$P + v = \{p + v \mid p \in P\}. \tag{1}$$

A pattern P is *translatable* by a vector v in dataset D if and only if $P + v \subseteq D$ [41]. The *translational equivalence* relation is used in pattern comparisons by many algorithms in this thesis.

Definition 2. Translational equivalence.
Two patterns P and Q are translationally equivalent if and only if there exists a vector v such that $P + v = Q$. This is denoted by $P \equiv_{\mathrm{T}} Q$ [39].

Translational equivalence is an equivalence relation [41]. If a dataset D contains a subset Q that is translationally equivalent to a pattern P, then P is said to occur in D, and Q is an occurrence of P in D. The algorithms of the SIA family of algorithms are used to find occurrences of maximal translatable patterns in a multidimensional dataset.

Definition 3. Maximal Translatable Pattern (MTP) (Eq. 1 of [41]).
The maximal translatable pattern for a vector v in a dataset D, denoted by $MTP(v, D)$, is the largest pattern translatable by v in D. This is formally

expressed as

$$MTP(v, D) = \{d \mid d \in D \wedge d + v \in D\}. \quad (2)$$

For the MTP of a vector v in D to be non-empty, v must occur as a difference vector between at least two points in D [41]. In order to find all MTPs in a dataset D, it is sufficient to compute the MTPs only for difference vectors between points of D. All non-empty MTPs for a dataset D can thus be found by computing the set

$$\mathcal{P}(D) = \{MTP(d_2 - d_1, D) \mid d_1, d_2 \in D\}.$$

The MTP for $\bar{0}$ in D is the whole dataset D. Finding the MTP for $\bar{0}$ is therefore not considered important. It is not even necessary to compute MTPs for all difference vectors between points of D to discover all non-empty MTPs in D. In Lemma 1 of [41], it is stated that for a dataset D and vector v,

$$MTP(v, D) + v = MTP(-v, D).$$

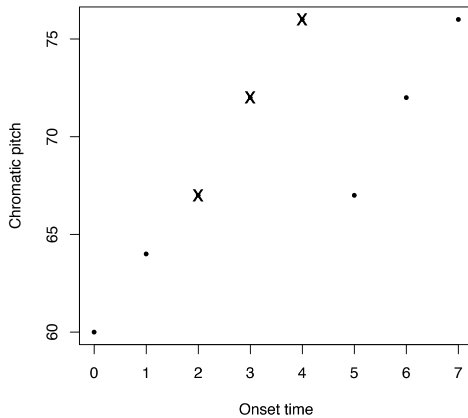
In order to find all MTPs (besides the MTP for $\bar{0}$) in a dataset D, computing the set

$$\mathcal{P}'(D) = \{MTP(d_2 - d_1, D) \mid d_1, d_2 \in D \wedge d_1 < d_2\} \quad (3)$$

is thus sufficient [41]. To retain information about the vector for each MTP, instead of computing $\mathcal{P}'(D)$, the set

$$\mathcal{S}(D) = \{\langle d_2 - d_1, MTP(d_2 - d_1, D)\rangle \mid d_1, d_2 \in D \wedge d_1 < d_2\} \quad (4)$$

can be computed. Each element $\langle v, P\rangle$ in $\mathcal{S}(D)$ is an ordered pair where v is a vector and P is the corresponding MTP in D.



(a) 2-dimensional plot

v	$MTP(v, D)$
$(1, 4)$	$\{(0, 60), (3, 72), (6, 72)\}$
$(1, 5)$	$\{(2, 67), (5, 67)\}$
$(2, 9)$	$\{(2, 67), (5, 67)\}$
(3, 0)	**{(2, 67), (3, 72), (4, 76)}**
$(3, 12)$	$\{(0, 60), (1, 64)\}$
$(6, 12)$	$\{(0, 60), (1, 64)\}$

(b) MTPs with size at least 2

Figure 3: MTPs in the BWV 846 excerpt.

Figure 3 above shows a 2-dimensional representation of the Bach excerpt used in Figure 1. In Figure 3a, the MTP for the vector $(3, 0)$ is displayed with crosses and in Table 3b, the corresponding line is bolded. MTPs of size 1 were omitted from the table.

Meredith et al. [41] consider MTPs to be often associated with musically significant repetitions. The repetitions of a pattern that are discovered by finding MTPs are exact in the sense that the points of a pattern must match exactly when translated. How exact these matches are from a musical point of view depends on the choice of representation. For example, if only the onsets of notes are included in the vectors and durations are excluded, then it is enough that the note onsets match. Thus, variation in note durations is allowed. Multidimensional representations allow there to be additional points between the points of a repeated pattern. *Embellishments* of a musical pattern where notes have been added between the notes of the pattern can therefore be found by discovering MTPs [41]. The *translationally exact once* hypothesis [8] states that if a pattern is repeated in varied form in a piece of music, then some majority subset of the pattern is also repeated exactly at least once in the piece. Assuming the hypothesis is true, finding exactly repeated patterns, such as MTPs, can be used as a part of finding approximate repetitions by clustering similar patterns together.

The total number of points in all MTPs in a dataset D is equal to the number of positive difference vectors between points of D:

$$\sum_{P \in \mathcal{P}'(D)} |P| = \frac{n(n-1)}{2}. \tag{5}$$

There can thus be at most $n(n-1)/2$ non-empty MTPs in a dataset in the case that all MTPs consist of only a single point [41]. The number of MTPs in a dataset also has a lower bound.

Theorem 4. *For a dataset D of size n, the set $\mathcal{P}'(D)$ contains at least $n-1$ MTPs.*

Proof. All points in D are distinct, and thus all difference vectors from any $d \in D$ to lexicographically greater points in D are distinct. There are $n-1$ distinct difference vectors originating from the lexicographically smallest point in D. The set $\mathcal{P}'(D)$ contains an MTP for each positive difference vector occurring between points of D. There are therefore at least $n-1$ MTPs in $\mathcal{P}'(D)$. □

The number of MTPs that can occur in a dataset is thus bounded by

$$n-1 \leq |\mathcal{P}'(D)| \leq \frac{n(n-1)}{2}. \tag{6}$$

The examples in Eq. 7 and Eq. 8 show that it is possible to construct datasets in which the minimum or maximum number of MTPs occur. The

2-dimensional dataset

$$D_{min} = \{(i, c) \mid i = 0, 1, \ldots, n-1\}, \tag{7}$$

where c is a constant and n the size of the dataset, has exactly $n-1$ MTPs. For any $d_1, d_2 \in D_{min}$ such that $d_1 < d_2$, the difference between d_1 and d_2 can be expressed as $d_2 - d_1 = (m, 0)$, where m is an integer such that $1 \leq m \leq n-1$. There are therefore at most $n-1$ distinct positive difference vectors between the points of D_{min}. There are exactly $n-1$ distinct difference vectors originating from the lexicographically smallest point in D_{min}. Therefore $|\mathcal{P}'(D_{min})| = n-1$.

For each MTP to contain only a single point, all positive difference vectors between the points of a dataset must be distinct. This occurs in the 2-dimensional dataset

$$D_{max} = \left\{ \left(i, \sum_{m=0}^{i} \epsilon m \right) \mid i = 0, 1, \ldots, n-1 \right\}, \tag{8}$$

where $\epsilon > 0$ and n is the size of the dataset. The factor ϵ can be used to scale down the values of the second components. Let $a, b, c, d \in D_{max}$ such that $a > b$ and $c > d$. The only solution to the equation $a - b = c - d$ is that $a = c$ and $b = d$, indicating that all positive difference vectors between points of D_{max} are distinct and $|\mathcal{P}'(D_{max})| = n(n-1)/2$.

The sizes and numbers of MTPs that generally occur in datasets of various styles of music may not be close to the bounds in Eq. 6. The datasets D_{min} and D_{max} are of theoretical interest and are used for investigating the effect of MTP counts and sizes on the running time of algorithms in Sections 4.1.1 and 4.2.1.

From the definition of MTP (Definition 3), it is clear that any MTP in $\mathcal{S}(D)$ must occur at least twice in D. There can be more occurrences of an MTP than the two found by computing $\mathcal{S}(D)$. All occurrences of a pattern P in D can be found by computing the *translational equivalence class* of P in D.

Definition 5. Translational Equivalence Class (Eq. 10 of [41])
For pattern P in D, the translational equivalence class (TEC) of P in D is the set

$$TEC(P, D) = \{Q \mid Q \equiv_{\mathrm{T}} P \wedge Q \subseteq D\}. \tag{9}$$

All occurrences of all MTPs in D can be found by computing the set

$$\mathcal{T}(D) = \{TEC(MTP(d_2 - d_1, D), D) \mid d_1, d_2 \in D \wedge d_1 < d_2\}. \tag{10}$$

The TEC for pattern P in D can be represented as a pair $\langle P, T(P, D) \rangle$, where the first element is the pattern and the second element $T(P, D)$ is the set of

translators for P in D [41]. A translator t for P in D is a vector by which P is translatable in D. The set of translators for P in D is the set

$$T(P, D) = \{t \mid P + t \subseteq D\}. \tag{11}$$

The set of patterns that are translationally equivalent to P in D can be obtained by simply translating P by each $t \in T(P, D)$.

By using the above representation of a TEC, the set of all TECs in D can be represented by the set

$$\mathcal{T}'(D) = \{\langle MTP(d_2 - d_1, D), T(MTP(d_2 - d_1, D), D)\rangle | d_1, d_2 \in D \wedge d_1 < d_2\}. \tag{12}$$

Table 1 below shows the TECs for all MTPs in the Bach excerpt of Figure 1. Each TEC is represented by a pattern and its set of translators. The table illustrates three aspects of MTP TECs:

1. A pattern consisting of a single point p has all the differences between p and the points in the dataset as its translators. The TEC of a single-point pattern is not particularly interesting because regardless of which point in the dataset is selected as the pattern, the pattern will have the same set of occurrences in the dataset. The pattern will also cover the entire dataset.

2. There are 6 MTPs of size greater than 1 in the Bach excerpt (see Figure 3b), but there are only 4 TECs in Table 1 with a pattern larger than a single point. This is because some of the MTPs are translationally equivalent, and therefore they belong to the same TEC. There can be fewer TECs of MTPs in a dataset than there are MTPs.

3. The zero vector is present in each pattern's set of translators. If a pattern is a subset of a dataset D, then the pattern is always translatable by the zero vector in D, making the zero vector in the set of translators redundant. To further compress the representation of TECs, it is possible to omit the zero vector from the set of translators [38]. In this thesis $\bar{0}$ is always assumed to be included in the set of translators of a TEC.

MTP	Translators
$\{(4, 76)\}$	$\{(-4, -16), (-3, -12), (-2, -9), (-1, -4), (0, 0), (1, -9), (2, -4), (3, 0)\}$
$\{(0, 60), (1, 64)\}$	$\{(0, 0), (3, 12), (6, 12)\}$
$\{(2, 67), (5, 67)\}$	$\{(0, 0), (1, 5), (2, 9)\}$
$\{(2, 67), (3, 72), (4, 76)\}$	$\{(0, 0), (3, 0)\}$
$\{(0, 60), (3, 72), (6, 72)\}$	$\{(0, 0), (1, 4)\}$

Table 1: MTP TECs in the BWV 846 excerpt.

3.2 The Structure Induction Algorithm SIA

The *Structure Induction Algorithm* SIA by Meredith et al. [41] computes the set $\mathcal{S}(D)$ (Eq. 4) of all MTPs in a multidimensional dataset D.

SIA is depicted in Algorithm 1. First, the dataset D is sorted in ascending lexicographical order to produce the sorted set D_s (line 2). This step can be performed in $\mathcal{O}(kn \log n)$ time using a modified merge-sort [41].

Algorithm 1 The Structure Induction Algorithm for computing the set of MTPs in a dataset (Figure 13.5 of [39]).

```
 1: function SIA(D)
 2:     D_s ← Sort_Lex(D)
 3:     V ← ⟨⟩
 4:     for i ← 0 to |D_s| − 2 do                    ▷ Compute difference vectors
 5:         for j ← i + 1 to |D_s| − 1 do
 6:             V ← V ⊕ ⟨⟨D_s[j] − D_s[i], i⟩⟩
 7:     V_s ← Sort_Lex(V)
 8:     M ← ⟨⟩
 9:     v ← V_s[0][0]
10:     P ← ⟨D_s[V_s[0][1]]⟩
11:     for i ← 1 to |V_s| − 1 do                    ▷ Partition V_s
12:         if V_s[i][0] = v then
13:             P ← P ⊕ ⟨D_s[V_s[i][1]]⟩
14:         else
15:             M ← M ⊕ ⟨⟨v, P⟩⟩
16:             v ← V_s[i][0]
17:             P ← ⟨D_s[V_s[i][1]]⟩
18:     M ← M ⊕ ⟨⟨v, P⟩⟩
19:     return M
```

On lines 3–6 the set

$$V = \{\langle D_s[j] - D_s[i], i\rangle \mid 0 \leq i < j < |D_s|\} \tag{13}$$

is computed. The elements of V are ordered pairs where the first element is the difference vector from the $(i+1)$th element to the $(j+1)$th element of D_s. The second element is the index of the vector from which the difference is computed. This is called the *origin index* of the difference vector. V can also be thought of as a vector table, as depicted in Table 2 below for the BWV 846 excerpt. As it is only necessary to compute the positive difference vectors (see Eq. 3) to find all MTPs, the set of difference vectors in V is sufficient for finding all MTPs in D. In the vector table this means that only values on the subdiagonals (or superdiagonals if the placement of the *from* and *to* vectors is switched) are computed. The total number of k-dimensional

difference vectors in V is $n(n-1)/2$. The space required by V is thus $\mathcal{O}(kn^2)$, and V can be computed in worst-case $\mathcal{O}(kn^2)$ time [41].

To \ **From**	$(0,60)$	$(1,64)$	$(2,67)$	$(3,72)$	$(4,76)$	$(5,67)$	$(6,72)$	$(7,76)$
$(0,60)$								
$(1,64)$	$\langle(1,4),0\rangle$							
$(2,67)$	$\langle(2,7),0\rangle$	$\langle(1,3),1\rangle$						
$(3,72)$	$\langle(3,12),0\rangle$	$\langle(2,8),1\rangle$	$\langle(1,5),2\rangle$					
$(4,76)$	$\langle(4,16),0\rangle$	$\langle(3,12),1\rangle$	$\langle(2,9),2\rangle$	$\langle(1,4),3\rangle$				
$(5,67)$	$\langle(5,7),0\rangle$	$\langle(4,3),1\rangle$	$\langle(3,0),2\rangle$	$\langle(2,-5),3\rangle$	$\langle(1,-9),4\rangle$			
$(6,72)$	$\langle(6,12),0\rangle$	$\langle(5,8),1\rangle$	$\langle(4,5),2\rangle$	$\langle(3,0),3\rangle$	$\langle(2,-4),4\rangle$	$\langle(1,5),5\rangle$		
$(7,76)$	$\langle(7,16),0\rangle$	$\langle(6,12),1\rangle$	$\langle(5,9),2\rangle$	$\langle(4,4),3\rangle$	$\langle(3,0),4\rangle$	$\langle(2,9),5\rangle$	$\langle(1,4),6\rangle$	

Table 2: Vector table V for the BWV 846 excerpt.

If there is a pair $\langle v, i\rangle \in V$, then the point $D_s[i]$ is translatable by v in D. All points translatable by vector v in D can be found by finding all pairs in V that have v as their first element. Therefore

$$MTP(v, D) = \{D_s[p[1]] \mid p \in V \wedge p[0] = v\}.$$

All MTPs in D can be found by partitioning V so that the pairs with the same vector as their first element are placed in the same partition. The second elements (the origin indices) of pairs in a partition are then used to find the MTP points from D_s.

Partitioning V is performed by first sorting V to produce the ordered set V_s (line 7). Using a modified version of merge-sort, the sorting of V can be accomplished in worst-case $\mathcal{O}(kn^2 \log n)$ time [41]. The lexicographical ordering of V_s ensures that pairs with the same vector as their first element are placed in consecutive indices.

On lines 8–18 the set V_s is partitioned. The vector-MTP pairs are collected into the set M by iterating through V_s. Vector v is used for keeping track of the difference vector and the set P is used for collecting the MTP points. As long as the first element of the pair at the current index is equal to v (line 12), the points at the origin indices in D_s belong to the same MTP. When a pair is found that does not have v as its first element, then all points of the MTP for v in D have been collected in P. The pair $\langle v, P\rangle$ is added to M, and v and P are updated for collecting the next MTP (lines 15–17). After the loop of lines 11–17 is finished, the last MTP is added to M on line 18 so that $M = \mathcal{S}(D)$.

Finding the MTPs from V_s takes $\mathcal{O}(kn^2)$ time in the worst case. The time complexity of SIA is dominated by sorting V on line 7, resulting in an overall worst-case time complexity of $\mathcal{O}(kn^2 \log n)$. The space complexity of SIA is $\mathcal{O}(kn^2)$, which results from the size of V [41].

3.3 Variants of SIA

SIA outputs at most $n(n-1)/2$ MTPs for a dataset with n points [41]. Considering that the number of subsets in such a dataset is 2^n, the output of SIA leaves out a great number of possible patterns. However, the set of discovered patterns can still be very large and contain a great number of patterns that are not musically important [41]. In this thesis the *quality* of a pattern is used to refer to the musical significance or importance of the pattern. Two variants of SIA by Collins [7] are presented: SIACT (Section 3.3.1) aims to improve the quality of discovered patterns and SIAR (Section 3.3.2) aims to both improve the quality of patterns and decrease running time by limiting the size of the output.

3.3.1 The SIACT algorithm

The goal of SIACT is to solve the *problem of isolated membership* [13, 7, 12]. The problem of isolated membership occurs when an MTP P contains a subpattern $P' \subset P$ that is musically more important than P. The MTP P thus consists of an important pattern P' and *temporally isolated members*. There can also be multiple subpatterns in an MTP that are musically more important than the MTP itself.

SIACT begins by computing the set of all MTPs in a dataset D by using SIA. The output of SIA is processed by applying a *compactness trawler* procedure to obtain patterns without isolated members. The compactness of a pattern P that occurs in D is defined as the ratio of points belonging to P and the number of points in D that are in the *region* of P [42]. In [7], the region of P in D is defined as the set of points in D that are lexicographically between the first and last points of P_s, the lexicographically sorted version of P. For a pattern P its compactness in D is defined by

$$c(P, D) = \frac{|P|}{|\{d \in D \mid P_s[0] \leq d \leq P_s[|P|-1]\}|}. \tag{14}$$

For a pattern in a k-dimensional dataset D of size n, its compactness in D can be computed in worst-case $\mathcal{O}(kn)$ time, and the complexity can be reduced if D is sorted lexicographically [7].

SIACT takes two parameters: the compactness threshold $0 < a \leq 1$ and the cardinality threshold $b \geq 1$. After all MTPs in the input dataset D have been computed using SIA, the compactness trawler procedure is run on each MTP. The output of SIACT is the set of patterns produced by performing the compactness trawler procedure on each MTP. The compactness trawler algorithm presented in [7] is shown in Algorithm 2. The CompactnessTrawl function returns all contiguous subpatterns of P_s that have compactness of at least a in D and are of at least size b. Points that do not belong to any pattern in the set X returned by CompactnessTrawl are considered isolated members and are discarded.

Algorithm 2 The compactness trawler procedure used in SIACT.

```
1: function CompactnessTrawl(P_s, D, a, b)
2:     X ← ⟨⟩
3:     i ← 0
4:     for j ← 0 to |P_s| − 1 do
5:         Q ← ⟨P_s[i], ..., P_s[j]⟩
6:         Q′ ← Q
7:         if j < |P_s| − 1 then
8:             Q′ ← Q′ ⊕ ⟨P_s[j + 1]⟩
9:         if c(Q′, D) < a then
10:            if |Q| ≥ b then
11:                X ← X ⊕ Q
12:            i ← j + 1
13:    return X
```

According to Collins [7], performing compactness trawling takes $\mathcal{O}(kn)$ time, but Collins does not provide an analysis of this and does not make clear what the time complexity of computing compactness (line 9 of Algorithm 2) is assumed to be. As SIACT uses SIA, it is clear that the time and space complexities of SIACT are at least those of SIA.

In [13], a comparative evaluation of SIACT, SIA, and COSIATEC (see Section 3.5.2) was conducted. The algorithms were applied to the discovery of patterns within pieces of Baroque keyboard music. SIACT was able to discover more of the musically salient patterns present in the ground truth analyses than SIA and COSIATEC.

Meredith [39] has criticized the compactness trawler in SIACT for the fact that the output depends on the order in which the pattern is scanned. The compactness trawler procedure scans patterns from the lexicographically smallest point to the largest. If the order is reversed, the procedure can output a different set of patterns. According to Meredith [39], the musical and psychological basis for a pattern selection procedure whose output depends on scanning order is questionable.

3.3.2 The SIAR algorithm

SIAR aims to improve upon the running time and pattern quality of SIA by computing only a subset of the MTPs in the input dataset [7]. Instead of computing all subdiagonals of the vector table V (see Table 2), as is done in SIA, SIAR computes only r subdiagonals of V. In [7], the difference vectors are placed on the superdiagonals, but in this thesis, the difference vectors are always placed on the subdiagonals for consistency. Collins [7] considers SIAR to be similar to using a sliding window of size r in SIA. SIAR can be used instead of SIA in SIACT [8]. No detailed description of the implementation

of SIAR is provided in [7]. The description of SIAR in Algorithm 3 is based on Figure 13.14 of [39].

Algorithm 3 Structure Induction Algorithm for r subdiagonals.

```
 1: function SIAR(D, r)
 2:     D_s ← Sort_Lex(D)
 3:     V ← ⟨⟨D_s[j] − D_s[i], i⟩ | 0 ≤ i < j < |D_s| ∧ j ≤ i + r⟩
 4:     V_s ← Sort_Lex(V)
 5:     E ← ⟨patterns obtained by partitioning V_s by first elements⟩
 6:     L ← ⟨e[j] − e[i] | e ∈ E, 0 ≤ i < j < |e|⟩
 7:     L_s ← Sort_Lex(L)
 8:     v ← L_s[0]
 9:     f ← 1
10:     M ← ⟨⟩
11:     for i ← 1 to |L_s| − 1 do
12:         if L_s[i] = v then
13:             f ← f + 1
14:         else
15:             M ← M ⊕ ⟨⟨v, f⟩⟩, f ← 1, v ← L_s[i]
16:     M ← M ⊕ ⟨⟨v, f⟩⟩
17:     M ← SortDescendingByFrequency(M)
18:     S ← ⟨⟩
19:     for i ← 0 to |M| − 1 do                          ▷ Compute MTPs
20:         S ← S ⊕ ⟨D_s ∩ (D_s − M[i][0])⟩
21:     return S
```

SIAR starts by sorting the dataset D. On line 3 the r subdiagonals of the difference vector table V are computed. This can be performed exactly as in SIA (lines 4–6 of Algorithm 1) but with the added limitation that $j \leq i + r$. On line 4 V is sorted to obtain V_s which is partitioned into translatable patterns based on the first elements, i.e., the difference vectors. The translatable patterns are stored in the ordered set E (line 5). Lines 2–5 of SIAR are implemented just like in SIA (see Algorithm 1), with the exceptions that all subdiagonals are not computed and that the set E does not contain the translators related to the patterns.

On line 6 all *intrapattern differences* are computed and stored in the ordered multiset L. Intrapattern differences are positive difference vectors that occur between points belonging to the same pattern. This is roughly equal to running SIA so that each of the patterns in E is used as an input dataset but without computing the MTPs [39]. The ordered set M is computed next. M contains all the distinct vectors in L in descending order of frequency (the number of occurrences). M is computed by first sorting L (line 7) and then computing the frequencies of vectors in L by iterating

through L_s and adding the vector-frequency pairs to M (lines 11–16). M is then sorted in descending order based on the frequencies collected in the loop (line 17).

The MTP for vector v in dataset D can be found by computing the intersection of D and D translated by $-v$ [39]. This method is used in the loop on line 20 to find the MTP in D for each vector in M. The MTPs are collected in the set S, which is returned by the algorithm.

No time or space complexity analysis of SIAR is provided in [7]. The implementation of SIAR presented in [39] is likewise provided without any complexity analysis. Collins states in [7] that the initial results on using SIAR were found positive in regard to both running time and quality of patterns, but further research is required. In the study of [39], using SIAR instead of SIA as a part of COSIATEC did not improve the quality of the output when quality was measured using compression ratio.

One of the main goals of SIAR is to improve upon the running time of SIA. An analysis of the worst-case time and space complexity of the implementation of SIAR presented in [39] (see Algorithm 3 above) is presented here. The time and space complexity of SIAR depends on the sizes of the sets L and M. Lemmas 6 and 7 provide asymptotic upper bounds for their sizes.

Lemma 6. *Let D be a dataset of size n and r the number of subdiagonals computed in the difference vector table. Then $|L| = \mathcal{O}(rn^2)$.*

Proof. The size of L is equal to the total number of intrapattern differences:

$$|L| = \sum_{e \in E} \frac{|e|(|e|-1)}{2}. \tag{15}$$

The total number of points in the patterns in E is equal to the number of difference vectors on the r subdiagonals of the difference vector table V. For a given r and n, the sizes of the patterns and the size of L thus depend on the number of patterns in E. Suppose that splitting a pattern of size p into two smaller patterns of sizes $p - m$ and m, where $p > m$, increases the size of L. Based on Eq. 15, it would follow that

$$\begin{aligned}
&\frac{p(p-1)}{2} < \frac{(p-m)(p-m-1)}{2} + \frac{m(m-1)}{2} \\
&\Rightarrow p^2 - p < p^2 - p - 2pm + 2m^2 \\
&\Rightarrow pm < m^2 \\
&\Rightarrow p < m,
\end{aligned}$$

which contradicts $p > m$. Therefore, the size of L cannot increase as the number of patterns increases, and L is largest when there is the least number of patterns in E.

There are at least r patterns in E because all difference vectors originating from the same point are distinct (see proof of Theorem 4) and because there

are points from which r difference vectors are computed. There are thus at least r distinct difference vectors in the difference vector table V, which means that at least r translatable patterns will be discovered on line 5 when V is partitioned. In the case that there are exactly r patterns in E, the sizes of the patterns can be inferred from the difference vector table V. In each column of V, the vectors are distinct and in ascending order as the column is read from top to bottom (see Table 2). For there to be exactly r patterns in E, there must be only r distinct difference vectors in V. This can only occur if all vectors within each subdiagonal are equal. For example, all places on the first subdiagonal contain the difference vector v_1, on the second subdiagonal all contain v_2 and so on. Each subdiagonal corresponds to a translatable pattern. When the number of patterns in E is r, the sizes of the patterns in E are equal to the sizes of the subdiagonals, that is $(n-1), (n-2), \ldots, (n-r)$.

By plugging in the above pattern sizes into Eq. 15, an upper bound for the size of L is obtained:

$$|L| \leq \sum_{i=1}^{r} \frac{(n-i)(n-i-1)}{2} = \mathcal{O}(rn^2).$$

□

The case where $|L| = \sum_{i=1}^{r}(n-i)(n-i-1)/2$ occurs when the input dataset is D_{min} (Eq. 7) as each subdiagonal of V corresponds to an MTP when V is computed for a D_{min} dataset.

The set M is computed by removing all duplicates from L, and thus the size of M is equal to the number of distinct intrapattern differences in L.

Lemma 7. *Let D be a dataset of size n. Then $|M| = \mathcal{O}(n^2)$.*

Proof. All intrapattern differences in L are positive difference vectors that occur between points of D. There cannot be more distinct intrapattern differences than there are positive difference vectors between points of D. The size of M is therefore bounded by

$$|M| \leq \frac{n(n-1)}{2} = \mathcal{O}(n^2).$$

□

It is reasonable to consider how tight a bound the one given in Lemma 7 is. Consider the 2-dimensional dataset

$$D' = D_{max} \cup (D_{max} + (0,1)),$$

where D_{max} is as defined by Eq. 8, $|D_{max}| = n$, and $|D'| = 2n$. The set $D_{max} + (0,1)$ is D_{max} translated by the vector $(0,1)$. All difference vectors

between points of D_{max} are distinct, and D_{max} clearly forms a pattern translatable by $(0, 1)$ in D'. In the sorted dataset D'_s, every other point is from D_{max} and every other from $D_{max} + (0, 1)$, i.e., if $D'_s[i] \in D_{max}$, then $D'_s[i + 1] \in D_{max} + (0, 1)$. This means that the translatable pattern formed by D_{max} will be discovered on lines 2–5 of SIAR even when $r = 1$. There are $n(n - 1)/2$ distinct intrapattern differences in D_{max}, and therefore the size of M is quadratic in n when SIAR is run on the dataset D'.

Using Lemmas 6 and 7, it is possible to analyze the worst-case time complexity of SIAR.

Theorem 8. *Let D be a k-dimensional dataset of size n. Then the worst-case time complexity of running* SIAR *on D with parameter r ($r \ll n$) is $\mathcal{O}(kn^3)$, and the worst-case space complexity is $\mathcal{O}(krn^2)$.*

Proof. Lines 2–5 of Algorithm 3 are essentially the same as running SIA but with only r subdiagonals of V computed. The time complexity of lines 2–5 is therefore dominated by sorting V. The size of V is equal to the number of elements on the r subdiagonals, that is

$$|V| = \sum_{i=1}^{r}(n - i) = rn - \frac{r(r + 1)}{2} = \mathcal{O}(rn).$$

Sorting V takes $\mathcal{O}(krn \log rn)$ time in the worst case, and the worst-case space complexity of lines 2–5 is $\mathcal{O}(krn)$, which is caused by the size of V.

Computing L on line 6 requires performing $|L|$ vector subtractions, and sorting L takes $\mathcal{O}(k|L| \log |L|)$ time. Based on Lemma 6, the worst-case time complexity of computing and sorting L is $\mathcal{O}(krn^2(\log r + \log n))$.

The time required to compute M on lines 11–16 also depends on the size of L. M is computed by iterating through all elements of L and performing a vector comparison for each element. The worst-case time complexity of computing M is thus $\mathcal{O}(krn^2)$. Based on Lemma 7, sorting M can be performed in $\mathcal{O}(n^2 \log n)$ time using a comparison-based sorting algorithm. Sorting M does not depend on the dimensionality k because sorting is based on comparing frequencies.

The number of iterations in the loop on lines 18–20 is equal to $|M|$. In the body of the loop, the intersection can be computed in $\mathcal{O}(kn)$ time by using the sorted dataset. Translating the dataset also takes $\mathcal{O}(kn)$ time. The loop on lines 18–20 thus takes $\mathcal{O}(|M|kn)$ time. Applying Lemma 7, the worst-case time complexity of computing the MTPs on lines 18–20 is $\mathcal{O}(kn^3)$.

The worst-case time complexity of SIAR is dominated by finding the MTP for each vector in M. The worst-case time complexity of SIAR is $\mathcal{O}(kn^3)$, and the worst case occurs when the size of M is quadratic in n. The space complexity of SIAR is dominated by the size of L, resulting in an overall worst-case space complexity of $\mathcal{O}(krn^2)$. □

Although the worst-case time complexity of SIAR is greater than that of SIA, SIAR can be faster on small datasets. In practice, the running time of SIAR depends greatly on the size of translatable patterns found in the input dataset. Section 4.1.1 provides empirical results on the running times of both SIAR and SIA. It is especially notable that the worst-case time complexity of SIAR does not depend on the parameter r. In Theorem 8, it is assumed that r is much smaller than n. In the case that $r = n$, the worst-case time complexity of SIAR is $\mathcal{O}(kn^3 \log n)$, which is caused by sorting L.

3.4 The SIATEC algorithm

The algorithm SIATEC by Meredith et al. [41] computes the set $\mathcal{T}'(D)$ (Eq. 12) of TECs for all MTPs in a dataset D. SIATEC is depicted in Algorithms 4 and 5, which are based on Figure 13.7 of [39] and the presentation of SIATEC in [41].

SIATEC begins by finding all MTPs in D by computing the set of difference vectors V (lines 3–9 of Algorithm 4), sorting it to obtain V_s (line 11), and then computing the set of MTPs M by partitioning V_s (lines 12–25). This is roughly equal to running SIA on the input dataset D.

MTPs can be translationally equivalent (see Definition 2), in which case their TECs will be equal. The version of SIATEC presented in [41] uses the *vectorized representations* of MTPs to avoid computing the same TEC multiple times. The vectorized representation of a pattern Q is an ordered set of difference vectors that occur between consecutive points of Q, that is

$$VEC(Q) = \langle Q[1] - Q[0], Q[2] - P[1], \ldots, Q[l-1] - Q[l-2] \rangle, \qquad (16)$$

where l is the size of Q (Eq. 18 of [41]). If the points in patterns Q and U are in ascending lexicographical order, then Q and U are translationally equivalent if and only if their vectorized representations are equal [41]. Vectorized representations can thus be used to compare patterns for translational equivalence. On line 26 SIATEC computes the vectorized representation of each MTP and sorts the MTPs in ascending order of the size of their vectorized representation. MTPs that have vectorized representations of equal size are sorted lexicographically based on their vectorized representations. After sorting, MTPs that are translationally equivalent occur in consecutive indices of M. Avoiding the computation of the same TEC multiple times is performed by incrementing the index i until an MTP that does not have the same vectorized representation as P is found (lines 34–36).

In the presentation of SIATEC in [41], the sets V_s and D_s are used to compute the vectorized representations of the MTPs. The description of SIATEC in Algorithm 4 somewhat simplifies this process, although the result is the same. In a newer presentation of SIATEC in [39], computing the vectorized representations of MTPs is omitted altogether, and the same TEC can be computed multiple times. No reason for this omission is given in [39].

Algorithm 4 The SIATEC algorithm for computing the set of MTP TECs in a dataset.

1: **function** SIATEC(D)
2: $\quad D_s \leftarrow \text{Sort}_{Lex}(D)$
3: $\quad V \leftarrow \langle\rangle$
4: $\quad W \leftarrow$ empty $|D| \times |D|$ array
5: $\quad$ **for** $i \leftarrow 0$ to $|D| - 1$ **do** $\qquad \triangleright$ Compute difference vectors
6: $\qquad$ **for** $j \leftarrow 0$ to $|D| - 1$ **do**
7: $\qquad\quad w \leftarrow \langle D_s[j] - D_s[i], i\rangle$
8: $\qquad\quad$ **if** $j > i$ **then**
9: $\qquad\qquad V \leftarrow V \oplus \langle w\rangle$
10: $\qquad\quad W[i][j] \leftarrow w$
11: $\quad V_s \leftarrow \text{Sort}_{Lex}(V)$
12: $\quad M \leftarrow \langle\rangle$
13: $\quad v \leftarrow V_s[0][0]$
14: $\quad P \leftarrow \langle D_s[V_s[0][1]]\rangle$
15: $\quad C \leftarrow \langle V_s[0][1]\rangle$
16: $\quad$ **for** $i \leftarrow 1$ to $|V_s| - 1$ **do** $\qquad \triangleright$ Partition V_s
17: $\qquad$ **if** $V_s[i][0] = v$ **then**
18: $\qquad\quad P \leftarrow P \oplus \langle D_s[V_s[i][1]]\rangle$
19: $\qquad\quad C \leftarrow C \oplus \langle V_s[i][1]\rangle$
20: $\qquad$ **else**
21: $\qquad\quad M \leftarrow M \oplus \langle\langle P, C, v\rangle\rangle$
22: $\qquad\quad v \leftarrow V_s[i][0]$
23: $\qquad\quad P \leftarrow \langle D_s[V_s[i][1]]\rangle$
24: $\qquad\quad C \leftarrow \langle V_s[i][1]\rangle$
25: $\quad M \leftarrow M \oplus \langle\langle P, C, v\rangle\rangle$
26: $\quad M \leftarrow \text{SortByVectorizedRepresentations}(M)$
27: $\quad \mathcal{T} \leftarrow \langle\rangle$
28: $\quad i \leftarrow 0$
29: $\quad$ **while** $i < |M|$ **do** $\qquad \triangleright$ Compute TECs
30: $\qquad P \leftarrow M[i][0]$
31: $\qquad C \leftarrow M[i][1]$
32: $\qquad X \leftarrow \text{FindTranslators}(P, C, W, D)$
33: $\qquad \mathcal{T} \leftarrow \mathcal{T} \oplus \langle\langle P, X\rangle\rangle$
34: $\qquad i \leftarrow i + 1$
35: $\qquad$ **while** $i < |M| \wedge VEC(M[i][0]) = VEC(P)$ **do**
36: $\qquad\quad i \leftarrow i + 1$
37: $\quad$ **return** $\mathcal{T}$

In SIATEC all difference vectors between the points of D_s are computed

and stored in the table W along with their origin indices so that

$$W[i][j] = \langle D_s[j] - D_s[i], i \rangle$$

for $0 \leq i < |D|$ and $0 \leq j < |D|$ [41]. Each MTP is added to M in the form of a triple $\langle P, C, v \rangle$, where P is the pattern, C is an ordered set of indices for the points of P in D_s, and v is the vector for which P is the MTP in D.

Algorithm 5 Procedure used by SIATEC on line 32 to find the set of translators for an MTP.

```
 1: function FindTranslators(P, C, W, D)
 2:     R ← ⟨0⟩
 3:     for j ← 1 to |P| − 1 do                          ▷ Initialize R
 4:         R ← R ⊕ ⟨0⟩
 5:     X ← ⟨⟩
 6:     while R[0] ≤ |D| − |P| do
 7:         for j ← 1 to |P| − 1 do                      ▷ Update R
 8:             R[j] ← R[0] + j
 9:         v_0 ← W[C[0]][R[0]][0]
10:         found ← false
11:         for c ← 1 to |P| − 1 do
12:             while R[c] < |D| ∧ W[C[c]][R[c]][0] < v_0 do
13:                 R[c] ← R[c] + 1
14:             if R[c] ≥ |D| ∨ v_0 ≠ W[C[c]][R[c]][0] then
15:                 break                    ▷ v_0 cannot be a translator
16:             if c = |P| − 1 then     ▷ v_0 found in all pattern columns
17:                 found ← true
18:         if found ∨ |P| = 1 then
19:             X ← X ⊕ ⟨v_0⟩
20:         R[0] ← R[0] + 1
21:     return X
```

The set of indices C for an MTP P and the vector table W are used in the FindTranslators procedure (Algorithm 5) to find the set of translators $T(P, D)$. Once all MTPs have been computed, SIATEC uses the FindTranslators procedure to find the set of translators for each MTP in M (lines 27–33 of Algorithm 4). The set $\mathcal{T}$, returned by SIATEC, contains the TEC for each MTP in D as a pair consisting of the pattern and its set of translators in D. Essentially, $\mathcal{T} = \mathcal{T}'(D)$ on line 37 of SIATEC.

The set of translators for a point at index i in D_s can be computed using W:

$$T(\{D_s[i]\}, D) = \bigcup_{j=0}^{|D|-1} \{W[i][j][0]\}.$$

When the set of indices C for the points of a pattern P in D_s is known, the set of translators for P can also be found using W [41]:

$$T(P, D) = \bigcap_{i \in C} T(\{D_s[i]\}, D) = \bigcap_{i \in C} \left(\bigcup_{j=0}^{|D|-1} \{W[i][j][0]\} \right).$$

FindTranslators uses the above results to find the set of translators for a pattern P in D using the table W and the set of indices C of the points of P in D_s. W is a table, just like V (see Table 2), but with all the values computed. Going down any column in W, the difference vectors are in ascending order so the intersection of a set of columns in W can be computed by scanning through each of the columns once. In Algorithm 5 the array R is used for keeping track of the indices in each column of W that contains the difference vectors originating from a point belonging to P. The value $R[i]$ is the index in the column for the $(i+1)$th point of P. If the same difference vector is found in two columns, $W[i_1][j_1][0] = W[i_2][j_2]][0]$, and $i_1 < i_2$, then $j_1 < j_2$. This is exploited in updating R on lines 7–8 of Algorithm 5. If v_0 is found in all columns for the pattern P, then v_0 is a translator for P in D. The set X is used for keeping track of the found translators, and at the end of the FindTranslators routine, X is the set $T(P, D)$ as defined by Eq. 11.

The worst-case time complexity of finding all MTPs in SIATEC is $\mathcal{O}(kn^2 \log n)$, just as in SIA. The worst-case time complexity of sorting the MTPs based on their vectorized representations is also $\mathcal{O}(kn^2 \log n)$ [41]. Finding translators for MTPs requires scanning $\mathcal{O}(n)$ lines of W for each of the $n(n-1)/2$ points in the set of MTPs. The worst-case time complexity of finding translators and the whole SIATEC is thus $\mathcal{O}(kn^3)$. The space complexity of SIATEC is dominated by the size of the table W, resulting in an overall space complexity of $\mathcal{O}(kn^2)$ [41].

3.5 Variants of SIATEC

The set of all MTPs for a dataset often contains many patterns that are not musically important [41]. The set of TECs for all MTPs computed by SIATEC can therefore also contain TECs for patterns that are not musically important. Three variants of SIATEC are presented in this section, COSIATEC (Section 3.5.2), SIATECCompress (Section 3.5.3), and Forth's algorithm (Section 3.5.4). The above algorithms use SIATEC to compute TECs and then select a subset of TECs using heuristic functions. The goal of the algorithms is to provide a musically better quality output than SIATEC by only outputting the musically most important TECs. SIAR and SIACT can be used in SIATEC instead of SIA, and the modified version of SIATEC can be used with the variants of SIATEC although these variants do not necessarily improve the quality of the output [38].

3.5.1 Heuristic functions

COSIATEC, SIATECCompress, and Forth's algorithm all use *compression ratio* (or *factor*) as one of their most significant heuristic functions. The idea behind using compression ratio as a musical heuristic function is that the compressed representation or *encoding* of a score can be considered an analysis of the piece [35, 38, 39]. COSIATEC, SIATECCompress, and Forth's algorithm are essentially point-set compression algorithms [38]. The compression ratio of a TEC depends on the size of its pattern, set of translators, and *covered set*. The covered set of a TEC is the union of all points in the set of patterns in the TEC [38]. For a TEC $Q = \langle P, T \rangle$, where P is the pattern and T is the set of translators (including $\bar{0}$), the covered set of Q is defined by

$$\mathrm{COV}(Q) = \bigcup_{t \in T} P + t. \tag{17}$$

A set of TECs $\mathcal{T}$ is said to cover a dataset D if

$$D = \bigcup_{Q \in \mathcal{T}} \mathrm{COV}(Q).$$

The compression ratio of a TEC $Q = \langle P, T \rangle$ is defined by

$$\mathrm{CR}(Q) = \frac{|\mathrm{COV}(Q)|}{|P| + |T| - 1}. \tag{18}$$

Eq. 18 is based on Eq. 5 of [38]. In this thesis, the set of translators T is assumed to contain $\bar{0}$, unlike in [38]. Subtracting 1 from the size of the set of translators is therefore necessary in Eq. 18 to make it equal to Eq. 5 of [38].

In addition to compression ratio, compactness is also used as a heuristic. The compactness of a pattern is the ratio of the points belonging to the pattern and all points in the pattern's region [38]. In Section 3.3.1, one definition of compactness is presented in Eq. 14. Another type of compactness is *bounding-box compactness*. In bounding-box compactness the region is defined as the smallest box that contains all points of the pattern, and the edges of the box are aligned to the axes [41]. Figure 4 below shows an example of a bounding-box in two dimensions for the pattern denoted by crosses. The bounding-box compactness of the pattern in the example is exactly 1 as all points within the bounding-box belong to the pattern.

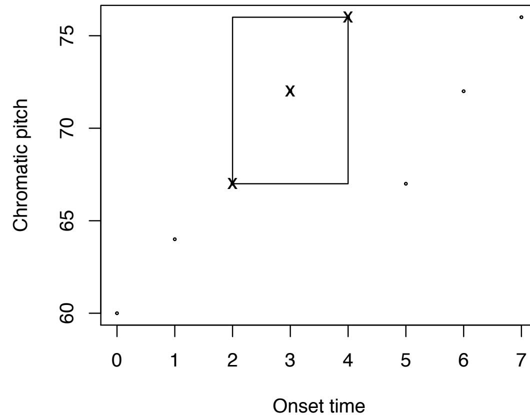


Figure 4: Bounding-box in two dimensions.

The relation of different heuristic functions to the perceived importance of musical patterns was investigated by Collins et al. [11]. Undergraduate music students ($n = 12$) were asked to rate the musical importance of patterns selected from mazurkas by Chopin. Approximately half of the patterns were selected by hand and the rest were selected randomly from the output of SIATEC. Based on the ratings, a predictive linear regression model was built. Compactness and compression ratio were found to be important parameters in the model, indicating that they are related to the perceived importance of musical patterns. However, the generality of the results is very limited because the experiment used a stylistically limited selection of patterns and the number of subjects was only 12.

3.5.2 The COSIATEC algorithm

COSIATEC [42, 36] computes a set of TECs that cover the input dataset D and form a compressed representation of D. The covered sets of the TECs output by COSIATEC do not overlap, i.e., the intersection of the covered sets of the TECs is empty. The TECs in the output are in descending order of quality.

The pseudocode for COSIATEC is shown in Algorithm 6. COSIATEC copies the input dataset D into P and then keeps finding the best TEC T_B from P and removing its covered set from P (lines 5–7) until P is empty. Removing the covered set of T_B from P ensures that no two TECs in $\mathcal{T}$ share points. COSIATEC is essentially a greedy compression algorithm [36].

Algorithm 6 COSIATEC (Figure 1 of [36]).

```
1: function COSIATEC(D)
2:     P ← Copy(D)
3:     𝒯 ← ⟨⟩
4:     while P ≠ ∅ do
5:         T_B ← GetBestTec(P, D)
6:         𝒯 ← 𝒯 ⊕ ⟨T_B⟩
7:         P ← P \ COV(T_B)
8:     return 𝒯
```

The main logic of COSIATEC is implemented in the GetBestTec function. GetBestTec computes all MTPs in P, iterates through them and computes the TEC for each. If the current computed TEC is the best so far, it will be stored. The best MTP TEC found in P is returned by GetBestTec [39]. The result is the same as running SIATEC on P and then iterating through the TECs to find the best one [38]. The implementation of COSIATEC presented in [42] also computes the *conjugate TEC* for each TEC and selects the conjugate if it is found better. The conjugate TEC for a TEC $Q = \langle P, T \rangle$ consists of pattern P' formed by translating the first point of pattern P by all translators in T. The set of translators T' for the conjugate TEC is obtained by computing the difference vectors from the first point of P to all other points in P [42]. In the presentation of COSIATEC in [39], computing conjugate TECs is omitted. The quality of TECs is compared in the following way: given two TECs in $P \subseteq D$, the better TEC is the one with

1. higher compression ratio,
2. higher bounding-box compactness in D,
3. larger covered set,
4. larger pattern,
5. lesser difference between the first components of the first and last points of the pattern (*pattern width*), or
6. lesser bounding box area (*pattern area*) [42].

The properties that are lower down in the list are considered only if the TECs are equal in the properties higher up in the list. For example, bounding-box compactness is only compared if the compression ratios are equal. The version of COSIATEC in [39] omits comparison of pattern size, pattern width, and pattern area from the evaluation of TEC quality.

The running time of COSIATEC can be great with large datasets because COSIATEC runs SIATEC on each iteration of the loop and the worst-case

time complexity of SIATEC is $\mathcal{O}(kn^3)$ for a k-dimensional dataset with n points [42]. On the other hand, the output of COSIATEC can be of better quality than that of SIATECCompress or Forth's algorithm. In [37], COSIATEC was found to perform best in folk song classification. COSIATEC also produced more compact encodings of the input dataset than other similar algorithms in both [37] and [39].

3.5.3 The SIATECCompress algorithm

SIATECCompress [42, 39] computes a set of TECs that cover the input dataset D and form a compressed representation of D, just like COSIATEC, with the exception that in the output of SIATECCompress, the covered sets of the TECs may overlap. SIATECCompress only runs SIATEC once and is faster than COSIATEC [42].

Algorithm 7 SIATECCompress (Figure 13.16 of [39]).

1: **function** SIATECCompress(D)
2: $\mathcal{T} \leftarrow \text{SIATEC}(D)$
3: $\mathcal{T} \leftarrow \text{SortDescendingByQuality}(\mathcal{T})$
4: $D' \leftarrow \emptyset$
5: $E \leftarrow \langle\rangle$
6: **for** $i \leftarrow 0$ to $|\mathcal{T}| - 1$ **do**
7: $T \leftarrow \mathcal{T}[i]$
8: $S \leftarrow \text{COV}(T)$
9: **if** $|S \setminus D'| > |pattern(T)| + |translators(T)|$ **then**
10: $E \leftarrow E \oplus \langle T \rangle$
11: $D' \leftarrow D' \cup S$
12: **if** $|D'| = |D|$ **then**
13: **break**
14: $R \leftarrow D \setminus D'$
15: **if** $|R| > 0$ **then**
16: $E \leftarrow E \oplus \langle\langle R, \langle\rangle\rangle\rangle$
17: **return** E

SIATECCompress, depicted in Algorithm 7, starts by computing all TECs in the dataset and then sorts the TECs in descending order based on their quality (lines 2–3). The quality of TECs is evaluated the same way as in COSIATEC [42] (see Section 3.5.2).

A subset of TECs is selected from the sorted set of TECs $\mathcal{T}$ in the for-loop on lines 6–13. Starting from the best TEC, SIATECCompress iterates through the set of TECs $\mathcal{T}$, adding TECs to the set E until the dataset D is covered by E or no more TECs of sufficient quality are left in $\mathcal{T}$. A TEC T is added to E if the number of new points that T adds to E is larger

than the number of points needed to represent T as a pair of pattern and set of translators (line 9). The set D' is used to keep track of points already covered by TECs in $\mathcal{T}$. The loop is terminated on line 13 if the whole dataset is covered by the selected TECs.

If the loop is not terminated on line 13, there can be points in the dataset that are not covered by the TECs in E. These points form the *residual point-set* R [39]. If R is not empty, then it is added to E as a TEC, with R as the pattern and and an empty set of translators (line 16).

SIATECCompress does not compress the input dataset as well as COSIATEC, but it is faster [39]. In the study of [37], SIATECCompress did not perform as well as COSIATEC or Forth's algorithm in folk song classification. In finding repeated patterns SIATECCompress can perform similarly to COSIATEC [38].

3.5.4 Forth's algorithm

Forth's algorithm [18] runs SIATEC once and selects a set of TECs from the output of SIATEC. Like SIATECCompress, Forth's algorithm selects a set of TECs whose covered sets may overlap. It is possible for Forth's algorithm to output a set of TECs that does not cover the input dataset [39]. Forth's algorithm takes two threshold parameters that are used in selecting TECs. Compression ratio and compactness of patterns are the most important criteria for pattern selection also in Forth's algorithm [18].

Forth's algorithm, depicted in Algorithm 8 below, handles TECs as covered sets and takes two threshold parameters c_{min} and σ_{min}. First, the set of TECs $\mathcal{T}$ for the input dataset D is computed using SIATEC (line 2). The set P is used for keeping track of the points covered by the already selected TECs, and the output is collected in the ordered set S. TECs are selected from $\mathcal{T}$ until either D is covered, or there are no more TECs of sufficient quality in $\mathcal{T}$. On line 7 the covered set of the best TEC is assigned to C_{best}. FINDBEST returns the covered set of the TEC that maximizes the weight function (Eq. 19), and the number of new points added to the cover by the TEC is at least c_{min}. The weight function used in FINDBEST is

$$\text{weight}(T) = c \cdot \text{CR}(T)_n \cdot \text{compactness-v}(T)_n, \tag{19}$$

where T is a TEC, c is the number of points in $\text{COV}(T)$ that are not in P, $\text{CR}(T)$ is compression ratio (Eq. 18), and compactness-v(T) is compactness where region is defined as in Eq. 14, but the notes in the region also have to occur in a voice that is present in the pattern. Compression ratio and compactness are both linearly normalized to be in the interval $[0, 1]$ [18], which is denoted by the subscript n.

Algorithm 8 Forth's algorithm. After Figure 13.10 of [39].

```
 1: function FORTH(D, c_min, σ_min)
 2:     𝒯 ← SIATEC(D)
 3:     S ← ⟨⟩, P ← ∅
 4:     found ← true
 5:     while P ≠ D ∧ found do
 6:         found ← false
 7:         C_best ← FINDBEST(𝒯, c_min, P)
 8:         if C_best ≠ nil then
 9:             found ← true
10:             P ← P ∪ C_best
11:             i ← 0, primaryFound ← false
12:             while ¬primaryFound ∧ i < |S| do      ▷ Find primary TEC
13:                 if |S[i][0] ∩ C_best| / |S[i][0]| > σ_min then
14:                     S[i] ← S[i] ⊕ ⟨C_best⟩           ▷ Add as secondary
15:                     primaryFound ← true
16:                 i ← i + 1
17:             if ¬primaryFound then
18:                 S ← S ⊕ ⟨⟨C_best⟩⟩                    ▷ Add as primary
19:     return S
```

In the output of Forth's algorithm, TECs are either *primary* or *secondary* [39]. If C_{best} is found, then on lines 11–16 Forth's algorithm searches for a primary TEC in S to which C_{best} could be added as a secondary TEC. If there is a primary TEC in S such that it shares a proportion of points in C_{best} that exceeds the threshold σ_{min} (line 13), then C_{best} is secondary to the primary TEC. Otherwise, C_{best} is added as a primary TEC. The output set S contains ordered sets where the first element is the covered set of a primary TEC and the other elements are covered sets of the related secondary TECs.

No analysis of time complexity or experimental running time measurements are provided for Forth's algorithm in [18]. Forth's algorithm only runs SIATEC once so it can be expected to be faster than COSIATEC. In folk-song classification Forth's algorithm performed better than SIATEC-Compress but not as well as COSIATEC [37]. In repeated pattern discovery Forth's algorithm can perform better than COSIATEC and SIATECCompress, depending on whether compactness trawling (see Section 3.3.1) is used [38].

4 Improving the running time of MTP and TEC computation by hashing

Hashing can be used to improve the running time of both MTP and TEC computation. This section presents the algorithms SIAH (Section 4.1) and SIATECH (Section 4.2), which use hashing to improve the running time of computing the set of MTPs and TECs for a dataset. Both algorithms are compared empirically against algorithms described in Section 3.

4.1 The SIAH algorithm

In [39], Meredith suggests that the running time of SIA can be improved by using hashing to partition the difference vector table V instead of sorting the table. The average, or expected, running time thus obtained would be $\mathcal{O}(kn^2)$. This section presents the algorithm SIAH, which uses hashing to compute the set of MTPs for a dataset.

In SIAH the partitioning of the set of positive difference vectors is accomplished by using a *dictionary* structure where the *keys* are placed into a hash table. In a dictionary the keys can be used to access *satellite data* [17, p.229–230]. In the case of the dictionary needed for SIAH, the keys are difference vectors, and the satellite data for each key is a list of origin indices for that difference vector. The operations that the dictionary needs to support are inserting and searching. For a dictionary $\mathcal{H}$, $\mathcal{H}[v]$ is used to denote the satellite data associated with key v, and $keyset(\mathcal{H})$ is the set of keys that have been inserted into $\mathcal{H}$.

The implementation of SIAH requires a hashing scheme for k-dimensional vectors such that when they are used as keys in a dictionary, inserting and searching take $\mathcal{O}(k)$ expected time. This expected running time can be obtained by using *universal hashing*. In universal hashing the hash function is selected at random from a class of functions when program execution is started. The class of functions is designed in such a way that the probability of two keys being mapped to the same slot in the hash table by a randomly selected function is at most $1/m$, where m is the number of slots [17, p. 265–267].

In [28], Lemire and Kaser present multiple classes of strongly universal hash functions for strings. A class of hash functions is strongly universal if the hash values of any two keys are independent of each other [28]. The MULTILINEAR class of strongly universal hash functions presented in Theorem 3.1(i) of [28] can be used in the implementation of SIAH. Strongly universal hashing is ensured for strings of fixed length. MULTILINEAR is defined by

$$h(s) = \left(\left(m_1 + \sum_{i=1}^{n} m_{i+1} s_i \right) \mod 2^K \right) \div 2^{L-1}, \tag{20}$$

where the multipliers m_i are random integers in the range $[0, 2^K)$, and s_i denotes the ith character of the string s. The characters s_i are integers in the range $[0, 2^L)$, K and L are positive integers such that $K > L - 1$, and n is the length of s. The symbol $\div$ denotes division where the result is rounded down to the nearest integer. The time complexity of computing $h(s)$ is $\mathcal{O}(n)$ [28].

The MULTILINEAR class of hash functions can be used for hashing k-dimensional vectors if each component is expressed using a fixed number of bits. For example, in an implementation of SIAH, the components of vectors could be expressed as 64-bit floating point numbers. The bits of components can be concatenated together to form a string representation of a vector such that each block of L consecutive bits is considered a character in an integer alphabet. The length of the string representation is linear in the dimensionality k and can be computed in $\mathcal{O}(k)$ time. The length of the string representation is also fixed because in any input dataset all vectors have the same dimensionality. The value of a hash function from the MULTILINEAR class can be computed in $\mathcal{O}(k)$ time for the string representation of a k-dimensional vector. Using a hash function from the MULTILINEAR class thus ensures $\mathcal{O}(k)$ expected time insert and search operations for vectors in a hash table.

SIAH is depicted in Algorithm 9. The algorithm is very similar to SIA except that there is no need to sort the set of difference vectors between points in D. No difference vector table is used as the origin indices can be placed directly in the dictionary $\mathcal{H}$. On lines 4–10 all positive difference vectors between points of D are computed. The difference vectors are used as keys in the dictionary structure $\mathcal{H}$, and a list of origin indices is associated with each key. If the difference vector d is already a key in $\mathcal{H}$, then the origin index is appended to the end of the list (line 8). Otherwise, the index is added as a new list on line 10. No sorting of the indices is necessary to ensure that the points in each pattern are in ascending order because i is only incremented in the loop on lines 4–10. At the end of the loop, $\mathcal{H}$ contains all positive difference vectors between points of D as keys and for each difference vector a list of its origin indices in D_s in ascending order.

On lines 12–17 the MTPs are collected into the set M by iterating through the keys of $\mathcal{H}$ and using the list of indices I for each key to find the points in D_s. The set M contains all non-empty MTPs in the dataset D at the end of the algorithm. In SIAH the set M is not ordered because in the output of SIAH, the MTPs are not in any particular order, unlike in the output of SIA. Theorem 9 provides the time and space complexity of SIAH

Algorithm 9 SIA with partitioning by hashing.

```
1:  function SIAH(D)
2:      D_s ← Sort_Lex(D)
3:      ℋ ← empty dictionary
4:      for i ← 0 to |D_s| − 2 do                ▷ Compute difference vectors
5:          for j ← i + 1 to |D_s| − 1 do
6:              d ← D_s[j] − D_s[i]
7:              if d ∈ keyset(ℋ) then
8:                  ℋ[d] ← ℋ[d] ⊕ ⟨i⟩
9:              else
10:                 ℋ[d] ← ⟨i⟩
11:     M ← {}
12:     for v ∈ keyset(ℋ) do                     ▷ Find MTPs using ℋ
13:         I ← ℋ[v]
14:         P ← ⟨⟩
15:         for i ← 0 to |I| − 1 do
16:             P ← P ⊕ ⟨D_s[I[i]]⟩
17:         M ← M ∪ {⟨v, P⟩}
18:     return M
```

Theorem 9. *Let D be a k-dimensional dataset of size n. The expected running time of* SIAH *on D is $\mathcal{O}(kn^2)$, and the worst-case space complexity of* SIAH *on D is $\mathcal{O}(kn^2)$.*

Proof. Sorting D on line 2 takes $\mathcal{O}(kn \log n)$ time when a comparison-based sorting algorithm is used. In the loop on lines 4–10, exactly $n(n-1)/2$ vector subtractions are computed, and the same number of searches and inserts on $\mathcal{H}$ are performed. The subtractions are computable in worst-case $\mathcal{O}(k)$ time, and the operations on $\mathcal{H}$ take $\mathcal{O}(k)$ expected time. The expected running time of lines 2–10 is therefore $\mathcal{O}(kn^2)$.

Computing the set M also takes $\mathcal{O}(kn^2)$ expected time. In the loop on lines 12–17, the number of search operations on $\mathcal{H}$ is at most $n(n-1)/2$ because there can be at most that many distinct positive difference vectors between points of D. The total number of indices through which the loop iterates is exactly the total number of points in all MTPs found in the dataset, which is $n(n-1)/2$. The overall expected running time of SIAH is thus $\mathcal{O}(kn^2)$.

The hash table in $\mathcal{H}$ can be implemented so that its size is linear in the number of keys [17, p. 256], and the number of indices in the lists associated with the keys is the total number of points in MTPs. The space required by $\mathcal{H}$ is thus $\mathcal{O}(kn^2)$. The space required by M is also $\mathcal{O}(kn^2)$. Therefore the worst-case space complexity of SIAH is $\mathcal{O}(kn^2)$. □

The quadratic expected running time of SIAH depends on the operations

on $\mathcal{H}$, taking $\mathcal{O}(k)$ expected time. Two distinct keys can be mapped to the same slot by the hash function, causing a *hash collision* to occur [17, p. 257]. It is possible that all difference vectors between the points of D are distinct and still map to the same slot in the hash table in $\mathcal{H}$. In such case the worst-case time complexity of search operations becomes linear in the number of keys (see [17, p. 258–259]), resulting in a worst-case time complexity of $\mathcal{O}(kn^2)$ for searching $\mathcal{H}$. The worst-case time complexity of SIAH is thus $\mathcal{O}(kn^4)$. However, using universal hashing makes the worst case very unlikely, and SIAH is fast in practice, as is shown in Section 4.1.1.

4.1.1 Experiments

An experimental evaluation of the running times of SIA, SIAR, and SIAH is presented in this section. SIACT is not included in the comparisons because its purpose is not to improve running time. The purpose of the experiments was to compare the running times of the algorithms and to investigate the effect of output size on running time. Output size is measured in the number of MTPs.

The algorithms were implemented in Python 3 without any additional libraries. The source code for the implementations is available at `https://github.com/otsob/repeated_pattern_discovery`. Python 3 was chosen because the implementations were only intended for comparisons between algorithms. Therefore, the implementations did not need to perform optimally as long as they were comparable with each other. All sorting operations in the algorithms used Python's built-in sort-function[1]. The dictionary structure in SIAH was implemented using Python's built-in dictionary structure[2], which uses a hash table for the set of keys. The hash function for vectors was based on the MULTILINEAR family (see Eq. 20) with $K = 64$ and $L = 32$. The components of the vectors were expressed as floating point numbers. The computation of the hash function was simplified by using the floating point numbers directly as the characters of the string s in Eq. 20 and performing the arithmetic on the floating point numbers. This potentially decreased the performance of the hash function, but the time saved by avoiding the transformation of vectors into bit strings was more significant overall. The hash values for vectors were cached to avoid unnecessarily computing the hash value multiple times for the same vector. In SIAR the parameter r was set to 3. This value for r was chosen because the same value is used in [39] for comparisons of output quality between unmodified COSIATEC and COSIATEC with SIAR.

The experiments were run on a machine equipped with two Intel Xeon

[1] `https://docs.python.org/3/library/stdtypes.html#list.sort` (accessed 12 February 2018).

[2] `https://docs.python.org/3/library/stdtypes.html#dict` (accessed 12 February 2018).

E5540 CPUs and 32GB of memory running Ubuntu 14.04. Three types of 2-dimensional datasets were used: D_{min}, D_{max}, and D_{rand} sets. In the D_{min} datasets (see Eq. 7) the number of MTPs is $n - 1$, and in the D_{max} datasets (see Eq. 8) the number of MTPs is $n(n - 1)/2$. These datasets have the minimum or maximum number of MTP occurrences possible (see Eq. 6). A scaling factor $\epsilon = 0.01$ was used in the D_{max} datasets. The D_{rand} datasets were created by generating patterns of random vectors with pattern sizes drawn uniformly at random from the range $[n/100, n/4]$. Each pattern was copied and translated 1 to 30 times by a random vector. The number of translated copies was also selected uniformly at random. The purpose of generating random datasets in this way was to ensure that there would be translatable patterns in the datasets. The D_{rand} datasets were used in place of datasets of music due to the poor availability of large scores in suitable format. The dimensionality of datasets was set to $k = 2$ in order to limit the time used in vector operations. The time complexity of all compared algorithms is linear in the number of dimensions. Therefore, investigating the effects of dimensionality on running time was not considered important.

The maximum sizes of the datasets were initially limited to $n = 7000$ with the aim of keeping the running times below one hour. In the case of the D_{rand} datasets, the size was extended to $n = 8000$ due to the running time of SIAR seeming anomalously large at $n = 7000$.

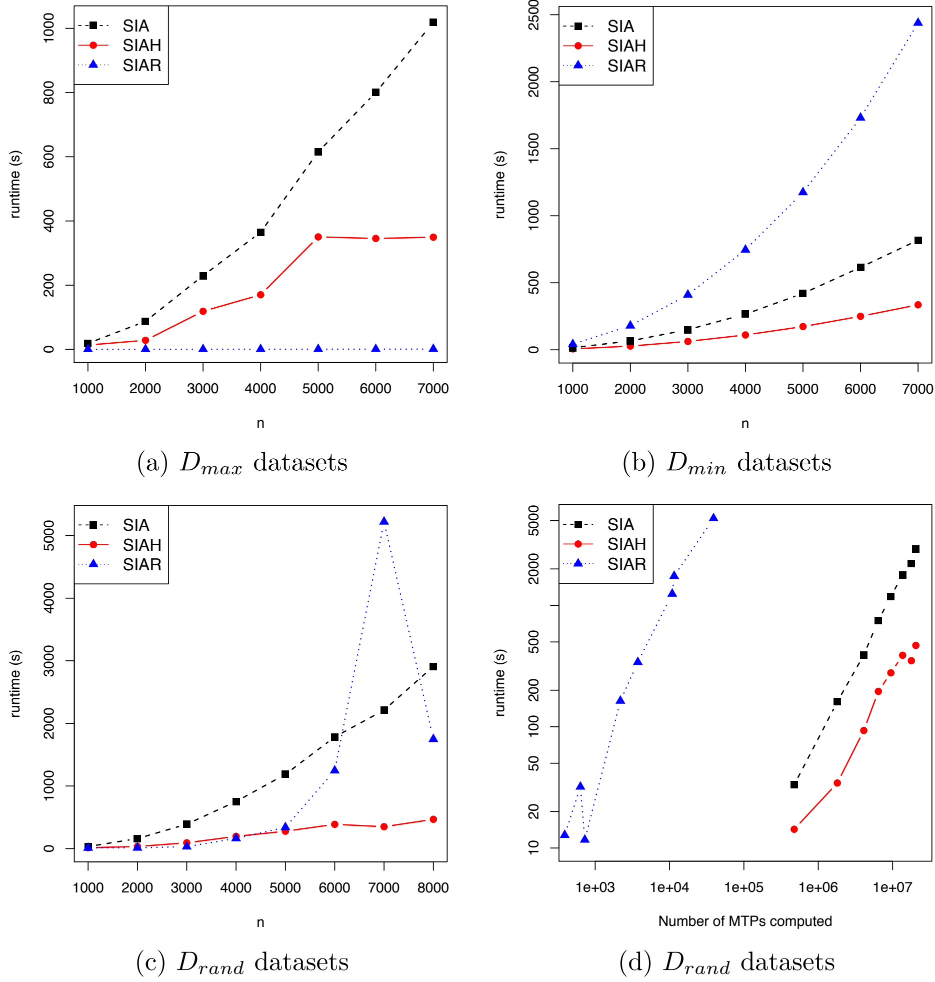
Figures 5a, 5b, and 5c below show running time against dataset size for the different datasets. In Figure 5d, running times for D_{rand} datasets are shown against output size on a logarithmic scale.

The output produced by SIAR from the D_{max} datasets is empty because all translatable patterns that are discovered on the r subdiagonals consist of only a single point. There are no intrapattern differences to be computed from the patterns and no MTPs are computed. The running time of SIAR was thus very low for the D_{max} datasets. SIAH was consistently faster than SIA on the D_{max} datasets.

The performance of SIA and SIAH on the D_{min} datasets was similar to their performance with the D_{max} datasets, with SIAH being consistently faster than SIA. The greatest difference between the D_{max} and D_{min} datasets was found in the performance of SIAR, which was slower than both SIA and SIAH with the D_{min} datasets of all tested sizes. This was because the D_{min} datasets caused the case in which the number of non-distinct intrapattern differences is maximized (see Lemma 6 and Theorem 8).

On the D_{rand} datasets SIAH is clearly the fastest with the largest dataset sizes. The most interesting result with the D_{rand} datasets occurred in the running time of SIAR. At $n = 7000$ the running time of SIAR was over 5220s. The D_{rand} measurements were extended to datasets with $n = 8000$ to see if the running time of SIAR would increase further with the size of the dataset increasing. With $n = 8000$ the running time of SIAR dropped to 1748s. The reason for this was that SIAR computed 39 063 MTPs from

the D_{rand} dataset with $n = 7000$, and only 11 536 MTPs from the dataset with $n = 8000$. The running time of SIAR was affected more by the number of MTPs it computes than by the size of the input dataset. This is in line with the time complexity analysis of SIAR in the proof of Theorem 8.



(a) D_{max} datasets

(b) D_{min} datasets

(c) D_{rand} datasets

(d) D_{rand} datasets

Figure 5: Running times of MTP discovery algorithms.

Figure 5d shows the running times of the algorithms for the D_{rand} datasets against output size, i.e., the number of MTPs the algorithm computed from the dataset. The output sizes of SIA and SIAH was the same because they both compute all MTPs in the dataset. The output size of SIAR was smaller than the output of SIA and SIAH for all D_{rand} datasets. Although the output of SIAR was small compared to SIA and SIAH, SIAR took more time per MTP. In SIAR the time complexity of computing each MTP that is added to the output is linear in the size of the dataset. This is likely to

be the reason why the running time of SIAR was so strongly affected by the number of MTPs it output. The running time benefits that SIAR gains over SIA by only computing the r subdiagonals are lost when the size of the output grows.

The running times of SIA were consistently higher on the D_{rand} datasets than on the D_{max} and D_{min} datasets. The cause of this is not clear although it might be related to the number of memory allocations performed on the dynamic lists that are used in the implementations. Finding the reasons for the differences in the performance of SIA on different types of datasets would require further measurements which are beyond the scope of this thesis.

In two cases the running time of SIAH decreases when the size of the dataset grows. This occurs with the D_{max} datasets when moving from $n = 5000$ to $n = 6000$ and with the D_{rand} datasets when moving from $n = 6000$ to $n = 7000$. The randomly selected multipliers in the hash function could not have been the cause of this as they were kept the same across all experiments, except for the runs on the $n = 8000$ D_{rand} datasets, which were run separately. It is possible that with the difference vectors computed between the points of the larger dataset, fewer hash collisions occurred, and the hash table functioned more effectively. Overall, the performance of SIAH was very similar with different types of datasets, which implies that SIAH is likely to be fast in practice with a variety of different datasets.

4.2 The SIATECH algorithm

This section presents a novel algorithm SIATECH that uses hashing to compute the set of all MTP TECs $\mathcal{T}'(D)$ (Eq. 12) for a dataset D. The purpose of SIATECH is to improve upon the running time of SIATEC. An empirical comparison of running times between SIATECH and SIATEC is provided in Section 4.2.1.

SIATECH, depicted in Algorithm 10 below, uses a dictionary $\mathcal{H}$ where the keys are inserted into a hash table similarly to SIAH. In addition to using $\mathcal{H}$ to compute MTPs, SIATECH also uses $\mathcal{H}$ to find the translators for each MTP. On lines 4–10 all positive difference vectors are computed and the corresponding indices in D_s are stored in $\mathcal{H}$. In SIATECH the *target* indices of positive difference vectors are also stored. For a vector $v \in keyset(\mathcal{H})$, the list $\mathcal{H}[v]$ contains pairs of indices $\langle i, j \rangle$, where i is the origin index and j the target index, such that $D_s[i] + v = D_s[j]$. Once all positive difference vectors are computed, the target indices of a vector v in the list $\mathcal{H}[v]$ are the indices of points in D_s that are produced when $MTP(v, D)$ is translated by v. The target indices are used by the FindTranslatorsH procedure depicted in Algorithm 11. The origin indices in $\mathcal{H}[v]$ are used to compute the MTP P for v in D (lines 14–17 of Algorithm 10). Lines 2–17 of SIATECH are roughly equal to running SIAH on the dataset D.

To avoid computing the same TEC multiple times, the translators for

P are computed only if a TEC for a translationally equivalent MTP has not already been computed. Translational equivalence is compared using the vectorized representations of MTPs. The set C contains vectorized representations of the patterns in the previously computed TECs. If the vectorized representation of P is in C, then the TEC for P is already in the set of computed TECs $\mathcal{T}$, and it is unnecessary to compute the translators for P in D. The set C is implemented as a hash table by using the MULTILINEAR class of hash functions (see Eq. 20). The MULTILINEAR class of hash functions provides strongly universal hashing for variable length strings that are not terminated by a zero character (Theorem 3.1. of [28]). A string representation can be formed for any pattern by concatenating the string representations of the vectors (see Section 4.1) in the pattern. A non-zero character can be added to the end of the string representations of patterns to ensure that the strings do not end in a zero character. Forming such a string for a k-dimensional pattern P can be performed in $\mathcal{O}(k|P|)$ time, and the length of the string is $\mathcal{O}(k|P|)$. The time complexity of computing the value of a hash function from the MULTILINEAR class is linear in the length of the string [28]. Therefore, the hash function for the string representation of a pattern can be computed in $\mathcal{O}(k|P|)$ time.

If P contains only a single point, then its set of translators T in D is computed on lines 20–22. Otherwise, the FINDTRANSLATORSH procedure is used. At the end of the loop, the TEC $\langle P, T \rangle$ is added to $\mathcal{T}$, and the vectorized representation of P is added to C (lines 25–26). On line 27 the set $\mathcal{T}$ contains the TEC for each MTP in D. The MTPs are not computed in any particular order in SIATECH. There are two consequences to this. Firstly, the output of SIATECH is not in any particular order and therefore the set $\mathcal{T}$ returned by SIATECH is not an ordered set. Secondly, the occurrence of an MTP that is used as the pattern in the representation of a TEC might not be the same in the output of SIATEC and SIATECH. Although both SIATEC and SIATECH compute the set $\mathcal{T}'(D)$, they might produce different representations of TECs.

The overall structure of SIATECH is very similar to that of SIATEC. Both algorithms first compute MTPs and then use a procedure to compute the sets of translators for TECs. The main difference between the two algorithms is in their method of finding translators. SIATECH uses the vectorized representations (see Eq. 16) of MTPs and the dictionary $\mathcal{H}$ to find the set of translators of each TEC.

Algorithm 10 SIATEC with hashing.

```
 1: function SIATECH(D)
 2:     D_s ← Sort_Lex(D)
 3:     H ← empty dictionary
 4:     for i ← 0 to |D_s| − 2 do                    ▷ Compute difference vectors
 5:         for j ← i + 1 to |D_s| − 1 do
 6:             d ← D_s[j] − D_s[i]
 7:             if d ∈ keyset(H) then
 8:                 H[d] ← H[d] ⊕ ⟨⟨i, j⟩⟩
 9:             else
10:                 H[d] ← ⟨⟨i, j⟩⟩
11:     T ← {}
12:     C ← {}
13:     for v ∈ keyset(H) do
14:         P ← ⟨⟩
15:         for i ← 0 to |H[v]| − 1 do                ▷ Compute MTP
16:             k ← H[v][i][0]
17:             P ← P ⊕ ⟨D_s[k]⟩
18:         if VEC(P) ∉ C then
19:             T ← ⟨⟩
20:             if |P| = 1 then ▷ Compute translators for single point MTP
21:                 for i ← 0 to |D_s| do
22:                     T ← T ⊕ ⟨P[0] − D_s[i]⟩
23:             else
24:                 T ← FindTranslatorsH(P, H, D_s)
25:             T ← T ∪ {⟨P, T⟩}
26:             C ← C ∪ {VEC(P)}
27:     return T
```

The translators for a pattern P are computed by finding all subsets of the dataset D that are translationally equivalent to P. The vectorized representation of P is employed in the process by applying the following results. Let P_s and Q_s be sorted patterns of length $l > 1$ and i an integer such that $0 \leq i \leq l - 2$. Any point in Q_s, except the first one, can be expressed using the previous point and the vectorized representation of Q_s, that is

$$Q_s[i+1] = Q_s[i] + VEC(Q_s)[i].$$

If $VEC(P_s) = VEC(Q_s)$, then

$$Q_s[i+1] = Q_s[i] + VEC(P_s)[i].$$

If the above equation holds for all values of i, then the vectorized representations are equal. Lemma 10 shows how the above relates to translational

equivalence.

Lemma 10. *Let P_s and Q_s be lexicographically sorted patterns of length $l > 1$. Then $Q_s[i+1] = Q_s[i] + VEC(P_s)[i]$ for $i = 1, \ldots, l-2$ if and only if $P_s \equiv_{\mathrm{T}} Q_s$.*

Proof. By applying the definition of vectorized representation (Eq. 16) and the relation $P_s \equiv_{\mathrm{T}} Q_s \Leftrightarrow VEC(P_s) = VEC(Q_s)$ (Eq. 19 of [41]), it can be directly deduced that

$$
\begin{aligned}
Q_s[i+1] = Q_s[i] + VEC(P_s)[i],\ i = 1, \ldots, l-2 \\
\Leftrightarrow VEC(P_s)[i] = Q_s[i+1] - Q_s[i],\ i = 1, \ldots, l-2 \\
\Leftrightarrow VEC(P_s)[i] = VEC(Q_s)[i],\ i = 1, \ldots, l-2 \\
\Leftrightarrow VEC(P_s) = VEC(Q_s) \\
\Leftrightarrow P_s \equiv_{\mathrm{T}} Q_s.
\end{aligned}
$$

□

The FindTranslatorsH procedure of SIATECH is based on the result presented in Lemma 10. Recall that the target indices in $\mathcal{H}[v]$ are the indices of all points in D_s that can be produced by translating points in D by v. The purpose of storing target indices in $\mathcal{H}$ is to make translating any points during the FindTranslatorsH procedure unnecessary. The points within any MTP computed in SIATECH are in ascending lexicographical order. All patterns given to FindTranslatorsH as input are therefore in ascending lexicographical order.

The FindTranslatorsH procedure is depicted in Algorithm 11. The procedure finds the lexicographically greatest points, i.e., the last points, of all patterns in D that are translationally equivalent to P. This is accomplished by traversing sequences of points in D such that the second point in a sequence can be produced by translating the first point by $VEC(P_s)[0]$, the third point by translating the second point by $VEC(P_s)[1]$ and so on. Each of the fully traversed sequences forms a pattern that is translationally equivalent to P. This is shown in the proof of Theorem 11. The last points of the sequences are used for computing the set of translators for P in D.

In Algorithm 11, the ordered set A is used for keeping track of the indices in D_s of the last points of the sequences. Once A is initialized on lines 2–5, A contains the target indices in $\mathcal{H}[VEC(P)[0]]$. These are the indices in D_s of all points that can be produced by translating points in D by $VEC(P)[0]$.

Algorithm 11 The procedure used by SIATECH to find the translators for an MTP.

```
1: function FindTranslatorsH(P, ℋ, D_s)
2:     A ← ⟨⟩
3:     v ← VEC(P)[0]
4:     for i ← 0 to |ℋ[v]| − 1 do                        ▷ Initialize A
5:         A ← A ⊕ ⟨ℋ[v][i][1]⟩
6:     for i ← 1 to |VEC(P)| − 1 do
7:         v ← VEC(P)[i]
8:         L ← ℋ[v]
9:         A′ ← ⟨⟩
10:        j ← 0
11:        k ← 0
12:        while j < |A| ∧ k < |L| do                    ▷ Find matching indices
13:            if A[j] = L[k][0] then
14:                A′ ← A′ ⊕ L[k][1]
15:                j ← j + 1
16:                k ← k + 1
17:            else if A[j] < L[k][0] then
18:                j ← j + 1
19:            else if A[j] > L[k][0] then
20:                k ← k + 1
21:        A ← A′                                        ▷ Update A
22:    T ← ⟨⟩
23:    p ← P[|P| − 1]      ▷ Use the last point of P to compute translators.
24:    for i ← 0 to |A| − 1 do
25:        T ← T ⊕ ⟨D_s[A[i]] − p⟩
26:    return T
```

On lines 6–21 the set A is updated by finding the indices of points that can be used to extend the sequences. This is done by comparing the indices in A with the origin indices in L. The list L contains the index pairs in $\mathcal{H}[VEC(P)[i]]$. Let $a \in A$ be the index of the last point of a sequence S and $\langle o, t\rangle$ be an index pair in L. If $a = o$, then the point $D_s[a]$ can be translated by $VEC(P)[i]$ to produce the point $D_s[t]$. This means that the sequence S can be extended by $D_s[t]$. The index t is added to A' as it is the index of a point that can be used to extend a sequence. The loop on lines 12–20 finds all indices in A that match origin indices in L and adds the corresponding target indices in L to A'. As all indices in the lists in $\mathcal{H}$ are in ascending order, finding the matching indices can be performed by scanning through each list once. At the end of the loop, A is updated by assigning A' to it (line 21).

On line 22 the set A contains the indices of the last points of sequences in D that can be produced using $VEC(P)$. The last point of P, the indices in A, and the sorted dataset D_s can thus be used to compute the set of translators T for P in D on lines 22–25. The correctness of the FINDTRANSLATORSH procedure is shown in the proof of Theorem 11.

Note that the set A will not be empty at any point of the FINDTRANSLATORSH procedure once it has been initialized. Each pattern P that is given to FINDTRANSLATORSH as an argument in SIATECH is an MTP. Therefore, there are always at least two translators for P in D. No benefit can thus be gained from checking whether A is empty and terminating the loop on lines 6–21 before $i = |VEC(P)|$.

Theorem 11. *For a lexicographically sorted pattern P_s in dataset D such that $|P_s| > 1$,* FINDTRANSLATORSH *computes the set of translators $T(P_s, D)$ as defined by Eq. 11.*

Proof. All references to lines in the proof refer to Algorithm 11. Each index in the set A corresponds to a point in D. Therefore, the proof can be simplified by referring to the points in D directly by using the following notation. Let $A_1 \subset D$ denote the set of points that corresponds to the indices in A after initialization on lines 2–5. Let $A_{i+1} \subset D$ denote the set of points that corresponds to the indices in A at the end of the ith iteration of the loop on lines 6–21. Let l denote the size of P_s.

The set A_1 is computed by finding all points translatable by $VEC(P_s)[0]$ in D and then translating them by $VEC(P_s)[0]$, that is,

$$A_1 = MTP(VEC(P_s)[0], D) + VEC(P_s)[0].$$

Therefore, for each point $a_1 \in A_1$ there exists a point $a_0 \in D$ such that $a_1 = a_0 + VEC(P_s)[0]$.

In the loop on lines 6–21, the computation corresponds to first finding all points in A_i that are translatable by $VEC(P_s)[i]$ in D and then translating them by $VEC(P_s)[i]$ to produce the set A_{i+1}. Therefore, for each $a_{i+1} \in A_{i+1}$ there exists a point $a_i \in A_i$ such that $a_{i+1} = a_i + VEC(P_s)[i]$.

On the last iteration of the loop on lines 6–21, the set A_{l-1} is computed. For each point $a_{l-1} \in A_{l-1}$ there exists a sequence of points $S = \langle a_0, \ldots, a_{l-1} \rangle$ such that $a_{i+1} = a_i + VEC(P_s)[i]$ for $i = 0, \ldots, l-2$. From Lemma 10 it follows that $S \equiv_{\mathrm{T}} P_s$. Each vector in $VEC(P_s)$ is positive and each point in S besides the first one is produced by adding a vector from $VEC(P_s)$ to the previous point. The points in S are clearly in ascending lexicographical order, and a_{l-1} is the lexicographically greatest point in S.

The set of difference vectors between the last point of P_s and the points in A_{l-1} is thus the set of translators for P_s in D. This set is computed on lines 22–25 and returned by the algorithm. FINDTRANSLATORSH therefore computes exactly the set $T(P_s, D)$. □

Based on Theorem 11 it is evident that SIATECH computes the TECs for MTPs correctly. Therefore, the output of SIATECH on dataset D is the set $\mathcal{T}'(D)$ (Eq. 12). The time and space complexity of SIATECH is analyzed next.

Theorem 12. *Let D be a k-dimensional dataset of size n. The expected running time of* SIATECH *on D is $\mathcal{O}(kn^3)$, and the worst-case space complexity of* SIATECH *on D is $\mathcal{O}(kn^2)$.*

Proof. Computing the MTPs in D on lines 2–17 of SIATECH is practically equal to running SIAH on D. The expected running time of lines 2–17 is therefore $\mathcal{O}(kn^2)$ (see Theorem 9).

The size of the vectorized representation of a pattern P is $|P| - 1$. The total number of vectors in the vectorized representations of all MTPs computed by SIATECH is thus $\mathcal{O}(n^2)$. The set C is implemented as a hash table by using a string representation of P and a hash function from the MULTILINEAR class. The expected running time of searching C for a pattern P is $\mathcal{O}(k|P|)$. Inserting P into C also takes $\mathcal{O}(k|P|)$ expected time. All search operations on C (line 18) and insertions into C (line 26) during the execution of SIATECH therefore take $\mathcal{O}(kn^2)$ expected time in total.

A pattern with size 1 is handled only once, and computing the set of translators for this kind of single-point pattern takes $\mathcal{O}(kn)$ time. The more significant part of SIATECH in regard to running time is the FINDTRANSLATORSH procedure. For a pattern P the FINDTRANSLATORSH procedure iterates through $VEC(P)$, and for each vector $v \in VEC(P)$ the procedure performs a search in $\mathcal{H}$ and iterates through $\mathcal{H}[v]$ and A. The search takes $\mathcal{O}(k)$ expected time. The size of $\mathcal{H}[v]$ is the number of times v occurs as a positive difference vector between points of D. This is equal to the size of $MTP(v, D)$. The size of both $\mathcal{H}[v]$ and A is bounded from above by the size of the largest MTP in the set of all MTPs $\mathcal{S}(D)$ (see Eq. 4) computed in SIATECH. Let m denote the size of the largest MTP in $\mathcal{S}(D)$. The expected running time of computing translators for all MTPs in $\mathcal{S}(D)$ is $\mathcal{O}(kmn^2)$. The largest MTP in $\mathcal{S}(D)$ can have at most $n - 1$ points. Thus, the expected running time of computing translators for all MTPs is $\mathcal{O}(kn^3)$. The expected running time of SIATECH is dominated by computing the translators, resulting in an overall expected running time of $\mathcal{O}(kn^3)$.

The number of index pairs in the lists in $\mathcal{H}$ is $\mathcal{O}(n^2)$. The space complexity of SIATECH is dominated by the size of $\mathcal{H}$ and C, which both contain $\mathcal{O}(n^2)$ k-dimensional vectors, leading to a worst-case space complexity of $\mathcal{O}(kn^2)$ for SIATECH. □

In the worst case the search operations on $\mathcal{H}$ and C are linear in the number of keys. This occurs if all keys used in $\mathcal{H}$ and C map to the same slot. For both the number of keys is quadratic in n. The search operations on C on line 18 of Algorithm 10 take $\mathcal{O}(kn^4)$ time in total in the worst case. Running

FindTranslatorsH on all vectors in the vectorized representations of the MTPs also takes $\mathcal{O}(kn^4)$ time in the worst case because a search of $\mathcal{H}$ is performed on each of the $\mathcal{O}(n^2)$ vectors. The worst-case time complexity of SIATECH is $\mathcal{O}(kn^4)$.

The asymptotic upper bound on the expected running time of SIATECH is the same as the asymptotic upper bound on the worst-case time complexity of SIATEC. Also the worst-case time complexity of SIATECH is greater than that of SIATEC. However, the FindTranslatorsH procedure used by SIATECH can be more efficient than the corresponding FindTranslators procedure used by SIATEC. The running time of FindTranslators depends on the size of the input dataset, whereas the running time of FindTranslatorsH depends on the size of MTPs found in the input dataset. If the maximum size of the MTPs found in the input dataset is small, then SIATECH can be notably faster than SIATEC.

4.2.1 Experiments

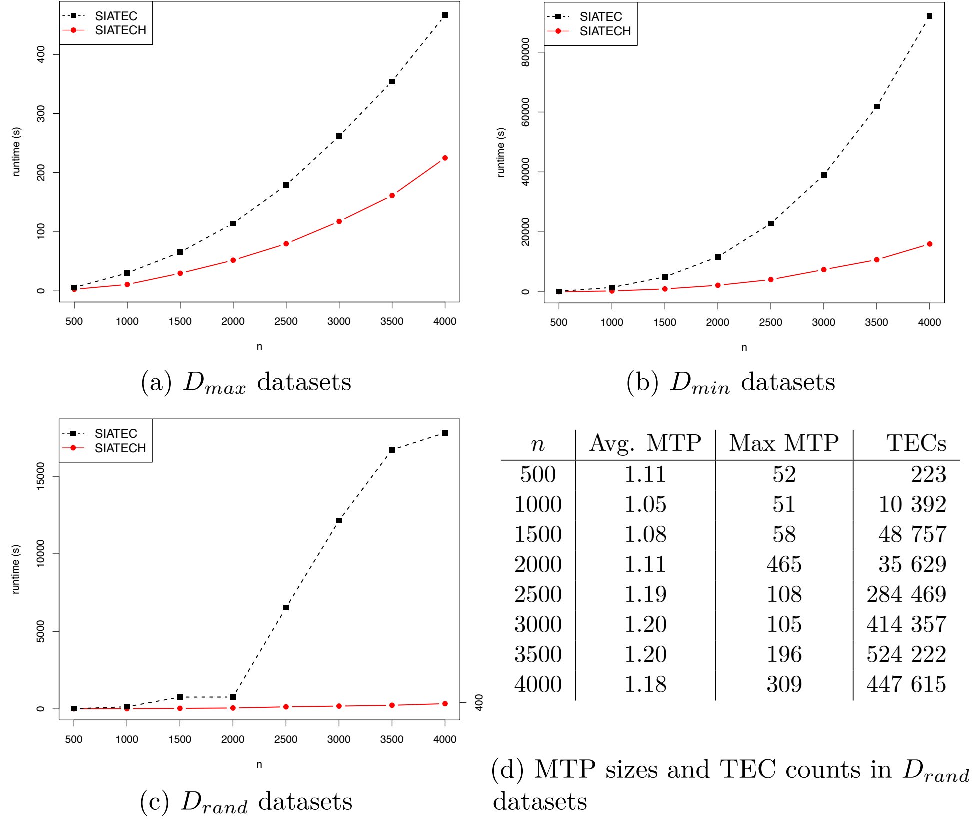
Experiments were conducted to compare the running times of SIATEC and SIATECH. The experimental setup was the same as in the experiments presented in Section 4.1.1. The algorithms were implemented in Python 3, and the experiments were run on a machine with two Intel Xeon E5540 CPUs and 32GB of memory. A version of SIATEC that does not compute the same TEC multiple times was chosen because computing the same TEC multiple times was found to greatly increase the running time of SIATEC. SIATECH does not compute the same TEC multiple times either so such an implementation of SIATEC was considered more comparable to SIATECH. SIATEC was implemented according to [41], where the vectorized representations of MTPs were computed using the sorted difference vector table V_s and the sorted dataset D_s.

The set C in SIATECH was implemented using Python's built-in set[3] data structure. The hash function for patterns was based on the Multilinear class of hash functions. The hash function was simplified by using the components of the vectors directly as the characters in the string representation and by performing the computation using floating-point arithmetic. A non-zero character was not added to the end of the string representation. These simplifications potentially compromise the guarantee of strongly universal hashing, but the performance of the hash function was considered adequate even in the simplified form. The hash function for vectors was implemented as specified in Section 4.1.1.

The D_{max} (Eq. 8), D_{min} (Eq. 7), and D_{rand} datasets were used as the input datasets in the experiments. All datasets were 2-dimensional, and their size was limited to $n = 4000$ due to the large running times of the

[3] `https://docs.python.org/3/library/stdtypes.html#set` (accessed 22 February 2018).

algorithms. The D_{rand} datasets were generated in the same manner as in the experiments presented in Section 4.1.1. New D_{rand} datasets were generated for the experiments on SIATEC and SIATECH.



(a) D_{max} datasets

(b) D_{min} datasets

(c) D_{rand} datasets

n	Avg. MTP	Max MTP	TECs
500	1.11	52	223
1000	1.05	51	10 392
1500	1.08	58	48 757
2000	1.11	465	35 629
2500	1.19	108	284 469
3000	1.20	105	414 357
3500	1.20	196	524 222
4000	1.18	309	447 615

(d) MTP sizes and TEC counts in D_{rand} datasets

Figure 6: Runtimes of TEC discovery algorithms.

Figure 6a shows the running time measurements for the D_{max} datasets, Figure 6b for the D_{min} datasets, and Figure 6c for the D_{rand} datasets. In Figure 6d, the average and maximum sizes of MTPs and the number of TECs in the D_{rand} datasets is shown.

All MTPs found in the D_{max} datasets contain only a single point. Therefore, there is only a single TEC to be found in the datasets regardless of their size. Running SIATEC and SIATECH on the D_{max} datasets is quite similar to running the algorithms without computing the translators for any MTPs. The running times on these datasets are not interesting as such, but they offer a point of comparison for evaluating the effectiveness of the procedures the algorithms use for computing translators.

The running times of both SIATEC and SIATECH are considerably greater on the D_{min} datasets than on the D_{max} datasets. In the case of SIATECH, this is caused by the fact that the vector $(1, 0)$ occurs $n - 1$ times

as a difference vector between the points of a D_{min} dataset of size n, i.e., $|MTP((1,0), D_{min})| = n - 1$. The vectorized representations of MTPs in the D_{min} datasets also only consist of the vector $(1,0)$. The size of the list L in Algorithm 11 will thus be $n-1$ for each vector in the vectorized representation of P. Assuming that the searches in $\mathcal{H}$ are performed in $\mathcal{O}(k)$ time, this is the worst case for the FindTranslatorsH procedure.

The FindTranslators procedure (Algorithm 5) of SIATEC uses the difference vector table W to find the translators. In the case of the D_{min} datasets, each diagonal of W contains only a single distinct difference vector, i.e., on the first subdiagonal all vectors are $(1,0)$, on the second subdiagonal all vectors are $(2,0)$ and so on. The FindTranslators procedure will thus access all columns at the indices in C for each vector in the column $C[0]$ that is within the range defined by the condition of the while-loop (line 6). In short, the loop on lines 11–17 will never be terminated on line 15. The D_{min} datasets thus cause worst-case performance also for the FindTranslators procedure of SIATEC.

For computing the MTPs in a dataset, SIATEC uses SIA, and SIATECH uses SIAH. The performance of SIA is quite similar on the D_{max} and D_{min} datasets, as is the performance of SIAH (see Figures 5a and 5b). Therefore, it can be reasoned that the difference in the running times of SIATEC and SIATECH between the D_{max} and D_{min} datasets is mostly caused by the time required to compute the translators for the MTPs. For both algorithms the biggest factor contributing to their running times is the computation of the translators. Although the D_{min} datasets do not necessarily cause worst-case performance overall for SIATEC or SIATECH, they cause worst-case performance for the procedures they use for computing the sets of translators for TECs.

On the D_{max} and D_{min} datasets, the growth of running time against dataset size is similar for both SIATEC and SIATECH. The difference in the performance of SIATEC and SIATECH is more notable on the D_{rand} datasets. The running time of SIATECH is considerably smaller than the running time of SIATEC for the D_{rand} datasets where $n \geq 2500$. When going from $n = 2000$ to $n = 2500$, the number of TECs in the datasets increases drastically (see Figure 6d). In SIATEC the time complexity of the FindTranslators procedure has a lower bound that is linear in the size of the dataset. Therefore, the increase in the number of TECs along with an increase in dataset size causes a steep increase in the running time of SIATEC. On the other hand, the running time of SIATECH is not as greatly affected by the increase in the number of TECs. The running time of the FindTranslatorsH procedure of SIATECH depends more on the size of the MTPs found in the dataset than on the size of the dataset.

Recall that in the FindTranslatorsH procedure the time required per vector v in $VEC(P)$ is linear in the size of $MTP(v, D)$, where D is the dataset. Although the size of an MTP in a dataset of size n can be at most

$n-1$, the MTPs in the D_{rand} datasets are generally much smaller. The average size of the MTPs in the D_{rand} datasets is at most 1.20, and the maximum size of MTPs is less than $n/10$ for most of the datasets. Computing the set of translators in SIATECH is thus very fast in the case of the D_{rand} datasets.

SIATECH is faster than SIATEC on all datasets used in the experiments. The difference between the running times of the algorithms is greatest on the D_{rand} datasets where the number of TECs is large and the size of MTPs is small. The advantage of SIATECH on these datasets is mostly due to the effectiveness of the FINDTRANSLATORSH procedure. However, it is possible that the structure of the D_{rand} datasets makes the performance difference between SIATEC and SIATECH greater than what it would be on music datasets in general.

5 Improving the running time of TEC computation by filtering

The variants of SIATEC described in Section 3.5 use heuristic functions to filter the output of SIATEC. The goal of the algorithms is to improve only the quality of the output by filtering out TECs that are not considered musically important. The running time of the variants is greater than the running time of SIATEC because all of the variant algorithms run SIATEC at least once on the input dataset during their execution. This section explores the possibility of using filtering to improve running time by avoiding the computation of certain TECs. Section 5.1 presents results on how the set $\mathcal{S}(D)$ (Eq. 4) of MTPs in a dataset D can be used to compute an upper bound on the compression ratio of a TEC in D without computing the TEC. In Section 5.2, the upper bound on compression ratio is used for filtering MTPs in a novel algorithm SIATECHF.

5.1 An upper bound on compression ratio

Recall that the TEC of a pattern P in a dataset D is the set of all subsets of D that are translationally equivalent to P (see Definition 5). Let Q be $TEC(P, D)$ represented as pattern and its set of translators in D, that is, $Q = \langle P, T \rangle$. If the set of translators T includes the zero vector, then $|T| = |TEC(P, D)|$. In other words, the number of times the pattern P occurs in D is equal to the size of T. It is possible to obtain an upper bound for the number of occurrences of pattern P in D by considering the intrapattern difference vectors. Let the set of positive intrapattern difference vectors of P be denoted by

$$\Delta(P) = \{p_2 - p_1 \mid p_1, p_2 \in P \wedge p_1 < p_2\}.$$

The same vector can occur multiple times in $\Delta(P)$, i.e. $\Delta(P)$ is a multiset. Translating P by any vector does not change $\Delta(P)$. Thus, every time the pattern P occurs in D all difference vectors in $\Delta(P)$ occur between some points in D. The pattern P cannot have more occurrences in D than the least frequent vector in $\Delta(P)$ has in $\Delta(D)$. Recall that the count of occurrences for a vector v in $\Delta(D)$ is equal to the size of $MTP(v, D)$. The size of the set of translators T is thus bounded by

$$|T| \leq \min_{v \in \Delta(P)} |MTP(v, D)|.$$

The size of P also sets an upper bound on the size of the set of translators in D. If $P \subseteq D$ and $|P| = |D|$, then P can only be translated by $\bar{0}$ in D. Decreasing the size of P by one can add at most one translator to T. Therefore, the size of the set of translators is also bounded by

$$|T| \leq |D| - |P| + 1.$$

By combining the above, the upper bound

$$|T| \leq \min \left(\min_{v \in \Delta(P)} |MTP(v, D)|, |D| - |P| + 1 \right) \tag{21}$$

for the size of the translator set of the TEC Q is obtained. Next, this upper bound is used to derive an upper bound on the size of the covered set of Q and the compression ratio of Q.

Let $\tau(P)$ denote the upper bound on the size of T as defined by Eq. 21. The size of the covered set (see Eq. 17) of $Q = \langle P, T \rangle$ is largest when the points produced by translating P by all translators in T are distinct. This is equal to all patterns in $TEC(P, D)$ being disjoint. The size of the covered set is therefore bounded by

$$|\text{COV}(Q)| \leq |P||T|.$$

By plugging in the upper bound for the size of T, the bound

$$|\text{COV}(Q)| \leq |P|\tau(P) \tag{22}$$

is obtained.

The compression ratio (see Eq. 18) of a TEC $Q = \langle P, T \rangle$ has an upper bound

$$\text{CR}(Q) \leq \frac{|P||T|}{|P| + |T| - 1},$$

which results from assuming that all patterns in the TEC are disjoint. The above upper bound increases if P is not empty and the size of T is increases. As $|T| \leq \tau(P)$, it is possible to plug $\tau(P)$ into the above inequality:

$$\text{CR}(Q) \leq \frac{|P|\tau(P)}{|P| + \tau(P) - 1}. \tag{23}$$

An upper bound on the compression ratio of TEC $Q = \langle P, T \rangle$ in D can thus be computed without computing the set of translators T by using the upper bound on T of Eq. 21.

5.2 SIATECHF: SIATECH with filtering

In [30], intrapattern difference vectors were used for filtering in an algorithm that performs pattern matching using a multidimensional representation of music. A similar idea is employed in the algorithm SIATECHF, which takes as its input a multidimensional dataset D and a compression ratio threshold c_{min}. The output of SIATECHF contains all MTP TECs in the input dataset that have a compression ratio of a least c_{min}. Intrapattern vectors and the upper bound on compression ratio of Eq. 23 are used to perform *prefiltering*: if the upper bound on compression ratio computed for an MTP does not exceed c_{min}, then the TEC for that MTP is not computed at all. In addition to prefiltering, it is necessary to compute the exact compression ratio of TECs before adding them to the output. This is called *postfiltering*. Prefiltering could also be performed by using the upper bound on the number of translators or the upper bound on the size of the covered set. Compression ratio is used in SIATECHF because compression ratio is considered to reflect the musical salience of patterns [11]. The output of SIATECHF can also be filtered further by using other heuristics, such as pattern compactness (see Section 3.5.1).

The size of the set of all intrapattern differences $\Delta(P)$ is quadratic in the size of P. Although using all intrapattern vectors to compute the upper bound is likely to result in a tighter bound, it can be too time consuming for prefiltering to actually improve running time. For prefiltering to improve the running time of SIATECHF, prefiltering must be faster than computing the TEC for an MTP and then performing postfiltering by computing the exact compression ratio of the TEC.

The expected running time of the FindTranslatorsH procedure (Algorithm 11) is $\mathcal{O}(kn|P|)$ and $\Omega(k|P|)$ for a k-dimensional pattern P. The exact compression ratio of a TEC $Q = \langle P, T \rangle$ depends on the size of the covered set of Q. Computing the covered set requires translating $|P|$ points by $|T|$ translators and collecting the distinct points thus produced. By using a hashtable to collect the distinct points, the covered set can be computed in $\mathcal{O}(k|P||T|)$ expected time. To ensure that prefiltering decreases running time, the time complexity of computing the upper bound on the compression ratio of $TEC(P, D)$ should be at most linear in the size of P. This can be accomplished by using only a subset of $\Delta(P)$. The inequality of Eq. 21 holds if $\Delta(P)$ is replaced by any subset of $\Delta(P)$. The upper bound on compression ratio thus also holds when using a subset of $\Delta(P)$.

For computing the upper bound on the number of translators (Eq. 21), SIATECHF uses the set

$$\Delta'(P) = VEC(P) \oplus \langle P[|P| - 1] - P[0] \rangle, \quad (24)$$

where $VEC(P)$ is the vectorized representation of P (see Eq. 16). The pattern P is assumed to be in ascending lexicographical order. The difference vector from the first point to the last point of P is added to $\Delta'(P)$ in order to diversify the set of vectors used for computing the upper bound. According to [24], small pitch intervals between consecutive notes are more common than large pitch intervals in multiple styles of music. The onset times of consecutive note events can also be very close to each other due to *meter*, which is the regular pulse of rhythm that often occurs in music. If the points in a pattern represent consecutive note events, then the vectors in their vectorized representation can be very similar to each other and also very frequent in $\Delta(D)$. Using a more diverse subset of $\Delta(P)$ than just $VEC(P)$ can therefore provide a tighter upper bound on compression ratio. Adding the difference vector from the first point to the last point of P thus potentially improves the bound while only adding one vector to the subset. The size of $\Delta'(P)$ is equal to the size of P. Another possibility would be to use the difference vectors originating from the first point of P. Finding the best choice of subset for computing the upper bound on compression ratio would require computing distributions of intrapattern vectors for MTPs that occur in a large and diverse corpus of music. Due to the poor availability of scores in suitable format, finding the optimal choice of intrapattern vectors is beyond the scope of this thesis.

For prefiltering to work efficiently, an effective method for finding the size of any MTP in $\mathcal{S}(D)$ is needed. This is provided by the dictionary $\mathcal{H}$. SIATECHF uses $\mathcal{H}$ in exactly the same way as SIATECH (see Algorithm 10). Given a k-dimensional dataset D, once all difference vectors and corresponding indices are inserted into the dictionary $\mathcal{H}$, it is possible to find the size of the MTP for any vector in $keyset(\mathcal{H})$ in $\mathcal{O}(k)$ expected time. This requires that the size of the lists is also computed and saved when the difference vectors are computed. The upper bound on the compression ratio of $TEC(P, D)$ can thus be computed in $\mathcal{O}(k|P|)$ expected time by using the set $\Delta'(P)$ and $\mathcal{H}$.

SIATECHF, depicted in Algorithm 12 below, is almost identical to SIATECH. The differences between the two are that SIATECHF takes the compression ratio threshold c_{min} as an argument, and on lines 9 and 12 pre- and postfiltering are performed. The computation of $\mathcal{H}$ (line 3), P (line 7), and T (line 10) is performed exactly as in SIATECH (see Algorithm 10).

Algorithm 12 SIATECH with pre- and postfiltering.

```
1:  function SIATECHF(D, c_min)
2:      D_s ← Sort_Lex(D)
3:      H ← ComputeDifferences(D_s)
4:      T ← {}
5:      C ← {}
6:      for v ∈ keyset(H) do
7:          P ← ComputeMTP(v, H)
8:          if VEC(P) ∉ C then
9:              if CR_UB(P) ≥ c_min then                          ▷ Prefiltering
10:                 T ← ComputeTranslators(P, H)
11:                 Q ← ⟨P, T⟩
12:                 if CR(Q) ≥ c_min then                         ▷ Postfiltering
13:                     T ← T ∪ {Q}
14:             C ← C ∪ {VEC(P)}
15:     return T
```

The upper bound on the compression ratio for pattern P is computed on line 9 by using the set $\Delta'(P)$ (Eq. 24) to compute the upper bound $\tau(P)$ on the number of translators as defined by Eq. 21. The value of $\tau(P)$ is plugged into Eq. 23 to compute the upper bound on the compression ratio of the TEC of P in D. If the upper bound on the compression ratio of the TEC for P in D is less than the threshold c_{min}, then the compression ratio of the TEC cannot exceed c_{min} and it is unnecessary to compute the translators for P in D.

The compression ratio of a TEC can, of course, be lower than the upper bound computed during prefiltering. It is therefore necessary to compute the exact compression ratio of Q and perform postfiltering on line 12. Only TECs that have a sufficiently high compression ratio are added to $\mathcal{T}$ so that SIATECHF returns only those TECs that have a compression ratio of at least c_{min}.

For SIATECHF to provide running time improvements over SIATECH on large datasets, its expected running time should not exceed that of SIATECH. Theorem 13 shows that performing pre- and postfiltering does not increase expected running time.

Theorem 13. *Let D be a k-dimensional dataset of size n. The expected running time of* SIATECHF *on D is $\mathcal{O}(kn^3)$, and the worst-case space complexity of* SIATECHF *on D is $\mathcal{O}(kn^2)$.*

Proof. The upper bound on the compression ratio of the TEC of a pattern P in D is computable in $\mathcal{O}(k|P|)$ expected time. The total number of points in all MTPs computed in SIATECHF is quadratic in n. Therefore, prefiltering takes $\mathcal{O}(kn^2)$ expected time in total during the execution of SIATECHF.

Computing the exact compression ratio of TEC $Q = \langle P, T \rangle$ requires computing the covered set of Q, which takes $\mathcal{O}(k|P||T|)$ expected time. The rest of the compression ratio computation consists of constant time operations. The number of translators is at most n for any pattern in D. Computing the compression ratios for all MTP TECs can be accomplished in $\mathcal{O}(kn^3)$ expected time.

The rest of SIATECHF is equal to SIATECH, which has an expected running time of $\mathcal{O}(kn^3)$ (Theorem 12). The overall expected running time of SIATECHF is therefore $\mathcal{O}(kn^3)$.

Pre- and postfiltering do not increase the space complexity of SIATECHF beyond that of SIATECH. From Theorem 12 it follows that the worst-case space complexity of SIATECHF is $\mathcal{O}(kn^2)$. □

The expected running time and space complexity of SIATECHF are equal to those of SIATECH. Whether SIATECHF is faster than SIATECH in practice is investigated in Section 5.2.1.

5.2.1 Experiments

Experiments were conducted to investigate the effects of pre- and postfiltering on the running time of computing TECs. A comparison of running time and output between SIATECHF and the point-set compression algorithms described in Section 3.5 was also conducted. All algorithms were implemented in Python 3, and the setup of the experiment was the same as described in Sections 4.1.1 and 4.2.1. For all measurements the algorithms were run on 2-dimensional random pattern D_{rand} datasets that were generated in the manner that is described in Section 4.1.1. New D_{rand} datasets were generated for the experiments presented in this section.

Figure 7 below shows the running times of SIATECH, SIATECH with added postfiltering (SIATECH-PF) and SIATECHF with two different compression ratio threshold values. The value of the used threshold is given in parentheses after the name of the algorithm in the plot legend.

Adding postfiltering to SIATECH increases the running time on the larger datasets as can be expected. On the smaller datasets the running time is slightly smaller. This is potentially caused by the smaller size of the output when postfiltering is used. A smaller output size possibly results in fewer memory allocations in the dynamic list used in the implementation to contain the output.

Prefiltering improves running time so that SIATECHF is faster than SIATECH on all input datasets. However, the improvement is not great. Increasing the compression ratio threshold was expected to improve the running time further as more patterns would be filtered already at the prefiltering stage. The running times of SIATECHF between threshold values $c_{min} = 3$ and $c_{min} = 5$ are almost identical. Based on the measurements

presented in Figure 7, performing prefiltering in SIATECHF does not provide a significant improvement in running time over SIATECH.

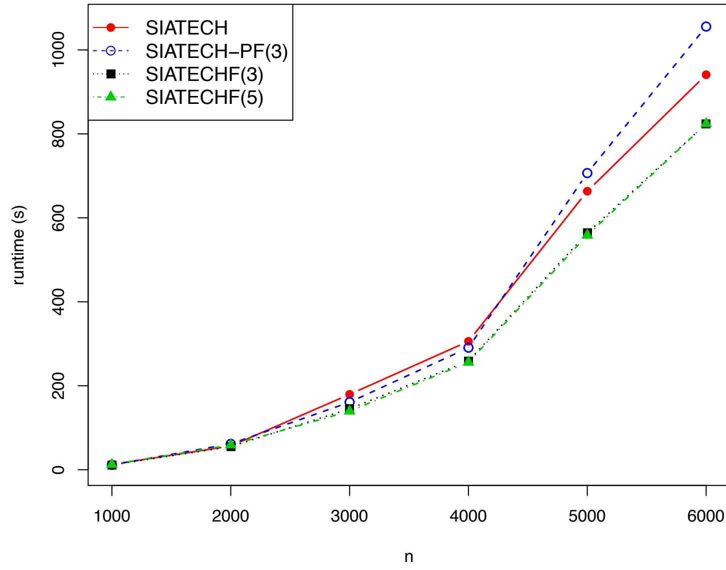


Figure 7: Effects of pre- and postfiltering on running time.

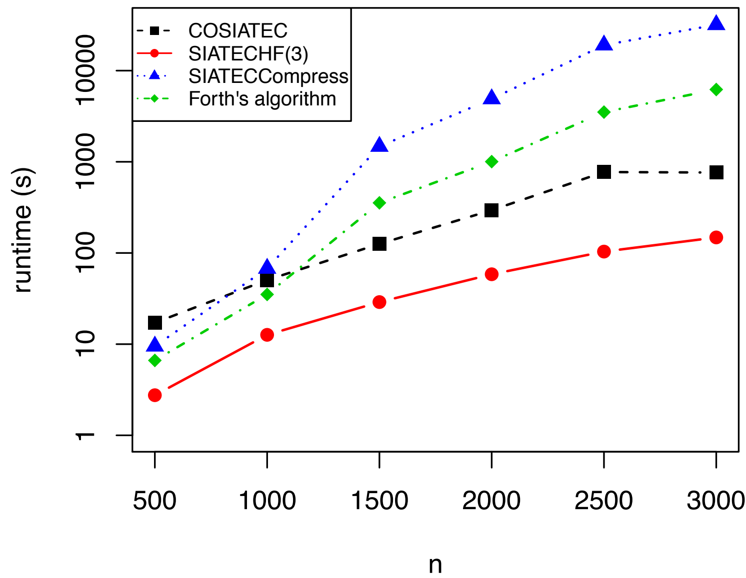


Figure 8: Filtering algorithm running times on D_{rand} datasets.

Figure 8 shows running time measurements of COSIATEC, SIATECCompress, Forth's algorithm, and SIATECHF on D_{rand} datasets. The running

times are given on logarithmic scale as the differences in the running times of the algorithms are great. In the implementations of COSIATEC, SIATECCompress, and Forth's algorithm, SIATEC was replaced by SIATECH to make their comparison with SIATECHF less affected by the running time differences between SIATEC and SIATECH. The parameter values for Forth's algorithm (see Algorithm 8) were set to $c_{min} = 15$ and $\sigma_{min} = 0.5$ after [39]. The compression ratio threshold in SIATECHF was set to $c_{min} = 3$ as the overall compression ratio of the output of COSIATEC is approximately 3 in [39].

The differences between the running times shown in Figure 8 are notable. However, a direct comparison of the running times is not very informative as the algorithms produce different outputs. SIATECHF is the fastest on all datasets. The most interesting result is that COSIATEC can be faster than SIATECCompress and Forth's algorithm when SIATEC is replaced by SIATECH.

Table 3 below shows the output sizes of Forth's algorithm, SIATECCompress, and SIATECHF on the same D_{rand} datasets that were used for the running time measurements. For each dataset the *recall* of SIATECHF is given when the output of the compared algorithm is taken as the ground truth, i.e., the correct set of TECs. In this case, recall is the fraction of TECs in the output of the compared algorithm that is also present in the output of SIATECHF. COSIATEC is not included in the comparison because the output of COSIATEC contains TECs that are not TECs of MTPs in the input dataset. For any dataset the output of SIATECHF would typically only include at most one TEC that is also present in the output of COSIATEC.

The output size of SIATECHF is greater on all datasets than the output of Forth's algorithm and SIATECCompress. However, the recall is low in most cases. The output of SIATECHF thus consists mostly of TECs that Forth's algorithm and SIATECCompress filter out as unimportant. Although Forth's algorithm and SIATECCompress use compression ratio in selecting TECs, their outputs also contain TECs with low compression ratios. This explains the generally low recall of SIATECHF when the output of Forth's algorithm or SIATECCompress is used as the ground truth.

Table 4 below shows the minimum, maximum, and average compression ratios of TECs in the outputs of COSIATEC, Forth's algorithm, and SIATECCompress. The outputs of all point-set compression algorithms contain TECs with a compression ratio of around 1 on all datasets used in the experiments. The average compression ratios of the TECs in the outputs are also often lower than the threshold $c_{min} = 3$ used in SIATECHF. On the D_{rand} dataset, with $n = 1000$, the TECs in the outputs of Forth's algorithm and SIATECCompress have the highest average compression ratio. In Table 3, SIATECHF has the highest recall on the same dataset.

Dataset size	Compared algorithm	Output size	SIATECHF output size	SIATECHF recall
500	Forth's algorithm	10	48	0.50
1000	Forth's algorithm	24	337	0.92
1500	Forth's algorithm	23	111	0.48
2000	Forth's algorithm	10	67	0.30
2500	Forth's algorithm	7	88	0.29
3000	Forth's algorithm	3	122	0.33
500	SIATECCompress	21	48	0.33
1000	SIATECCompress	21	337	0.95
1500	SIATECCompress	21	111	0.52
2000	SIATECCompress	27	67	0.78
2500	SIATECCompress	45	88	0.56
3000	SIATECCompress	45	122	0.40

Table 3: Output size comparison and recall of SIATECHF.

Dataset size	Algorithm	min CR	max CR	avg. CR
500	COSIATEC	1.00	3.86	2.04
1000	COSIATEC	1.00	5.91	2.64
1500	COSIATEC	1.00	6.30	2.74
2000	COSIATEC	1.00	5.40	3.37
2500	COSIATEC	1.00	5.09	3.08
3000	COSIATEC	1.00	7.75	3.50
500	Forth's algorithm	1.00	3.46	2.80
1000	Forth's algorithm	1.00	5.73	3.91
1500	Forth's algorithm	1.00	6.07	3.34
2000	Forth's algorithm	1.00	2.92	1.96
2500	Forth's algorithm	1.00	2.92	1.96
3000	Forth's algorithm	1.00	7.03	3.14
500	SIATECCompress	1.01	3.86	2.54
1000	SIATECCompress	1.01	5.91	4.49
1500	SIATECCompress	1.01	6.30	3.35
2000	SIATECCompress	1.02	5.40	3.62
2500	SIATECCompress	1.01	5.09	3.04
3000	SIATECCompress	1.01	7.75	3.17

Table 4: Minimum, maximum, and average compression ratios in outputs on D_{rand} datasets.

The reason why there are TECs with a low compression ratio in the outputs of COSIATEC, Forth's algorithm, and SIATECCompress is likely to be the algorithms' preference for TECs that do not overlap with already

selected TECs. In COSIATEC the covered sets of the TECs in the output do not overlap at all, and Forth's algorithm and SIATECCompress also filter out TECs that share multiple points with already selected TECs. The algorithms can therefore select TECs with a low compression ratio if the selected TECs add new points to the output. In SIATECHF compression ratio is the only criterion for TEC selection.

The use of prefiltering does not provide significant improvements in running time. The benefit of prefiltering is mostly that it cancels out the added cost of postfiltering. Filtering in SIATECHF can thus be performed without making the algorithm slower than SIATECH. Unlike the point-set compression algorithms, SIATECHF does not aim to produce a compressed representation of the input dataset. While the idea behind the point-set compression algorithms is to provide output that can be considered an analysis of the input score, SIATECHF simply finds TECs that exceed the given compression ratio threshold. The outputs of the point-set compression algorithms and SIATECHF are therefore very different.

SIATECHF, or a similar algorithm, could be usable in a pattern discovery application. The user of the application would give thresholds for parameters that the patterns in the output would need to exceed. Compression ratio provides a heuristic that emphasizes the importance of both the pattern size and the number of repetitions. However, the idea of compression ratio is not very intuitive in the context of music [11], and it could be difficult for users to select a suitable threshold value for compression ratio. Prefiltering can also be performed using MTP size and the number of repetitions by employing the upper bound of Eq. 21. These parameters could be easier for users to grasp in the context of pattern discovery in music.

One way to improve the output quality of SIATECHF would be to add further filtering based on other heuristics. SIATECHF could even be used in the point-set compression algorithms by running it with a low value for compression ratio threshold. For example, the running time of SIATECCompress is mostly caused by sorting the set of TECs and selecting the best TECs. As the output of SIATECHF is likely to be smaller than the set of all TECs for the input dataset, SIATECHF could offer significant running time improvements when used in the point-set compression algorithms. However, as the results presented in Tables 3 and 4 indicate, the output of the point-set compression algorithms would likely change when using SIATECHF. Whether leaving out TECs with a low compression ratio from the output of the point-set compression algorithms decreases the musical quality of the output requires further research.

6 Conclusions

In this thesis we set out to investigate two approaches to improving the running time of MTP and TEC computation. The first approach built on the suggestion of Meredith [39] to use hashing in SIA to improve its running time. The second approach involved the use of filtering by compression ratio to avoid the computation of musically unimportant TECs. The time and space complexities of algorithms for which asymptotic bounds are easily analyzable are given in Table 5. For the point-set compression algorithms, running time depends so greatly on the structure of the input dataset that analyzing time complexity as a function of input size is not very informative.

Algorithm	Time complexity	Space complexity
SIA	$\mathcal{O}(kn^2 \log n)$ worst-case	$\mathcal{O}(kn^2)$ worst-case
SIAR	$\mathcal{O}(kn^3)$ worst-case	$\mathcal{O}(krn^2)$ worst-case
SIAH	$\mathcal{O}(kn^2)$ expected	$\mathcal{O}(kn^2)$ worst-case
SIATEC	$\mathcal{O}(kn^3)$ worst-case	$\mathcal{O}(kn^2)$ worst-case
SIATECH	$\mathcal{O}(kn^3)$ expected	$\mathcal{O}(kn^2)$ worst-case
SIATECHF	$\mathcal{O}(kn^3)$ expected	$\mathcal{O}(kn^2)$ worst-case

Table 5: Algorithm time and space complexities on k-dimensional datasets of size n. The r in the space complexity of SIAR is the number of subdiagonals in the difference vector table (see Section 3.3.2).

The results on improving the running time of MTP and TEC computation in general were promising. Using hashing to partition the difference vectors between points of the input dataset in SIAH was found to provide great improvements in running time over SIA. The expected running time of SIAH is also lower than the worst-case running time of SIA and SIAR. The SIATECH algorithm did not provide improvements in time complexity over SIATEC. However, SIATECH improved the running time of TEC computation in practice. SIATECH was found to be significantly faster than SIATEC in cases where the size of MTPs in the input dataset is small. This improvement is attributable to the FindTranslatorsH procedure that SIATECH uses to compute the translator sets for MTPs. The worst-case time complexities of SIAH and SIATECH are greater than those of SIA and SIATEC. The use of universal hashing makes the worst case unlikely, and thus SIAH and SIATECH work efficiently in practice.

The results on using filtering to improve the running time of TEC computation were not as encouraging as the results on using hashing. Prefiltering in SIATECHF did not provide notable improvements in running time over SIATECH. The output of SIATECHF was found to differ greatly from the output of SIATECCompress and Forth's algorithm. Whether outputting only TECs that have a compression ratio exceeding a given threshold can be

used to find musically important patterns requires further research.

There are multiple interesting topics for further research and experimentation. The lack of large music datasets in suitable format poses challenges for experimentation. The running time of the algorithms often depends on the structure of the input dataset, that is, the size and number of MTPs and TECs in the dataset. The structure of the random pattern datasets that were used in the experiments does not necessarily correspond to that of music datasets. Measuring the running times of the algorithms on stylistically varied music datasets would provide more informative comparisons of the real-world performances of the algorithms. Evaluating the quality of filtering methods also requires a diverse set of ground truth analyses by domain experts. The availability of such analyses is currently very limited, which makes it challenging to develop new musically justified heuristics for filtering.

Improvements in the space complexity of the algorithms are needed if the algorithms are to be used in practice on large datasets such as orchestral scores. The quadratic space complexity of the algorithms can make it very impractical to run the algorithms on large datasets. To decrease space complexity, it is necessary to avoid keeping all MTP points in memory. This may require methods for filtering MTPs already during the computation of MTPs. The prefiltering method used in SIATECHF requires that all MTPs are computed, and thus it is not suitable as such for prefiltering MTPs.

Whether it is possible to compute the set of all MTPs $\mathcal{S}(D)$ (Eq. 4) for a dataset D in subquadratic time is not known. A subquadratic solution would clearly require a way to compute $\mathcal{S}(D)$ without computing all positive difference vectors between points of D. One possible approach to analyzing the hardness of computing $\mathcal{S}(D)$ is by relating it to the 3Sum problem. The 3Sum problem [20, 21] is a decision problem, where given a set of n real numbers, the goal is to decide whether there are 3 elements in the set such that their sum is 0. A problem P is said to be 3Sum-*hard* if an algorithm that solves P in subquadratic time can be used to solve 3Sum in subquadratic time [20]. Many problems in computational geometry are known to be 3Sum-hard [2]. In [32] and [6], it is shown that there are 3Sum-hard problems in pattern matching using multidimensional representations of music. It has been conjectured that no subquadratic solutions to 3Sum exist, implying a quadratic lower bound for 3Sum-hard problems [22]. However, subquadratic solutions to 3Sum have been recently discovered [22, 21, 19]. If a lower bound on the time complexity of computing $\mathcal{S}(D)$ exists, it provides further motivation for developing algorithms that do not compute all MTPs.

Although the impact of prefiltering on the running time of SIATECHF is not great, SIATECHF provides an example of how prefiltering can be employed in TEC computation. Avoiding the computation of musically unimportant MTPs and TECs altogether can decrease running time and space requirements, and improve the quality of output. It is therefore a promising approach to further improving repeated pattern discovery algorithms that

operate on multidimensional representations of music.

References

[1] Akutsu, Tatsuya: On determining the congruence of point sets in d dimensions. *Computational Geometry*, 9(4):247–256, 1998.

[2] Barequet, Gill and Har-Peled, Sariel: Polygon containment and translational min-Hausdorff-distance between segment sets are 3SUM-hard. *International Journal of Computational Geometry & Applications*, 11(4):465–474, 2001.

[3] Bent, Ian D., Hughes, David W., Provine, Robert C., Rastall, Richard, Kilmer, Anne, Hiley, David, Szendrei, Janka, Payne, Thomas B., Bent, Margaret, and Chew, Geoffrey: Notation. In *Grove Music Online. Oxford Music Online*. Oxford University Press, accessed December 4, 2017, 2017. `http://www.oxfordmusiconline.com/subscriber/article/grove/music/20114pg7`.

[4] Bent, Ian D. and Pople, Anthony: Analysis. In *Grove Music Online. Oxford Music Online*. Oxford University Press, accessed December 7, 2017, 2017. `http://www.oxfordmusiconline.com/subscriber/article/grove/music/41862`.

[5] Braß, Peter: Combinatorial Geometry Problems in Pattern Recognition. *Discrete & Computational Geometry*, 28:495–510, 2002.

[6] Clifford, Raphaël, Christodoulakis, Manolis, Crawford, Tim, Meredith, David, and Wiggins, Geraint: 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)*, Victoria, Canada, 2006.

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

[8] Collins, Tom, Arzt, Andreas, Flossmann, Sebastian, and Widmer, Gerhard: 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.

[9] Collins, Tom, Arzt, Andreas, Frostel, Harald, and Widmer, Gerhard: Using Geometric Symbolic Fingerprinting to Discover Distinctive Patterns

in Polyphonic Music Corpora. In Meredith, David (editor): *Computational Music Analysis*, pages 445–474. Springer International Publishing, 2016.

[10] Collins, Tom, Laney, Robin, Willis, Alistair, and Garthwaite, Paul H.: Using Discovered, Polyphonic Patterns to Filter Computer-generated Music. In *Proceedings of the International Conference on Computational Creativity*, pages 1–10, Lisbon, Portugal, 2010.

[11] Collins, Tom, Laney, Robin, Willis, Alistair, and Garthwaite, Paul H.: Modeling Pattern Importance in Chopin's Mazurkas. *Music Perception*, 28(4):387–414, 2011.

[12] Collins, Tom and Meredith, David: 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*, pages 88–99. Springer, Berlin, 2013.

[13] Collins, Tom, Thurlow, Jeremy, Laney, Robin, Willis, Alistair, and Garthwaite, Paul H.: 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.

[14] Conklin, Darrell: Representation and Discovery of Vertical Patterns in Music. In Anagnostopoulou, Christina, Ferrand, Miguel, and Smaill, Alan (editors): *Music and Artificial Intelligence. Lecture Notes in Computer Science, vol 2445*, pages 32–42. Springer, Berlin, 2002.

[15] Conklin, Darrell and Anagnostopoulou, Christina: Representation and Discovery of Multiple Viewpoint Patterns. In *Proceedings of the International Computer Music Conference (ICMC 2001)*, La Habana, Cuba, 2001.

[16] Conklin, Darrell and Witten, Ian H.: Multiple viewpoint systems for music prediction. *Journal of New Music Research*, 24(1):51–73, 1995.

[17] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., and Stein, Clifford: *Introduction to Algorithms.* The MIT Press, 3rd edition, 2009.

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

[19] Freund, Ari: Improved Subquadratic 3SUM. *Algorithmica*, 77(2):440–458, 2017.

[20] Gajentaan, Anka and Overmars, Mark H.: On a class of $O(n^2)$ problems in computational geometry. *Computational Geometry*, 5(3):165–185, 1995.

[21] Gold, Omer and Sharir, Micha: Improved Bounds for 3SUM, k-SUM, and Linear Degeneracy. *CoRR*, abs/1512.05279, 2015. `http://arxiv.org/abs/1512.05279`.

[22] Grønlund, Allan and Pettie, Seth: Threesomes, Degenerates, and Love Triangles. *CoRR*, abs/1404.0799, 2014. `http://arxiv.org/abs/1404.0799`.

[23] Heffernan, Paul J. and Schirra, Stefan: Approximate decision algorithms for point set congruence. *Computational Geometry*, 4(3):137–156, 1994.

[24] Huron, David: *Sweet Anticipation : Music and the Psychology of Expectation.* The MIT Press, 2006.

[25] Janssen, Berit, Haas, W. Bas de, Volk, Anja, and Kranenburg, Peter van: Finding Repeated Patterns in Music: State of Knowledge, Challenges, Perspectives. In Aramaki, Mitsuko, Derrien, Olivier, Kronland-Martinet, Richard, and Ystad, Sølvi (editors): *Sound, Music, and Motion. CMMR 2013. Lecture Notes in Computer Science, vol 8905*, pages 277–297. Springer, Cham, 2013.

[26] Lartillot, Olivier: Automated Motivic Analysis: An Exhaustive Approach Based on Closed and Cyclic Pattern Mining in Multidimensional Parametric Spaces. In Meredith, David (editor): *Computational Music Analysis*, pages 273–302. Springer International Publishing, 2016.

[27] Lartillot, Olivier and Toiviainen, Petri: Motivic matching strategies for automated pattern extraction. *Musicae Scientiae, Discussion Forum 4A*, pages 281–314, 2007.

[28] Lemire, Daniel and Kaser, Owen: Strongly Universal String Hashing is Fast. *The Computer Journal*, 57(11):1624–1638, 2014.

[29] Lemström, Kjell: *String Matching Techniques for Music Retrieval.* PhD thesis, University of Helsinki, 2000.

[30] Lemström, Kjell, Mikkilä, Niko, and Mäkinen, Veli: Filtering methods for content-based retrieval on indexed symbolic music databases. *Information Retrieval Journal*, 13(1):1–21, 2010.

[31] Louboutin, Corentin and Meredith, David: Using general-purpose compression algorithms for music analysis. *Journal of New Music Research*, 45(1):1–16, 2016.

[32] Lubiw, Anna and Tanur, Luke: Pattern matching in polyphonic music as a weighted geometric translation problem. In *Proceedings of the 5th International Conference on Music Information Retrieval (ISMIR 2004)*, Barcelona, Spain, 2004.

[33] Meredith, David: Point-set algorithms for pattern discovery and pattern matching in music. In Crawford, T. and Veltkamp, R. C. (editors): *Proceedings of the Dagstuhl Seminar on Content-Based Retrieval (No. 06171)*, Dagstuhl, Germany, 2006.

[34] Meredith, David: The ps13 pitch spelling algorithm. *Journal of New Music Research*, 35(2):121–159, 2006.

[35] Meredith, David: Analysis by compression: Automatic generation of compact geometric encodings of musical objects. In *The Music Encoding Conference*, Mainz, Germany, 2013.

[36] Meredith, David: COSIATEC and SIATECCompress: Pattern Discovery by Geometric Compression. In *MIREX 2013. Competition on Discovery of Repeated Themes and Sections*, Curitiba, Brazil, 2013.

[37] Meredith, David: Using point-set compression to classify folk songs. In *The Fourth International Workshop on Folk Music Analysis (FMA 2014)*, Istanbul, Turkey, 2014.

[38] Meredith, David: Music Analysis and Point-Set Compression. *Journal of New Music Research*, 44(3):245–270, 2015.

[39] Meredith, David: Analysing Music with Point-Set Compression Algorithms. In Meredith, David (editor): *Computational Music Analysis*, pages 335–366. Springer International Publishing, 2016.

[40] Meredith, David: Using SIATECCompress to Discover Repeated Themes and Sections in Polyphonic Music. In *MIREX 2016. Competition on Discovery of Repeated Themes and Sections*, New York, USA, 2016.

[41] Meredith, David, Lemström, Kjell, and Wiggins, Geraint A.: Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music. *Journal of New Music Research*, 31(4):321–345, 2002.

[42] Meredith, David, Lemström, Kjell, and Wiggins, Geraint A.: Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music. *Cambridge Music Processing Colloquium*, Department of Engineering, University of Cambridge, 2003.

[43] Meredith, David, Wiggins, Geraint A., and Lemström, Kjell: Pattern Induction and matching in polyphonic music and other multidimensional

datasets. In *Proceedings of the 5th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, Florida, USA, 2001.

[44] Meyer, Leonard B.: *Style and Music. Theory, History, and Ideology.* University of Pennsylvania Press, 1989.

[45] Rezende, P.J. de and Lee, D.T.: Point Set Pattern Matching in d-Dimensions. *Algorithmica*, 13(4):387–404, 1995.

[46] Rosen, Charles: *Sonata Forms. Revised Edition.* W. W. Norton & Company, Inc., 1988.

[47] Thompson, William F.: Intervals and Scales. In Deutsch, Diane (editor): *Psychology of Music*, pages 107–140. Elsevier, 2013.

[48] Ukkonen, Esko, Lemström, Kjell, and Mäkinen, Veli: 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.

[49] Velarde, Gissel, Meredith, David, and Weyde, Tillman: A Wavelet-Based Approach to Pattern Discovery in Melodies. In Meredith, David (editor): *Computational Music Analysis*, pages 303–333. Springer International Publishing, 2016.

[50] Wiggins, Geraint A., Lemström, Kjell, and Meredith, David: SIA(M)ESE: An Algorithm for Transposition Invariant, Polyphonic Content-Based Music Retrieval. In *Proceedings of the 3rd International Conference on Music Information Retrieval (ISMIR 2002)*, Paris, France, 2002.

[51] Wikipedia contributors: MIDI — *Wikipedia, The Free Encyclopedia*, 2017. `https://en.wikipedia.org/w/index.php?title=MIDI&oldid=812897338`, accessed December 4, 2017.