Chapter 13
Analysing Music with Point-Set Compression Algorithms

David Meredith

Abstract Several point-set pattern-discovery and compression algorithms designed for analysing music are reviewed and evaluated. Each algorithm takes as input a point-set representation of a score in which each note is represented as a point in pitch-time space. Each algorithm computes the *maximal translatable patterns* (MTPs) in this input and the *translational equivalence classes* (TECs) of these MTPs, where each TEC contains all the occurrences of a given MTP. Each TEC is encoded as a $\langle \text{pattern}, \text{vector set} \rangle$ pair, in which the vector set gives all the vectors by which the pattern can be translated in pitch-time space to give other patterns in the input dataset. Encoding TECs in this way leads, in general, to compression, since each occurrence of a pattern within a TEC (apart from one) is encoded by a single vector, that has the same information content as one point. The algorithms reviewed here adopt different strategies aimed at selecting a set of MTP TECs that collectively cover (or almost cover) the input dataset in a way that maximizes compression. The algorithms are evaluated on two musicological tasks: classifying folk song melodies into tune families and discovering repeated themes and sections in pieces of classical music. On the first task, the best-performing algorithms achieved success rates of around 84%. In the second task, the best algorithms achieved mean F_1 scores of around 0.49, with scores for individual pieces rising as high as 0.71.

13.1 Music Analysis and Data Compression

A *musical analysis* represents a particular way of understanding certain structural aspects of a *musical object*, where such an object may be any quantity of music, ranging from a motive, chord or even a single note through to a complete work or even an entire corpus of works (cf. Bent, 1987, p. 1). In the spirit of the theory of

David Meredith
Department of Architecture, Design and Media Technology, Aalborg University, Aalborg, Denmark
e-mail: dave@create.aau.dk

D. Meredith (ed.), *Computational Music Analysis*,
DOI 10.1007/978-3-319-25931-4_13

Kolmogorov complexity (Chaitin, 1966; Kolmogorov, 1965; Li and Vitányi, 2008; Solomonoff, 1964a,b) and the *minimum description length principle* (Rissanen, 1978), a musical analysis is conceived of here as being a compact or compressed encoding of an *in extenso description* of a musical object. An in extenso description is one in which the properties of each atomic component of the object are explicitly specified, without encoding any structural groupings of these components into higher-level constituents, and without encoding any relationships between atomic components (see Simon and Sumner (1968, 1993) for a similar use of the term, "in extenso"). Music theorists and analysts often refer to such in extenso descriptions as "musical surfaces" (Lerdahl and Jackendoff, 1983, pp. 3,10–11) (see also Chap. 2, this volume). The atomic components themselves might be, for example, notes in a score or events in a MIDI file or sample values in a PCM audio file. Their nature thus depends on both the nature of the object being described (e.g., a specification of what to play or a recording of an actual performance) and the level of detail or "granularity" of the description.

In contrast, while a musical analysis is itself a description of a musical object, it will typically differ from an in extenso description by representing the object as being constructed from *sets* of atomic components that form larger-scale constituents, such as motives, phrases, chords, voices and sections. An analysis will also typically encode *relationships* between such constituents (e.g., repetition, transposition, inversion, elaboration, augmentation and diminution).

The *Kolmogorov complexity* of an object is, roughly speaking, the length in bits of the shortest possible program that generates the object as its only output. In the spirit of Kolmogorov complexity, in this chapter, an analysis is thus conceived of as a program that outputs the in extenso description of a musical object that we want to analyse and explain. The precise way in which such a program generates the in extenso surface description of the object constitutes an hypothesis as to how that surface might have come about. Equivalently, we can thus consider an analysis to be a losslessly compressed encoding of an in extenso description of a musical object. In this way, an analysis is an *explanation for* or a *way of understanding* certain aspects of the structure of the musical object that it describes.

Suppose X and Y are two constituent sets of atomic components of a musical object (e.g., two sets of notes forming two occurrences of the subject in a fugue) and that the transformation, T, maps X onto Y (e.g., T could be "shift in time by x quarter notes and transpose by y semitones"). If T can be described more parsimoniously than Y, then the part of the musical surface consisting of X and Y (i.e., $X \cup Y$, or the union of the atomic components in X and Y) can be described more parsimoniously by giving an in extenso description of X together with a description of T, than it can by giving in extenso descriptions of both X and Y. In this way, by identifying structural relationships between constituents of a musical object, a musical analysis can convey at least as much information about that object as an in extenso description on the same level of detail, but may manage to do so more parsimoniously. A musical analysis can thus take the form of a compact description or compressed encoding of a musical object. Of course, a musical analysis usually conveys *more* information than an in extenso description on the same level of detail, since it also typically describes groupings of atomic components into larger constituents and structural relationships between these constituents.

Kolmogorov complexity (Chaitin, 1966; Kolmogorov, 1965; Li and Vitányi, 2008; Solomonoff, 1964a,b) (also known as *algorithmic information theory*, see also Chap. 7, this volume) suggests that the *length* of a program, whose output is an in extenso description of an object, can be used as a measure of the complexity of its corresponding explanation for the structure of that object: if we have two programs in the same programming language that generate the same output, then the shorter of the two will typically represent the simpler explanation for that output.

The level of structural detail on which an analysis (encoded as a program) explains the structure of a musical object is determined by the granularity of the in extenso description that it generates. Typically, much of the detailed structure of an object will not be encoded in the in extenso description generated by an analysis and this omitted structure will therefore go unexplained.

In the work presented in this chapter, I assume that the music analyst's goal is to find the *best possible* explanations for the structures of musical objects, which raises the question of how we are supposed to decide, given a pair of alternative explanations for the same musical object, which of these explanations is "better". In my view, we can only meaningfully claim that one analysis is "better than" another if it allows us to more successfully carry out some objectively evaluable task; and even then, we can only claim that the analysis is superior *for that task*. Such tasks might include, for example, detecting errors, memorizing pieces, identifying composers (or dates of composition, forms or genres), and predicting how incomplete pieces might be completed. It is possible that the best analysis for carrying out one such task might be different from the best analysis for carrying out another. However, the algorithms and experiments described in this chapter are founded on the hypothesis that the best possible explanations for a musical object (i.e., the best analyses for *all* objectively evaluable tasks) are those that are represented by that object's shortest possible descriptions—that is, the descriptions of the object whose lengths are equal to the Kolmogorov complexity of that object.

In general, the Kolmogorov complexity of an object is not computable.[1] This means that, if we have some in extenso description of an object and a compressed encoding of that description, then typically we cannot be sure that the compressed encoding is the shortest one possible. We can, of course, often prove that an encoding is *not* the shortest possible, simply by finding a shorter one. However, in the current context, the non-computability of Kolmogorov complexity does not pose a problem, as we will only use the relative lengths (i.e., information content) of analyses to predict which analyses will serve us better for carrying out musicological tasks. That is, in order to be able to evaluate whether we are making progress, we never really need to know if an analysis is the best possible (although, of course, that would be nice), we only need to be able to predict (at least some of the time) whether or not it will be better than another one.

[1] Actually prefix complexity, K, is upper semicomputable, but it cannot be approximated in general in a practically useful sense (Li and Vitányi, 2008, p. 216).

If we adopt the approach outlined above, then the music analyst's goal becomes that of finding the shortest possible "programs" (i.e., encodings, analyses) that generate the most detailed representations of as much music as possible. In other words, our ultimate goal may be considered to be to compress as detailed a description as possible of as much music as possible into as short an encoding as possible.

With this goal in mind, the main purpose of this chapter is to present, analyse and evaluate a number of compression algorithms that have been specifically designed for analysing music (see Sect. 13.3). Each of these algorithms takes an in extenso description of a musical object as input and generates a losslessly compressed encoding of this description as output. All of the algorithms examined in this study are based on Meredith et al.'s (2002) "Structure Induction Algorithm" (SIA) which takes as input a multidimensional point set (in this context, representing a musical object) and computes all the maximal repeated patterns in the point set.

The results of evaluating these compression algorithms on two musicological tasks will also be presented (see Sect. 13.4). In the first task, the algorithms were used as compressors to compute the *normalized compression distance* (NCD) (Li et al., 2004) between each pair of melodies in the *Annotated Corpus* of Dutch folk songs from the collection *Onder de groene linde* (Grijp, 2008; van Kranenburg et al., 2013; Volk and van Kranenburg, 2012). The NCD between two objects, x and y, is defined as follows:

$$\mathrm{NCD}(x,y) = \frac{Z(xy) - \min\{Z(x),Z(y)\}}{\max\{Z(x),Z(y)\}}\,, \tag{13.1}$$

where $Z(x)$ is the length of a compressed encoding of x and $Z(xy)$ is the length of a compressed encoding of a concatenation of x and y. The NCD between two objects is designed to be a practical alternative to the *normalized information distance* (NID), a universal similarity metric based on Kolmogorov complexity. These calculated NCDs were then used to classify the melodies into tune families and the classifications generated were compared with "ground-truth" classifications provided by musicologists (van Kranenburg et al., 2013; Volk and van Kranenburg, 2012).

In the second task, the algorithms were used to find repeated themes and sections in five pieces of Western classical music in various genres and from various historical periods (Collins, 2013a). The analyses generated by the algorithms were, again, compared with "ground-truth" analyses provided by expert analysts (Collins, 2013b).

Before presenting the algorithms, however, it is first necessary to review some basic concepts and terminology relating to representing music with point sets.

13.2 Representing Music with Point Sets

In the algorithms considered in this chapter, it is assumed that the music to be analysed is represented in the form of a multi-dimensional point set called a *dataset*, as described by Meredith et al. (2002). Most of these algorithms work with datasets of any dimensionality. However, it will be assumed here that each dataset is a set of

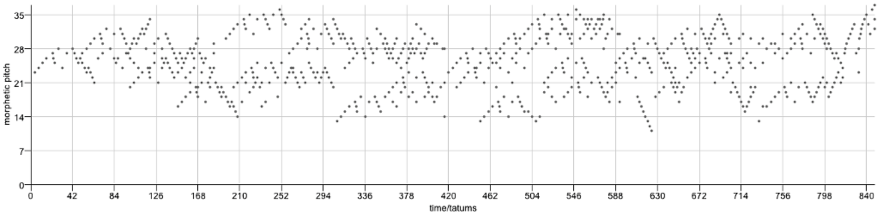

Fig. 13.1 An example of a dataset. A two-dimensional point-set representing the fugue from J. S. Bach's Prelude and Fugue in C minor, BWV 846. The horizontal axis represents onset time in tatums; the vertical axis represents morphetic pitch. Each point represents a note or a sequence of tied notes

two-dimensional points, $\langle t, p \rangle$, where t and p are integers representing, respectively, the onset time in *tatums* and the *chromatic* or *morphetic pitch* (Meredith, 2006b, 2007; Meredith et al., 2002) of a note or sequence of tied notes in a score. Assuming the musical object to be analysed is the score of a single movement, then the tatum for that score is defined to be the largest common divisor of every note onset and duration in the score. The chromatic pitch of a note is an integer indicating the key on a normal piano keyboard that would need to be pressed to play the note. For example, if we define the chromatic pitch of A0 to be 0, then the chromatic pitches of B♯3, C4 and C♯4 are 39, 39 and 40, respectively. The morphetic pitch of a note is an integer that depends on the position of the head of a note on the staff and the clef in operation on that staff. It indicates pitch height while ignoring accidental. For example, if we define the morphetic pitch of A0 to be 0, then the morphetic pitches of B♯3, C4 and C♯4 are 22, 23 and 23, respectively. For a more extensive and in-depth discussion of these and other pitch representations, see Meredith (2006b, pp. 126–130).

Figure 13.1 shows an example of such a dataset. When the music to be analysed is modal or uses the major–minor tonal system, the output of the algorithms described below is typically better when morphetic pitch is used. If morphetic pitch information is not available (e.g., because the data is only available in MIDI format), then, for modal or tonal music, it can be computed with usually very high accuracy from a representation that provides the chromatic pitch (or MIDI note number) of each note, by using an algorithm such as PS13s1 (Meredith, 2006b, 2007). For pieces of music not based on the modal or major–minor tonal system, using chromatic pitch may give better results than using morphetic pitch.

13.2.1 Maximal Translatable Patterns (MTPs)

If D is a dataset (as just defined), then any subset of D may be called a *pattern*. If $P_1, P_2 \subseteq D$, then P_1, P_2, are said to be *translationally equivalent*, denoted by $P_1 \equiv_{\mathrm{T}} P_2$, if and only if there exists a vector v, such that P_1 translated by v is equal to P_2. That is,

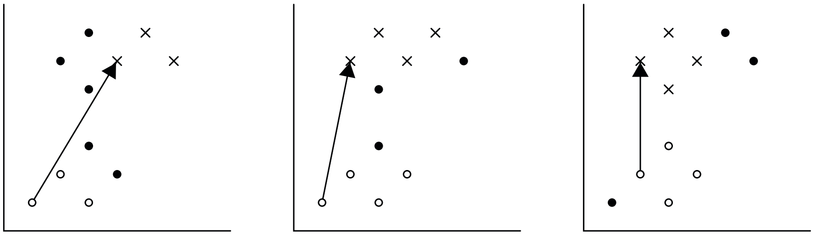

Fig. 13.2 Examples of maximal translatable patterns (MTPs). In each graph, the pattern of circles is the maximal translatable pattern (MTP) for the vector indicated by the arrow. The pattern of crosses in each graph is the pattern onto which the pattern of circles is mapped by the vector indicated by the arrow

$$P_1 \equiv_{\mathrm{T}} P_2 \iff (\exists v \mid P_2 = P_1 + v) \, , \tag{13.2}$$

where $P_1 + v$ denotes the pattern that results when P_1 is translated by the vector v. For example, in each of the graphs in Fig. 13.2, the pattern of circles is translationally equivalent to the pattern of crosses. A pattern, $P \subseteq D$, is said to be *translatable* within a dataset, D, if and only if there exists a vector, v, such that $P + v \subseteq D$. Given a vector, v, then the *maximal translatable pattern* (MTP) for v in the dataset, D, is defined and denoted as follows:

$$\mathrm{MTP}(v, D) = \{p \mid p \in D \land p + v \in D\} \, , \tag{13.3}$$

where $p + v$ is the point that results when p is translated by the vector, v. In other words, the MTP for a vector, v, in a dataset, D, is the set of points in D that can be translated by v to give other points that are also in D. Figure 13.2 shows some examples of maximal translatable patterns.

13.2.2 Translational Equivalence Classes (TECs)

When analysing a piece of music, we typically want to find *all the occurrences* of an interesting pattern, not just one occurrence. Thus, if we believe that MTPs are related in some way to the patterns that listeners and analysts find interesting, then we want to be able to find all the occurrences of each MTP. Given a pattern, P, in a dataset, D, the *translational equivalence class* (TEC) of P in D is defined and denoted as follows:

$$\mathrm{TEC}(P, D) = \{Q \mid Q \equiv_{\mathrm{T}} P \land Q \subseteq D\} \, . \tag{13.4}$$

That is, the TEC of a pattern, P, in a dataset contains all and only those patterns in the dataset that are translationally equivalent to P. Note that $P \equiv_{\mathrm{T}} P$, so $P \in \mathrm{TEC}(P, D)$. Figure 13.3 shows some examples of TECs.

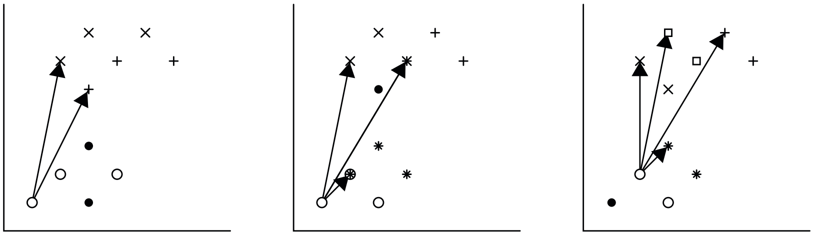

Fig. 13.3 Examples of translational equivalence classes (TECs). In each graph, the pattern of circles is translatable by the vectors indicated by the arrows. The TEC of each pattern of circles is the set of patterns containing the circle pattern itself along with the other patterns generated by translating the circle pattern by the vectors indicated. The covered set of each TEC is the set of points denoted by icons other than filled black dots

The *covered set* of a TEC, T, denoted by $\mathrm{COV}(T)$, is defined to be the union of the patterns in the TEC, T. That is,

$$\mathrm{COV}(T) = \bigcup_{P \in T} P \,. \tag{13.5}$$

Here, we will be particularly concerned with *MTP TECs*—that is, the translational equivalence classes of the maximal translatable patterns in a dataset.

Suppose we have a TEC, $T = \mathrm{TEC}(P,D)$, in a k-dimensional dataset, D. T contains the patterns in D that are translationally equivalent to P. Suppose T contains n translationally equivalent occurrences of the pattern, P, and that P contains m points. There are at least two ways in which one can specify T. First, one can explicitly define each of the n patterns in T by listing the m points in each pattern. This requires one to write down mn, k-dimensional points or kmn numbers. Alternatively, one can explicitly list the m points in just one of the patterns in T (e.g., P) and then give the $n-1$ vectors required to translate this pattern onto its other occurrences in the dataset. This requires one to write down m, k-dimensional points and $n-1$, k-dimensional vectors—that is, $k(m+n-1)$ numbers. If n and m are both greater than one, then $k(m+n-1)$ is less than kmn, implying that the second method of specifying a TEC gives us a *compressed* encoding of the TEC. Thus, if a dataset contains at least two non-intersecting occurrences of a pattern containing at least two points, it will be possible to encode the dataset in a compact manner by representing it as the union of the covered sets of a set of TECs, where each TEC, T, is encoded as an ordered pair, $\langle P,V \rangle$, where P is a pattern in the dataset, and V is the set of vectors that translate P onto its other occurrences in the dataset. When a TEC, $T = \langle P,V \rangle$, is represented in this way, we call V the *set of translators* for the TEC and P the TEC's *pattern*. We also denote and define the *compression factor* of a TEC, $T = \langle P,V \rangle$, as follows:

$$\mathrm{CF}(T) = \frac{|\mathrm{COV}(T)|}{|P| + |V|} \,, \tag{13.6}$$

where $|X|$ denotes the cardinality of set, X.[2] In this chapter, the pattern, P, of a TEC, used to encode it as a $\langle P, V\rangle$ pair, will be assumed to be the lexicographically earliest occurring member of the TEC (i.e., the one that contains the lexicographically earliest point).[3]

13.2.3 Approximate Versus Exact Matching

Each of the algorithms described in the next section takes a dataset as input, computes the MTPs in this dataset and may then go on to compute the TECs of these MTPs. Several of the algorithms generate compact encodings of the input dataset in the form of sets of selected TECs that collectively cover the input dataset. Two of these algorithms, COSIATEC (see Sect. 13.3.3) and SIATECCOMPRESS (see Sect. 13.3.7), generate *losslessly* compressed encodings from which the input dataset can be perfectly reconstructed. These encodings can therefore be interpreted as *explanations* for the input dataset that offer an account for *every note*, reflecting the assumption that, in a well-composed piece of music, notes will not be selected at random by the composer, but rather chosen carefully on the grounds of various aesthetic, expressive and structural reasons. Consequently, if one is interested in learning how to compose music in the style of some master composer, then one aims to understand the reasoning underlying the selection of *every note* in pieces in the form and style that one wishes to imitate. To do this, merely identifying instances in these pieces where patterns approximately resemble each other is not enough. One must also attempt to formulate precise explanations for the *differences* (however small) between these approximately matching patterns, so that one understands *exactly* how and why the patterns are transformed in the way they are. In other words, one needs to *exactly* characterize the transformations that map a pattern onto its various occurrences—even when the transformation is not simply a shift in time accompanied by a modal or chromatic transposition. This implies that, if one's goal is to achieve an understanding of a set of pieces that is complete and detailed enough to allow for high-quality novel pieces to be composed in the same style and form, then one requires *losslessly* compressed encodings of the existing pieces, based on *exact* matches (or, more generally, exactly characterized transformations) between pattern occurrences.

Nevertheless, if one's goal is only to produce an analysis that informs the listening process, or if one only needs to be able to classify existing pieces by genre, composer,

[2] In order to conform to standard usage in the data compression literature (see, e.g., Salomon and Motta, 2010, p. 12), in this chapter, the term "compression factor" is used to signify the quantity that I and other authors in this area have referred to as "compression ratio" in previous publications (e.g., Collins et al., 2011; Meredith et al., 2002).

[3] A collection of strings or tuples is sorted into *lexicographical* order, when the elements are sorted as they would be in a dictionary, with higher priority being given to elements occurring earlier. For example, if we lexicographically sort the set of points $\{\langle 1,0\rangle, \langle 0,1\rangle, \langle 1,1\rangle, \langle 0,0\rangle\}$, then we get the ordered set, $\langle\langle 0,0\rangle, \langle 0,1\rangle, \langle 1,0\rangle, \langle 1,1\rangle\rangle$.

period, etc., then approximate matching and lossily compressed encodings may suffice. In such cases, there are a number of ways in which the algorithms considered in this chapter can be adapted to account for occurrences of a pattern that are not exact transpositions in chromatic-pitch-time space. That is, there are a number of ways in which we might address what Collins et al. (2013) call the "inexactness problem" with the SIA-based algorithms. First, instead of using a chromatic-pitch vs. time representation as input, one can use some other point-set representation of a piece, such as morphetic-pitch vs. time, where translationally equivalent patterns correspond to sets of notes related by modal rather than chromatic transposition (e.g., C-D-E would match with D-E-F). Second, one can replace each occurrence of an MTP in the output of one of the algorithms described below with either the shortest segment in the music or bounding box in pitch-time space that contains the MTP (as was done in the experiment reported in Sect. 13.4.2 below). Third, Collins et al. (2013) address this "inexactness problem" in their SIARCT-CFP algorithm by combining their SIACT and SIAR algorithms (see Sects. 13.3.5 and 13.3.6, respectively) with a fingerprinting technique that computes a time-stamped hash key for each triple of notes in a dataset and then matches these keys (see Chap. 17, this volume).

13.3 The Algorithms

In this section, the algorithms evaluated in this study will be briefly described and reviewed.

13.3.1 SIA

SIA stands for "Structure Induction Algorithm" (Meredith, 2006a; Meredith et al., 2002, 2003, 2001). SIA finds all the maximal translatable patterns in a dataset containing n, k-dimensional points in $\Theta(kn^2 \lg n)$ time and $\Theta(kn^2)$ space.[4] SIA computes the vector from each point in the dataset to each lexicographically later point. Each of these vectors is stored in a list together with a pointer back to the origin point (see Fig. 13.4 (a) and (b)). This list of $\langle \text{vector}, \text{point} \rangle$ pairs is then sorted into lexicographical order, giving priority to the vector in each pair (see Fig. 13.4 (c)). The resulting sorted list is segmented at the points at which the vector changes (as indicated by the boxes in the column headed "Datapoint" in Fig. 13.4 (c)). The set of points in the entries within a segment in this list form the MTP for the vector for that segment. This means that all the MTPs can be obtained simply by scanning this list once (i.e., in $\Theta(kn^2)$ time, since the list has length $n(n-1)/2$).

[4] By using hashing instead of sorting to partition the inter-point vectors, the average running time can be reduced to $\Theta(kn^2)$. For an explanation of asymptotic notation, see Cormen et al. (2009, pp. 43–53)

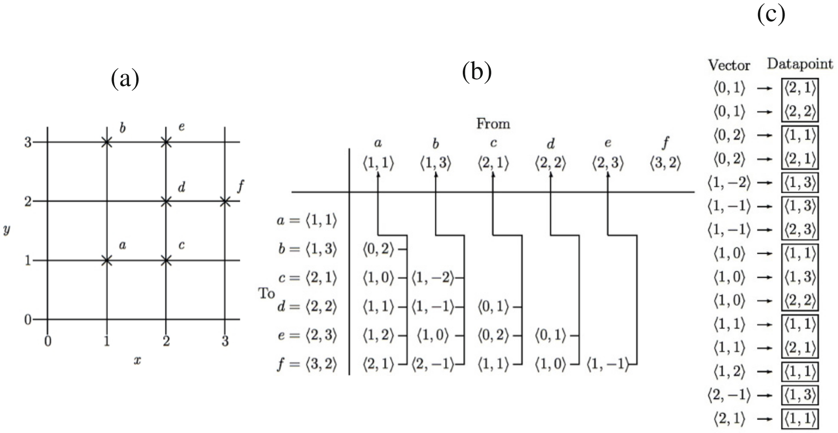

Fig. 13.4 The SIA algorithm. (a) A small dataset that could be provided as input to SIA. (b) The vector table computed by SIA for the dataset in (a). Each entry in the table gives the vector from a point to a lexicographically later point. Each entry has a pointer back to the origin point used to compute the vector. (c) The list of ⟨vector, point⟩ pairs that results when the entries in the vector table in (b) are sorted into lexicographical order. If this list is segmented at points at which the vector changes, then the set of points in the entries within a segment form the MTP for the vector for that segment. (Reproduced from Meredith et al. (2003))

Figure 13.5 gives pseudocode for a straightforward implementation of SIA. The pseudocode used in this chapter is based on that used by Meredith (2006b, 2007). In this pseudocode, unordered sets are denoted by italic upper-case letters (e.g., D in Fig. 13.5). Ordered sets are denoted by boldface upper-case letters (e.g., $\mathbf{V}$, $\mathbf{D}$ and $\mathbf{M}$ in Fig. 13.5). When written out in full, ordered sets are listed between angle brackets, "$\langle\cdot\rangle$". Concatenation is denoted by "$\oplus$" and the assignment operator is "$\leftarrow$". $\mathbf{A}[i]$ denotes the $(i+1)$th element of the ordered set (or one-dimensional array), $\mathbf{A}$, (i.e., zero-based indexing is used). If $\mathbf{B}$ is an ordered set of ordered sets (or a two-dimensional array), then $\mathbf{B}[i][j]$ denotes the $(j+1)$th element in the $(i+1)$th element of $\mathbf{B}$. Elements in arrays of higher dimension are indexed analogously. Block structure is indicated by indentation alone.

The SIA algorithm in Fig. 13.5 takes a single argument, D, which is assumed to be an unordered set of n, k-dimensional points. The first step is to sort the points in D into lexicographical order (line 1) to give an ordered set of points, $\mathbf{D}$. This takes $\Theta(kn\lg n)$ time in the worst case using a comparison sort. In lines 2–5, the vector, $\mathbf{D}[j]-\mathbf{D}[i]$, from each point, $\mathbf{D}[i]$, to each lexicographically later point, $\mathbf{D}[j]$, is calculated and stored with the index, i, of the origin point in a pair, $\langle\mathbf{D}[j]-\mathbf{D}[i],i\rangle$. Each of these pairs is added to an ordered set, $\mathbf{V}$, in line 5. In line 6, the pairs in $\mathbf{V}$ are sorted lexicographically (i.e., by assigning higher priority to the vector in each pair, and by sorting the vectors themselves lexicographically). This produces a list, $\mathbf{V}'$, such as the one shown in Fig. 13.4 (c), in which all pairs with a given vector occur in a contiguous segment. Line 6 takes $\Theta(kn^2\lg n)$ time and is asymptotically the most

```
SIA($D$)
1   $\mathbf{D} \leftarrow \textsc{Sort}_{\text{Lex}}(D)$
2   $\mathbf{V} \leftarrow \langle\rangle$
3   for $i \leftarrow 0$ to $|D|-2$
4       for $j \leftarrow i+1$ to $|D|-1$
5           $\mathbf{V} \leftarrow \mathbf{V} \oplus \langle\langle \mathbf{D}[j]-\mathbf{D}[i], i\rangle\rangle$
6   $\mathbf{V}' \leftarrow \textsc{Sort}_{\text{Lex}}(\mathbf{V})$
7   $\mathbf{M} \leftarrow \langle\rangle$
8   $v \leftarrow \mathbf{V}'[0][0]$
9   $\mathbf{P} \leftarrow \langle \mathbf{D}[\mathbf{V}'[0][1]]\rangle$
10  for $i \leftarrow 1$ to $|\mathbf{V}'|-1$
11      if $\mathbf{V}'[i][0] = v$
12          $\mathbf{P} \leftarrow \mathbf{P} \oplus \langle \mathbf{D}[\mathbf{V}'[i][1]]\rangle$
13      else
14          $\mathbf{M} \leftarrow \mathbf{M} \oplus \langle\langle \mathbf{P}, v\rangle\rangle$
15          $v \leftarrow \mathbf{V}'[i][0]$
16          $\mathbf{P} \leftarrow \langle \mathbf{D}[\mathbf{V}'[i][1]]\rangle$
17  $\mathbf{M} \leftarrow \mathbf{M} \oplus \langle\langle \mathbf{P}, v\rangle\rangle$
18  return $\mathbf{M}$
```

Fig. 13.5 The SIA algorithm

expensive step in the algorithm. In line 7, an empty ordered set, **M**, is initialized. This ordered set will be used to hold the $\langle$MTP, vector$\rangle$ pairs computed in lines 8–17. As described above, the MTPs and their associated vectors are found simply by scanning $\mathbf{V}'$ once, starting a new MTP (stored in **P**) each time the vector (stored in v) changes. This scan can be accomplished in $\Theta(kn^2)$ time.

The algorithm can easily be modified so that it only generates MTPs whose sizes lie within a particular user-specified range. It is also possible for the same pattern to be the MTP for more than one vector. If this is the case, there will be two or more $\langle$pattern, vector$\rangle$ pairs in the output of SIA that have the same pattern. This can be avoided and the output can be made more compact by generating instead a list of $\langle$pattern, vector set$\rangle$ pairs, such that the vector set in each pair contains all the vectors for which the pattern is an MTP. In order to accomplish this, the vectors for which a given pattern is the MTP are merged into a single vector set which is then paired with the pattern in the output.

Finally, it is possible to reduce the average running time of SIA to $\Theta(kn^2)$ by using hashing instead of sorting to partition the vector table into MTPs.

13.3.2 SIATEC

SIATEC (Meredith, 2006a; Meredith et al., 2002, 2003, 2001) computes all the MTP TECs in a k-dimensional dataset of size n in $O(kn^3)$ time and $\Theta(kn^2)$ space. In order to find the MTPs in a dataset, SIA only needs to compute the vectors from each point to each lexicographically later point. However, to compute *all occurrences* of the MTPs, it turns out to be beneficial in the SIATEC algorithm to compute the vectors between *all* pairs of points, resulting in a vector table like the one shown in Fig. 13.6. The SIATEC algorithm first finds all the MTPs using SIA. It then uses the vector table to find all the vectors by which each MTP is translatable within the dataset. The set of vectors by which a given pattern is translatable is equal to the intersection of the columns in the vector table headed by the points in the pattern (see Fig. 13.6). In a vector table computed by SIATEC each row descends lexicographically from left to right and each column increases lexicographically from top to bottom. SIATEC exploits these properties of the vector table to more efficiently find all the occurrences of each MTP (Meredith et al., 2002, pp. 335–338).

Figure 13.7 shows pseudocode for a straightforward implementation of SIATEC. The first step in the algorithm is to sort the points in the dataset, D, lexicographically, to produce the ordered point set, **D** (line 1). In line 2, an ordered set, **V** is initialized which serves the same purpose as it does in SIA (see Fig. 13.5). In line 3, the $|D| \times |D|$ array, **W**, is initialized. This array will be used to hold a vector table like the one in Fig. 13.6. Lines 4–9 compute the inter-point vectors that are stored in **W** and **V** (cf. lines 3–5 in Fig. 13.5). Lines 10–24 compute the MTPs and closely resemble lines 6–17 in Fig. 13.5. The difference is that, in SIATEC, each MTP is stored with an ordered set, **C**, containing the indices into the sorted dataset, **D**, of the points in the MTP. This sequence of indices is used in lines 25–48 to compute the translator set of the TEC for each MTP. The ordered set of MTP TECs, **T**, is computed in lines 25–48 and returned in line 49. The **for** loop starting in line 26 iterates over the list of MTPs, **M**. For each MTP, **P** (line 27), and its associated index set, **C** (line 28), the **while** loop in lines 33–47 finds all the vectors by which **P** can be translated, by computing the intersection of the columns in the vector table **W** indexed by the values in **C**. This is done by descending the columns in the table and starting and stopping these descents neither earlier nor later than necessary, respectively.

For a k-dimensional dataset of size n, the worst-case running time of line 1 of SIATEC is $\Theta(kn \lg n)$. Lines 2–9 run in $\Theta(kn^2)$ time. Line 10 takes $\Theta(kn^2 \lg n)$ time and lines 11–24 take $\Theta(kn^2)$ time. Let m_i be the cardinality of the $(i+1)$th MTP in **M** at line 25. The $(i+1)$th iteration of the **for** loop that starts in line 26 computes the intersection of the columns in the vector table, **W**, that are headed by the points in m_i. Each column has length n. Therefore the worst-case running time of lines 26–48

		From					
		$a = \langle 1,1\rangle$	$b = \langle 1,3\rangle$	$c = \langle 2,1\rangle$	$d = \langle 2,2\rangle$	$e = \langle 2,3\rangle$	$f = \langle 3,2\rangle$
To	$a = \langle 1,1\rangle$	$\langle 0,0\rangle$	$\langle 0,-2\rangle$	$\langle -1,0\rangle$	$\langle -1,-1\rangle$	$\langle -1,-2\rangle$	$\langle -2,-1\rangle$
	$b = \langle 1,3\rangle$	$\langle 0,2\rangle$	$\langle 0,0\rangle$	$\langle -1,2\rangle$	$\langle -1,1\rangle$	$\langle -1,0\rangle$	$\langle -2,1\rangle$
	$c = \langle 2,1\rangle$	$\langle 1,0\rangle$	$\langle 1,-2\rangle$	$\langle 0,0\rangle$	$\langle 0,-1\rangle$	$\langle 0,-2\rangle$	$\langle -1,-1\rangle$
	$d = \langle 2,2\rangle$	$\langle 1,1\rangle$	$\langle 1,-1\rangle$	$\langle 0,1\rangle$	$\langle 0,0\rangle$	$\langle 0,-1\rangle$	$\langle -1,0\rangle$
	$e = \langle 2,3\rangle$	$\langle 1,2\rangle$	$\langle 1,0\rangle$	$\langle 0,2\rangle$	$\langle 0,1\rangle$	$\langle 0,0\rangle$	$\langle -1,1\rangle$
	$f = \langle 3,2\rangle$	$\langle 2,1\rangle$	$\langle 2,-1\rangle$	$\langle 1,1\rangle$	$\langle 1,0\rangle$	$\langle 1,-1\rangle$	$\langle 0,0\rangle$

Fig. 13.6 The vector table computed by SIATEC for the dataset shown in Fig. 13.4 (a). (Reproduced from Meredith et al. (2003))

```
SIATEC($D$)
    $\mathbf{D} \leftarrow \textsc{Sort}_{\mathrm{Lex}}(D)$
    $\mathbf{V} \leftarrow \langle\rangle$
    Allocate a $|D| \times |D|$ array, $\mathbf{W}$
    for $i \leftarrow 0$ to $|D| - 1$
        for $j \leftarrow 0$ to $|D| - 1$
            $w \leftarrow \langle \mathbf{D}[j] - \mathbf{D}[i], i \rangle$
            if $j > i$
                $\mathbf{V} \leftarrow \mathbf{V} \oplus \langle w \rangle$
            $\mathbf{W}[i][j] \leftarrow w$
    $\mathbf{V}' \leftarrow \textsc{Sort}_{\mathrm{Lex}}(\mathbf{V})$
    $\mathbf{M} \leftarrow \langle\rangle$
    $v \leftarrow \mathbf{V}'[0][0]$
    $\mathbf{P} \leftarrow \langle \mathbf{D}[\mathbf{V}'[0][1]] \rangle$
    $\mathbf{C} \leftarrow \langle \mathbf{V}'[0][1] \rangle$
    for $i \leftarrow 1$ to $|\mathbf{V}'| - 1$
        if $\mathbf{V}'[i][0] = v$
            $\mathbf{P} \leftarrow \mathbf{P} \oplus \langle \mathbf{D}[\mathbf{V}'[i][1]] \rangle$
            $\mathbf{C} \leftarrow \mathbf{C} \oplus \langle \mathbf{V}'[i][1] \rangle$
        else
            $\mathbf{M} \leftarrow \mathbf{M} \oplus \langle \langle \mathbf{P}, \mathbf{C}, v \rangle \rangle$
            $v \leftarrow \mathbf{V}'[i][0]$
            $\mathbf{P} \leftarrow \langle \mathbf{D}[\mathbf{V}'[i][1]] \rangle$
            $\mathbf{C} \leftarrow \langle \mathbf{V}'[i][1] \rangle$
    $\mathbf{M} \leftarrow \mathbf{M} \oplus \langle \langle \mathbf{P}, \mathbf{C}, v \rangle \rangle$
    $\mathbf{T} \leftarrow \langle\rangle$
    for $i \leftarrow 0$ to $|\mathbf{M}| - 1$
        $\mathbf{P} \leftarrow \mathbf{M}[i][0]$
        $\mathbf{C} \leftarrow \mathbf{M}[i][1]$
        $\mathbf{R} \leftarrow \langle 0 \rangle$
        for $j \leftarrow 1$ to $|\mathbf{P}| - 1$
            $\mathbf{R} \leftarrow \mathbf{R} \oplus \langle 0 \rangle$
        $\mathbf{X} \leftarrow \langle\rangle$
        while $\mathbf{R}[0] \leq |D| - |\mathbf{P}|$
            for $j \leftarrow 1$ to $|\mathbf{P}| - 1$
                $\mathbf{R}[j] \leftarrow \mathbf{R}[0] + j$
            $v_0 \leftarrow \mathbf{W}[\mathbf{C}[0]][\mathbf{R}[0]][0]$
            $found \leftarrow$ false
            for $c \leftarrow 1$ to $|\mathbf{P}| - 1$
                while $\mathbf{R}[c] < |D| \wedge \mathbf{W}[\mathbf{C}[c]][\mathbf{R}[c]][0] < v_0$
                    $\mathbf{R}[c] \leftarrow \mathbf{R}[c] + 1$
                if $\mathbf{R}[c] \geq |D| \vee v_0 \neq \mathbf{W}[\mathbf{C}[c]][\mathbf{R}[c]][0]$
                    break
                if $c = |\mathbf{P}| - 1$
                    $found \leftarrow$ true
            if $found \vee |\mathbf{P}| = 1$
                $\mathbf{X} \leftarrow \mathbf{X} \oplus \langle v_0 \rangle$
            $\mathbf{R}[0] \leftarrow \mathbf{R}[0] + 1$
        $\mathbf{T} \leftarrow \mathbf{T} \oplus \langle \langle \mathbf{P}, \mathbf{X} \rangle \rangle$
    return $\mathbf{T}$
```

Fig. 13.7 The SIATEC algorithm

is given by

$$\Theta\left(\sum_{i=0}^{|\mathbf{M}|-1} knm_i\right) = \Theta\left(kn\sum_{i=0}^{|\mathbf{M}|-1} m_i\right) .$$

But we know that $\sum_{i=0}^{|\mathbf{M}|-1} m_i = n(n-1)/2$, since the MTPs are computed in lines 10–24 by scanning the sorted list, $\mathbf{V}'$. Therefore, the worst-case running time of lines 26–48 and the worst-case running time of SIATEC as a whole, is

$$\Theta\left(kn\sum_{i=0}^{|\mathbf{M}|-1} m_i\right) = \Theta\left(kn\frac{n(n-1)}{2}\right) = \Theta(kn^3) .$$

It is easy to see that the algorithm uses $\Theta(kn^2)$ space.

13.3.3 COSIATEC

COSIATEC (Meredith, 2006a; Meredith et al., 2003) is a greedy point-set compression algorithm, based on SIATEC ("COSIATEC" stands for "COmpression with SIATEC"). COSIATEC takes a dataset, D, as input and computes a compressed encoding of D in the form of an ordered set of MTP TECs, $\mathbf{T}$, such that $D = \bigcup_{T\in\mathbf{T}} \mathrm{COV}(T)$ and $\mathrm{COV}(T_1) \cap \mathrm{COV}(T_2) = \emptyset$ for all $T_1, T_2 \in \mathbf{T}$ where $T_1 \neq T_2$. In other words, COSIATEC strictly partitions a dataset, D, into the covered sets of a set of MTP TECs. If each of these MTP TECs is represented as a $\langle$pattern, translator set$\rangle$ pair, then this description of the dataset as a set of TECs is typically shorter than an in extenso description in which the points in the dataset are listed explicitly.

Figure 13.8 shows pseudocode for COSIATEC. The first step in the algorithm is to make a copy of the input dataset which is stored in the variable D' (line 1). Then, on each iteration of the **while** loop (lines 3–6), the algorithm finds the "best" MTP TEC in D', stores this in T and adds T to $\mathbf{T}$. It then removes the set of points covered by T from D' (line 6). When D' is empty, the algorithm terminates, returning the list of MTP TECs, $\mathbf{T}$. The sum of the number of translators and the number of points in

```
COSIATEC($D$)
1  $D' \leftarrow$ COPY($D$)
2  $\mathbf{T} \leftarrow \langle\rangle$
3  while $D' \neq \emptyset$
4     $T \leftarrow$ GETBESTTEC($D', D$)
5     $\mathbf{T} \leftarrow \mathbf{T} \oplus \langle T \rangle$
6     $D' \leftarrow D' \setminus \mathrm{COV}(T)$
7  return $\mathbf{T}$
```

Fig. 13.8 The COSIATEC algorithm

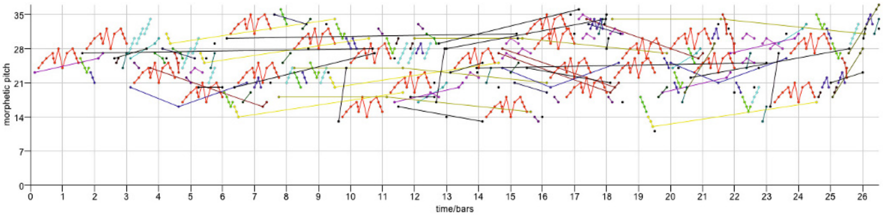

Fig. 13.9 A visualization of the output generated by COSIATEC for the fugue from J. S. Bach's Prelude and Fugue in C major (BWV 846) from the first book of *Das wohltemperirte Clavier*. Note that, for convenience, the time axis has been labelled in bars rather than tatums. See main text for details

this output encoding is never more than the number of points in the input dataset and can be much less than this, if there are many repeated patterns in the input dataset.

The GETBESTTEC function, called in line 4 of COSIATEC, computes the "best" TEC in D' by first finding all the MTPs using SIA, then iterating over these MTPs, finding the TEC for each MTP, and storing it if it is the best TEC so far. In this process, a TEC is considered "better" than another if it has a higher compression factor, as defined in (13.6). If two TECs have the same compression factor, then the better TEC is considered to be the one that has the higher *bounding-box compactness* (Meredith et al., 2002), defined as the ratio of the number of points in the TEC's pattern to the number of dataset points in the bounding box of this pattern. Collins et al. (2011) have provided empirical evidence that the compression factor and compactness of a TEC are important factors in determining its perceived "importance" or "noticeability". If two distinct TECs have the same compression factor and compactness, then the TEC with the larger covered set is considered superior.

Figure 13.9 shows a visualization of the output generated by COSIATEC for the fugue from J. S. Bach's Prelude and Fugue in C major (BWV 846) from the first book of *Das wohltemperirte Clavier*. In this figure, each TEC is drawn in a different colour. For example, occurrences of the subject (minus the first note) are shown in red. Points drawn in grey are elements of the *residual point set*. COSIATEC sometimes generates such a point set on the final iteration of its **while** loop (see Fig. 13.8) when it is left with a set of points that does not contain any MTPs of size greater than 1. In such cases, the final TEC in the encoding contains just the single occurrence of the residual point set. In the analysis shown in Fig. 13.9, the residual point set contains 26 notes (i.e., 3.57% of the notes in the piece). The overall compression factor achieved by COSIATEC on this piece is 2.90 (or 3.12, if we exclude the residual point set).

13.3.4 Forth's Algorithm

Forth (Forth and Wiggins, 2009; Forth, 2012) developed an algorithm, based on SIATEC, that, like COSIATEC, computes a set of TECs that collectively cover the

```
FORTHCOVER($D, \mathbf{C}, \mathbf{W}, c_{\min}, \sigma_{\min}$)
  $\mathbf{S} \leftarrow \langle\rangle$
  $P \leftarrow \emptyset$
  $found \leftarrow$ true
  while $P \neq D \wedge found$
    $found \leftarrow$ false
    $\gamma_{\max} \leftarrow 0$
    $C_{\mathrm{best}} \leftarrow$ nil
    $i_{\mathrm{best}} \leftarrow$ nil
    $\mathbf{R} \leftarrow \langle\rangle$
    for $i \leftarrow 0$ to $|\mathbf{C}| - 1$
      $c \leftarrow |\mathbf{C}[i] \setminus P|$
      if $c < c_{\min}$
        $\mathbf{R} \leftarrow \mathbf{R} \oplus \langle i \rangle$
        continue
      $\gamma \leftarrow c\mathbf{W}[i]$
      if $\gamma > \gamma_{\max}$
        $\gamma_{\max} \leftarrow \gamma$
        $C_{\mathrm{best}} \leftarrow \mathbf{C}[i]$
        $i_{\mathrm{best}} \leftarrow i$
    if $C_{\mathrm{best}} \neq$ nil
      $\mathbf{R} \leftarrow \mathbf{R} \oplus \langle i_{\mathrm{best}} \rangle$
      $found \leftarrow$ true
      $P \leftarrow P \cup C_{\mathrm{best}}$
      $i \leftarrow 0$
      $primaryFound \leftarrow$ false
      while $\neg primaryFound \wedge i < |\mathbf{S}|$
        if $(|\mathbf{S}[i][0] \cap C_{\mathrm{best}}|)/|\mathbf{S}[i][0]| > \sigma_{\min}$
          $\mathbf{S}[i] \leftarrow \mathbf{S}[i] \oplus \langle C_{\mathrm{best}} \rangle$
          $primaryFound \leftarrow$ true
        $i \leftarrow i + 1$
      if $\neg primaryFound$
        $\mathbf{S} \leftarrow \mathbf{S} \oplus \langle\langle C_{\mathrm{best}} \rangle\rangle$
    for each $i \in \mathbf{R}$
      Remove $\mathbf{W}[i]$ from $\mathbf{W}$ and $\mathbf{C}[i]$ from $\mathbf{C}$
  return $\mathbf{S}$
```

Fig. 13.10 An implementation of Forth's method for selecting a cover for the input dataset, D, from the set of TEC covered sets, $\mathbf{C}$

input dataset. However, unlike COSIATEC, Forth's algorithm runs SIATEC only once and the covers it generates are not, in general, strict partitions—the TECs in the list generated by Forth's algorithm may share covered points and, collectively, may not completely cover the input dataset. The first step in Forth's algorithm is to run SIATEC on the input dataset, D. This generates a sequence of MTP TECs, $\mathbf{T} = \langle T_1, T_2, \ldots T_n \rangle$. The algorithm then computes the covered set, $C_i = \bigcup_{P \in T_i} P$, for each TEC, T_i in $\mathbf{T}$, to produce a sequence of TEC covered sets, $\mathbf{C} = \langle C_1, C_2, \ldots C_n \rangle$. It then assigns a weight, W_i, to each covered set, C_i, to produce the sequence, $\mathbf{W} = \langle W_1, W_2, \ldots W_n \rangle$. W_i is intended to measure the "structural salience" (Forth, 2012,

p. 41) of the patterns in the TEC, T_i, and it is defined as follows:

$$W_i = w'_{\mathrm{cr},i} w'_{\mathrm{compV},i} , \tag{13.7}$$

where $w'_{\mathrm{cr},i}$ is a normalized value representing the compression factor of T_i and $w'_{\mathrm{compV},i}$ is a normalized value representing compactness. The algorithm then attempts to select a subset of **C** that covers the input dataset and maximizes the weights of the TECs in this cover.

Figure 13.10 gives pseudocode for Forth's cover selection algorithm. This algorithm takes five arguments: D, **C** and **W**, as defined above, along with two numerical parameters, $c_{\min}$ and $\sigma_{\min}$. On each iteration of the outer **while** loop (lines 4–34), the algorithm selects the "best" remaining TEC covered set in **C** and adds this to the cover, **S**, which is ultimately returned in line 35. The point set, P, initialized in line 2, is used to store the set of points covered by the TEC covered sets selected. In order for a TEC covered set to be added to the cover, the number of *new* points that it covers, c, (i.e., that are not already in P) must be at least $c_{\min}$ (see line 12). The TEC covered set that is added on a particular iteration of the **while** loop is then the one for which cW_i is a maximum (lines 15–19). If no TEC covered set is selected on a particular iteration, then the algorithm terminates, even if the dataset has not been completely covered. In lines 20–32, the algorithm determines whether the selected TEC covered set, C_{best} should be added as a "primary" or a "secondary" pattern: a pattern C_{s} is defined to be secondary to another (primary) pattern, C_{p}, if the proportion of points in C_{p} that are also in C_{s} is greater than the parameter, $\sigma_{\min}$ (line 27) (Forth, 2012, p. 38). On each iteration of the **while** loop, the best pattern and patterns that fail to cover a sufficient number of new points are removed from **C** to improve efficiency (lines 33–34).

Figure 13.11 shows the analysis generated by Forth's algorithm for the fugue from the Prelude and Fugue in C major (BWV 846) from the first book of J. S. Bach's *Das wohltemperirte Clavier*. Again, each TEC is drawn in a different colour. This analysis was obtained with $c_{\min}$ set to 15 and $\sigma_{\min}$ set to 0.5, as recommended by Forth (2012, pp. 38, 42).

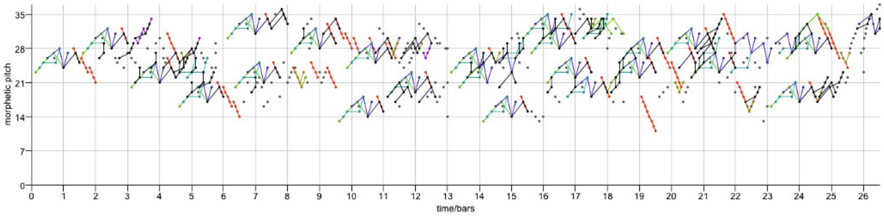

Fig. 13.11 A visualization of the output generated by Forth's algorithm for the fugue from J. S. Bach's Prelude and Fugue in C major (BWV 846) from the first book of *Das wohltemperirte Clavier*. For this analysis, $c_{\min}$ was set to 15 and $\sigma_{\min}$ was set to 0.5

13.3.5 SIACT

Collins et al. (2010) claim that all the algorithms described above can be affected by what they call the "problem of isolated membership". This 'problem' is defined to occur when "a musically important pattern is contained *within* an MTP, along with other temporally isolated members that may or may not be musically important" (Collins et al., 2010, p. 6). Collins et al. claim that "the larger the dataset, the more likely it is that the problem will occur" and that it could prevent the SIA-based algorithms from "discovering some translational patterns that a music analyst considers noticeable or important" (Collins et al., 2010, p. 6). Collins et al. propose that this problem can be solved by taking each MTP computed by SIA (sorted into lexicographical order) and "trawling" inside this MTP "from beginning to end, returning subsets that have a compactness greater than some threshold a and that contain at least b points" (Collins et al., 2010, p. 6). This method is implemented in an algorithm that they call SIACT, which first runs SIA on the dataset and then carries out "compactness trawling" (hence "SIA*CT*") on each of the MTPs found by SIA.

Suppose $\mathbf{P} = \langle p_1, p_2, \ldots p_m \rangle$ contains all and only the points in an MTP, sorted into lexicographical order. Suppose further that $i_{\mathrm{L}}(p_i)$ is the index of p_i in the lexicographically sorted dataset and that $\mathbf{I}_{\mathrm{L}}(\mathbf{P}) = \langle i_{\mathrm{L}}(p_1), i_{\mathrm{L}}(p_2), \ldots i_{\mathrm{L}}(p_m) \rangle$. For each MTP discovered by SIA, SIACT 'trawls' the MTP for lexicographically compact subsets using the CT algorithm shown in Fig. 13.12. Given $\mathbf{P}$, $\mathbf{I}_{\mathrm{L}}(\mathbf{P})$ and the thresholds, a and b, as just defined, CT first sets up two variables, X and Q (lines 1–2 in Fig. 13.12). X is used to store the compact subsets of $\mathbf{P}$ found by the algorithm and is eventually returned in line 18. Q is used to store each of these subsets as it is found. The algorithm then scans $\mathbf{P}$ in lexicographical order, considering a new point, $\mathbf{P}[i]$, with dataset index $\mathbf{I}_{\mathrm{L}}(\mathbf{P})[i]$, on each iteration of the **for** loop in lines 3–15. On each iteration of this **for** loop, if Q is empty, then $\mathbf{P}[i]$ is added to Q and the dataset index of $\mathbf{P}[i]$ is stored in the variable $i_{\mathrm{L},0}$ (lines 4–6). Otherwise, the compactness of the pattern that would result if $\mathbf{P}[i]$ were added to Q is calculated and, if this compactness is not less than a, then $\mathbf{P}[i]$ is added to Q (lines 8–10). If the compactness of the resulting pattern would be less than a, then the algorithm checks whether Q holds at least b points and, if it does, Q is added to X and reset to the empty set (lines 11–13). If $|Q|$ is less than b, then it is discarded (lines 14–15). When the algorithm has finished scanning all the points in $\mathbf{P}$, it is possible that Q contains a pattern of at least b points. If this is the case, then it is also added to X (lines 16–17) before the latter is returned in line 18.

A problem with the CT algorithm is that the patterns found for a particular MTP depend on the order in which points in the MTP are scanned, and there seems to be no good musical or psychological reason why this should be so—why should the decision as to whether a given member point is considered 'isolated' depend on the order in which the points in the pattern are scanned? Suppose, for example, that $\mathbf{P} = \langle q_5, q_8, q_9, q_{10}, q_{11} \rangle$, where q_i is (lexicographically) the ith point in the dataset. Now suppose that $a = 2/3$ and $b = 3$, as Collins et al. (2010, p. 7) suggest. If $\mathbf{P}$ is scanned from left to right, as it would be by the CT algorithm, then the only compact pattern trawled is $\{q_8, q_9, q_{10}, q_{11}\}$, indicating that the first point, q_5 is an

```
CT($\mathbf{P}, \mathbf{I}_{\mathrm{L}}(\mathbf{P}), a, b$)
  $X \leftarrow \emptyset$
  $Q \leftarrow \emptyset$
  for $i \leftarrow 0$ to $|\mathbf{P}| - 1$
     if $Q = \emptyset$
        $Q \leftarrow Q \cup \{\mathbf{P}[i]\}$
        $i_{\mathrm{L},0} \leftarrow \mathbf{I}_{\mathrm{L}}(\mathbf{P})[i]$
     else
        $s \leftarrow \mathbf{I}_{\mathrm{L}}(\mathbf{P})[i] - i_{\mathrm{L},0} + 1$
        if $(|Q| + 1)/s \geq a$
           $Q \leftarrow Q \cup \{\mathbf{P}[i]\}$
        else if $|Q| \geq b$
           $X \leftarrow X \cup \{Q\}$
           $Q \leftarrow \emptyset$
        else
           $Q \leftarrow \emptyset$
  if $|Q| \geq b$
     $X \leftarrow X \cup \{Q\}$
  return $X$
```

Fig. 13.12 The SIACT compactness trawler

'isolated member'. However, if $\mathbf{P}$ were scanned from right to left, then the found pattern would be $\{q_5, q_8, q_9, q_{10}, q_{11}\}$—that is, $\mathbf{P}$ would not be considered to suffer from the 'problem of isolated membership'. Similarly, if the pattern to be trawled were $\mathbf{P}_2 = \langle q_8, q_9, q_{10}, q_{11}, q_{14} \rangle$, then clearly q_{14} in this pattern is no less 'isolated' (lexicographically) than q_5 in $\mathbf{P}$. However, the CT algorithm would remove q_5 from $\mathbf{P}$, but not q_{14} from $\mathbf{P}_2$. In other words, the CT algorithm would judge $\mathbf{P}$ but not $\mathbf{P}_2$ to suffer from the 'problem of isolated membership'.

Figure 13.13 shows the output obtained with COSIATEC for the fugue from BWV 846 when SIA is replaced by SIACT, with the parameters a and b set to 0.67 and 3, respectively, as suggested by Collins et al. (2010, p. 7). This modification

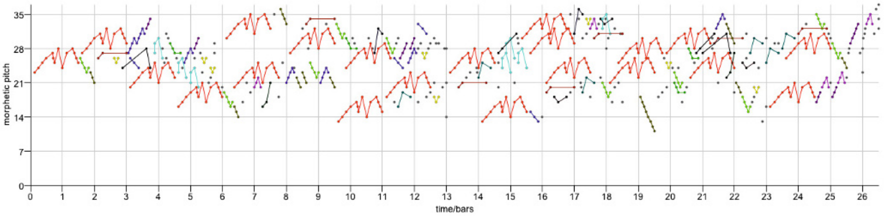

Fig. 13.13 A visualization of the output generated by the COSIATEC algorithm, with SIA replaced by SIACT, for the fugue from J. S. Bach's Prelude and Fugue in C major (BWV 846) from the first book of *Das wohltemperirte Clavier*. This analysis was obtained with parameter a set to 0.67 and parameter b set to 3

reduces the overall compression factor achieved from 2.90 to 2.53 and increases the size of the residual point set from 26 to 118 (16.19% of the dataset). With this residual point set excluded, the compression factor achieved over the remaining points is 3.59. Note that, when modified in this way, the encoding generated by COSIATEC includes the full subject, as opposed to the subject without the first note, discovered by the unmodified version of COSIATEC (see Fig. 13.9). On the other hand, this analysis using SIACT fails to find a pattern corresponding to the bass entry of the subject that starts $\frac{3}{4}$ of the way through bar 17.

13.3.6 SIAR

In an attempt to improve on the precision and running time of SIA, Collins (2011, pp. 282–283) defines an SIA-based algorithm called SIAR (which stands for "SIA for *r* superdiagonals"). Instead of computing the whole region below the leading diagonal in the vector table for a dataset (as in Fig. 13.4(b)), SIAR only computes the first *r* subdiagonals of this table. This is approximately equivalent to running SIA with a sliding window of size *r* (Collins, 2011; Collins et al., 2010). Pseudocode for SIAR is provided in Fig. 13.14, based on Collins's (2013c) own implementation.

The first step in SIAR is to calculate the first *r* subdiagonals of SIA's vector table (lines 1–7). This results in a list, $\mathbf{V}$, of $\langle v, i \rangle$ pairs, each representing the vector, v, from point p_i to p_j, where p_k is the $(k+1)$th point in $\mathbf{D}$, the lexicographically sorted input dataset. The next step in SIAR is to extract the (not necessarily maximal) translatable patterns discoverable from $\mathbf{V}$. This is done using the same technique as used in SIA: $\mathbf{V}$ is first sorted into lexicographical order and then segmented at points where the vector changes (lines 8–17). This produces an ordered set, $\mathbf{E}$, of patterns. SIA is then applied to each of the patterns in $\mathbf{E}$. However, the MTPs found by SIA are not required, so, for each pattern, $\mathbf{e} \in \mathbf{E}$, only the positive inter-point vectors for $\mathbf{e}$ are computed and these are stored in a list, $\mathbf{L}$ (lines 18–23). SIAR then produces a new list of vectors, $\mathbf{M}$, by removing duplicates in $\mathbf{L}$ and sorting the vectors into decreasing order by frequency of occurrence (lines 24–34). The final step in SIAR is to find the MTP for each of the vectors in $\mathbf{M}$, which, in Collins's (2013c) own implementation, is achieved using the method in lines 35–37 in Fig. 13.14. In line 37, the MTP for the vector $\mathbf{M}[i][0]$ is found by translating the input dataset, D, by the inverse of this vector and then finding the intersection between the resulting point set and the original dataset, D.

Figure 13.15 shows the analysis generated by COSIATEC for the fugue from BWV 846 when SIA is replaced by SIAR, with the parameter *r* set to 3. This modification reduces the overall compression factor from 2.90 to 2.41 and increases the size of the residual point set from 26 to 38 (5.21% of the dataset). If we exclude this residual point set, the compression factor over the remaining points is 2.61. Note that, if compactness trawling is enabled in this modified version of COSIATEC, then the output for this particular piece is identical to that shown in Fig. 13.13—that is, when SIA is replaced by SIACT.

```
SIAR$(D, r)$
    ▷ Sort the dataset into lexicographical order
    $\mathbf{D} \leftarrow \textsc{Sort}_{\text{Lex}}(D)$
    ▷ Compute $r$ subdiagonals of vector table and store in $\mathbf{V}$
    $\mathbf{V} \leftarrow \langle\rangle$
    for $i \leftarrow 0$ to $|D| - 2$
        $j \leftarrow i + 1$
        while $j < |D| \wedge j \leq i + r$
            $\mathbf{V} \leftarrow \mathbf{V} \oplus \langle\langle \mathbf{D}[j] - \mathbf{D}[i], i\rangle\rangle$
            $j \leftarrow j + 1$
    ▷ Store patterns in $\mathbf{E}$ by sorting and segmenting $\mathbf{V}$
    $\mathbf{V} \leftarrow \textsc{Sort}_{\text{Lex}}(\mathbf{V})$
    $\mathbf{E} \leftarrow \langle\rangle$
    $v \leftarrow \mathbf{V}[0][0]$
    $\mathbf{e} \leftarrow \langle \mathbf{D}[\mathbf{V}[0][1]]\rangle$
    for $i \leftarrow 1$ to $|\mathbf{V}| - 1$
        if $\mathbf{V}[i][0] = v$
            $\mathbf{e} \leftarrow \mathbf{e} \oplus \langle \mathbf{D}[\mathbf{V}[i][1]]\rangle$
        else
            $\mathbf{E} \leftarrow \mathbf{E} \oplus \langle \mathbf{e}\rangle$, $\mathbf{e} \leftarrow \langle \mathbf{D}[\mathbf{V}[i][1]]\rangle$, $v \leftarrow \mathbf{V}[i][0]$
    $\mathbf{E} \leftarrow \mathbf{E} \oplus \langle \mathbf{e}\rangle$
    ▷ For each pattern in $\mathbf{E}$, find $+$ve inter-point vectors and store in $\mathbf{L}$
    $\mathbf{L} \leftarrow \langle\rangle$
    for $i \leftarrow 0$ to $|\mathbf{E}| - 1$
        $\mathbf{e} \leftarrow \mathbf{E}[i]$
        for $j \leftarrow 0$ to $|\mathbf{e}| - 2$
            for $k \leftarrow j + 1$ to $|\mathbf{e}| - 1$
                $\mathbf{L} \leftarrow \mathbf{L} \oplus \langle \mathbf{e}[k] - \mathbf{e}[j]\rangle$
    ▷ Remove duplicates from $\mathbf{L}$ and order vectors by decreasing frequency of occurrence
    $\mathbf{L} \leftarrow \textsc{Sort}_{\text{Lex}}(\mathbf{L})$
    $v \leftarrow \mathbf{L}[0]$
    $f \leftarrow 1$
    $\mathbf{M} \leftarrow \langle\rangle$
    for $i \leftarrow 1$ to $|\mathbf{L}| - 1$
        if $\mathbf{L}[i] = v$
            $f \leftarrow f + 1$
        else
            $\mathbf{M} \leftarrow \mathbf{M} \oplus \langle\langle v, f\rangle\rangle$, $f \leftarrow 1$, $v \leftarrow \mathbf{L}[i]$
    $\mathbf{M} \leftarrow \mathbf{M} \oplus \langle\langle v, f\rangle\rangle$
    $\mathbf{M} \leftarrow \textsc{SortDescendingByFreq}(\mathbf{M})$
    ▷ Find the MTP for each vector in $\mathbf{M}$, store it in $\mathbf{S}$ and return $\mathbf{S}$
    $\mathbf{S} \leftarrow \langle\rangle$
    for $i \leftarrow 0$ to $|\mathbf{M}| - 1$
        $\mathbf{S} \leftarrow \mathbf{S} \oplus \langle D \cap (D - \mathbf{M}[i][0])\rangle$
    return $\mathbf{S}$
```

Fig. 13.14 Pseudocode for SIAR, based on Collins's (2013c) own implementation

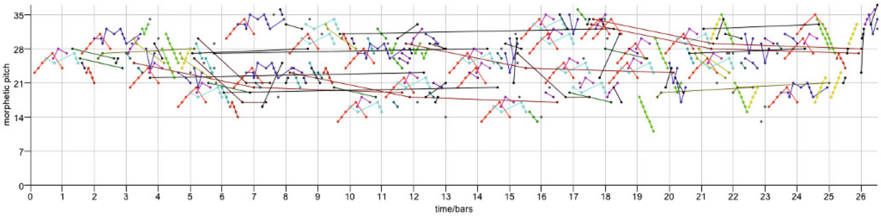

Fig. 13.15 A visualization of the output generated by the COSIATEC algorithm, with SIA replaced by SIAR, for the fugue from J. S. Bach's Prelude and Fugue in C major (BWV 846) from the first book of *Das wohltemperirte Clavier*. This analysis was obtained with the parameter r set to 3

13.3.7 SIATECCompress

One of the actions carried out by the GETBESTTEC function, called in line 4 of COSIATEC (see Fig. 13.8), is to run SIATEC on the set, D', which contains what remains of the input dataset, D, on each iteration of the **while** loop. Since SIATEC has worst-case running time $\Theta(n^3)$ where n is the number of points in the input dataset, running COSIATEC on large datasets can be time-consuming. On the other hand, because COSIATEC strictly partitions the dataset into non-overlapping MTP TEC covered sets, it tends to achieve high compression factors for many point-set representations of musical pieces (typically between 2 and 4 for a piece of classical or baroque music).

```
SIATECCOMPRESS($D$)
$\mathbf{T} \leftarrow$ SIATEC($D$)
$\mathbf{T} \leftarrow$ SORTTECSBYQUALITY($\mathbf{T}$)
$D' \leftarrow \emptyset$
$\mathbf{E} \leftarrow \langle\rangle$
for $i \leftarrow 0$ to $|\mathbf{T}| - 1$
    $T \leftarrow \mathbf{T}[i]$
    $S \leftarrow$ COV($T$)
    ▶ Recall that each TEC, $T$, is an ordered pair, $\langle$pattern, translator set$\rangle$
    if $|S \setminus D'| > |T[0]| + |T[1]|$
        $\mathbf{E} \leftarrow \mathbf{E} \oplus \langle T \rangle$
        $D' \leftarrow D' \cup S$
        if $|D'| = |D|$
            break
$R \leftarrow D \setminus D'$
if $|R| > 0$
    $\mathbf{E} \leftarrow \mathbf{E} \oplus \langle$ASTEC($R$)$\rangle$
return $\mathbf{E}$
```

Fig. 13.16 The SIATECCOMPRESS algorithm

Like COSIATEC, the SIATECCompress algorithm shown in Fig. 13.16 is a greedy compression algorithm based on SIATEC that computes an encoding of a dataset in the form of a union of TEC covered sets. Like Forth's algorithm (but unlike COSIATEC), SIATECCompress runs SIATEC only *once* (line 1) to get a list of TECs. This list is then sorted into decreasing order by quality (line 2), where the decision as to which of any two TECs is superior is made in the same way as in COSIATEC (described above). The algorithm then finds a compact encoding, **E**, of the dataset in the form of a set of TECs. It does this by iterating over the sorted list of TECs (lines 5–12), adding a new TEC, T, to **E** if the number of new points covered by T is greater than the size of its $\langle \text{pattern}, \text{translator set} \rangle$ representation (lines 8–12). Each time a TEC, T, is added to **E**, its covered set is added to the set D', which therefore maintains the set of points covered so far after each iteration. When D' is equal to D or all the TECs have been scanned, the **for** loop terminates. Any remaining uncovered points are aggregated into a residual point set, R, (line 13) which is re-expressed as a TEC with an empty translator set (line 15) and added to the encoding, **E**. SIATECCompress does not generally produce as compact an encoding as COSIATEC, since the TECs in its output may share points. However, it is faster than COSIATEC and can therefore be used practically on much larger datasets. Like COSIATEC, but unlike Forth's algorithm, SIATECCompress always produces a complete cover of the input dataset (although the last TEC in the encoding may consist of just a single residual point set).

Figure 13.17 shows the analysis generated by SIATECCompress for the fugue from BWV 846. The overall compression factor for this analysis is 1.94 compared with the value of 2.90 obtained using COSIATEC. The residual point set in this case contains 82 notes (11.25%) (cf. 26 notes, 3.57% for COSIATEC) and the compression factor excluding this residual point set is 2.20 (compared with 3.12 for COSIATEC).

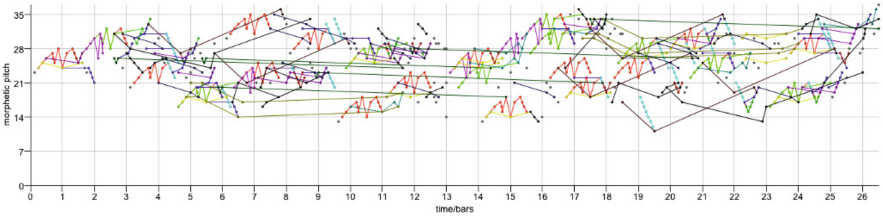

Fig. 13.17 A visualization of the output generated by the SIATECCompress algorithm, for the fugue from J. S. Bach's Prelude and Fugue in C major (BWV 846) from the first book of *Das wohltemperirte Clavier*

13.4 Evaluation

The algorithms described above were evaluated on two musicological tasks: classifying folk song melodies into tune families and discovering repeated themes and sections in polyphonic classical works. The results obtained will now be presented and discussed.

13.4.1 Task 1: Classifying Folk Song Melodies into Tune Families

For over a century, musicologists have been interested in measuring similarity between folk song melodies (Scheurleer, 1900; van Kranenburg et al., 2013), primarily with the purpose of classifying such melodies into *tune families* (Bayard, 1950), consisting of tunes that have a common ancestor in the tree of oral transmission. In the first evaluation task, COSIATEC, Forth's algorithm and SIATECCompress were used to classify a collection of folk song melodies into tune families. The collection used was the *Annotated Corpus* (van Kranenburg et al., 2013; Volk and van Kranenburg, 2012) of 360 melodies from the Dutch folk song collection, *Onder de groene linde* (Grijp, 2008), hosted by the Meertens Institute and accessible through the website of the Dutch Song Database (http://www.liederenbank.nl). The algorithms were used as compressors to calculate the *normalized compression distance* (NCD) (Li et al., 2004) between each pair of melodies in the collection (see (13.1)). Each melody was then classified using the 1-nearest-neighbour algorithm with leave-one-out cross-validation. The classifications obtained were compared with a ground-truth classification of the melodies provided by expert musicologists. Four versions of each algorithm were tested: the basic algorithm as described above, a version incorporating the CT algorithm (Fig. 13.12), a version using SIAR instead of SIA and a version using both SIAR and CT. As a baseline, one of the best general-purpose compression algorithms, BZIP2, was also used to calculate NCDs between the melodies.

Table 13.1 shows the results obtained in this task. In this table, algorithms with names containing "R" employed the SIAR algorithm with $r = 3$ in place of SIA. The value of 3 for r was chosen so as to be small, as the higher the value of r, the more SIAR approximates to SIA. Collins et al. (2013) ran SIAR with $r = 1$. Algorithms with names containing "CT" used Collins et al.'s (2010) compactness trawler, with parameters $a = 0.66$ and $b = 3$, chosen because these were the values suggested by Collins et al. (2010, p. 7). Forth's algorithm was run with $c_{\min} = 15$ and $\sigma_{\min} = 0.5$, as suggested by Forth (2012, pp. 38, 42). No attempt was made to find optimal values for these parameters, so overfitting is very unlikely. On the other hand, this also means that the chosen parameters are probably suboptimal. The column headed "SR" gives the classification success rate—i.e., the proportion of songs in the corpus correctly classified. The third and fourth columns give the mean compression factor achieved by each algorithm over, respectively, the corpus and the file-pairs used to compute the compression distances.

Table 13.1 Results on Task 1. *SR* is the classification success rate, CR_{AC} is the average compression factor over the melodies in the *Annotated Corpus*. CR_{pairs} is the average compression factor over the pairs of files used to obtain the NCD values

Algorithm	*SR*	CR_{AC}	CR_{pairs}
COSIATEC	0.8389	1.5791	1.6670
COSIARTEC	0.8361	1.5726	1.6569
COSIARCTTEC	0.7917	1.4547	1.5135
COSIACTTEC	0.7694	1.4556	1.5138
ForthCT	0.6417	1.1861	1.2428
ForthRCT	0.6417	1.1861	1.2428
Forth	0.6111	1.2643	1.2663
ForthR	0.6028	1.2555	1.2655
SIARCTTECCompress	0.5750	1.3213	1.3389
SIATECCompress	0.5694	1.3360	1.3256
SIACTTECCompress	0.5250	1.3197	1.3381
SIARTECCompress	0.5222	1.3283	1.3216
BZIP2	0.1250	2.7678	3.5061

The highest success rate of 84% was obtained using COSIATEC. Table 13.1 suggests that algorithms based on COSIATEC performed markedly better on this song classification task than those based on SIATECCOMPRESS or Forth's algorithm. Using SIAR instead of SIA and/or incorporating compactness trawling reduced the performance of COSIATEC. However, using both together, slightly improved the performance of SIATECCOMPRESS. Forth's algorithm performed slightly better than SIATECCOMPRESS. The performance of Forth's algorithm on this task was improved by incorporating compactness trawling; using SIAR instead of SIA in Forth's algorithm slightly reduced the performance of the basic algorithm and had no effect when compactness trawling was used. The results obtained using BZIP2 were much poorer than those obtained using the SIA-based algorithms, which may suggest that general-purpose compressors fail to capture certain musical structure that is important for this task. However, using an appropriate string-based input representation, where repeated substrings correspond to repeated segments of music, instead of a point set representation as was used here, may improve the performance of compressors such as BZIP2 that are designed primarily for text compression. Of the SIA-based algorithms, COSIATEC achieved the best compression on average, followed by SIATECCOMPRESS and then Forth's algorithm. COSIATEC also achieved the best success rate. However, since Forth's algorithm performed slightly better than SIATECCOMPRESS, it seems that compression factor alone was not a reliable indicator of classification accuracy on this task—indeed, the best compressor, BZIP2, produced the worst classifier. None of the algorithms achieved a success rate as high as the 99% obtained by van Kranenburg et al. (2013) on this corpus using several local features and an alignment-based approach. The success rate achieved by COSIATEC is within the 83–86% accuracy range obtained by Velarde et al.

(2013, p. 336) on this database using a wavelet-based representation, with similarity measured using Euclidean or city-block distance.

13.4.2 Task 2: Discovering Repeated Themes and Sections

In the second task, each of the SIA-based algorithms tested in Task 1 was used to discover repeated themes and sections in the five pieces in the JKU Patterns Development Database (JKU-PDD) (Collins, 2013a). This database contains Orlando Gibbons' madrigal, "Silver Swan" (1612); the fugue from J. S. Bach's Prelude and Fugue in A minor (BWV 889) from Book 2 of *Das wohltemperirte Clavier* (1742); the second movement of Mozart's Piano Sonata in E flat major (K. 282) (1774); the third movement of Beethoven's Piano Sonata in F minor, Op. 2, No. 1 (1795); and Chopin's Mazurka in B flat minor, Op. 24, No. 4 (1836). The database also contains encodings of ground-truth analyses by expert analysts that identify important patterns in the pieces. For each of these patterns, a ground-truth analysis encodes one or more occurrences, constituting an *occurrence set* for each of the ground-truth patterns. It is important to note, however, that each of these ground-truth occurrence sets does not necessarily contain *all* the occurrences within a piece of a particular pattern. For example, the ground-truth analyses fail to recognize that, in both of the minuet-and-trio movements by Beethoven and Mozart, the first section of the trio is recapitulated at the end of the second section in a slightly varied form. Why such occurrences have been omitted from the ground-truth is not clear.

It can also be argued that the ground-truth analyses in the JKU-PDD omit certain patterns that might reasonably be considered important or noticeable. For example, Figure 13.18 shows two patterns discovered by COSIATEC that help to account for the structure of the lyrical fourth section of the Chopin Mazurka, yet this section of the piece is completely ignored in the JKU-PDD ground truth. Again, why the

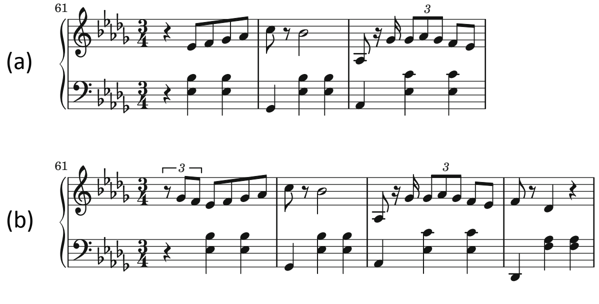

Fig. 13.18 Examples of noticeable and/or important patterns in Chopin's Mazurka in B flat minor, Op. 24, No. 4, that were discovered by COSIATEC, but are not recorded in the ground truth. Pattern (a) occurs independently of pattern (b) at bars 73 and 89

analysts whose work was used as a basis for the ground-truth should have ignored such patterns is not clear.

Each of the algorithms tested generates a set of TECs which is intended to contain all the occurrences of a particular pattern. That is, each TEC is intended to correspond to a ground-truth occurrence set. In general, in the ground-truth analyses, an "occurrence" of a pattern is not necessarily exactly translationally equivalent to it in pitch-time space. For example, it is common in the ground-truth analyses of the JKU-PDD for patterns to be specified by just the segments of the music that span them; or, if the pattern occurs entirely within a single voice, by the segment of that voice that spans the pattern. On the other hand, it is not uncommon for an MTP to contain only *some* of the notes within the shortest segment of the music or smallest rectangle in pitch-time space that contains it, resulting in it having a compactness less than 1. In such cases, a ground-truth pattern may be equal to the shortest segment containing an MTP or an MTP's bounding-box, rather than equal to the MTP itself. In this task, therefore, each of the 12 algorithms was tested on the JKU-PDD in three different "modes": "Raw" mode, "Segment" mode and "BB" mode. In "Raw" mode, the pattern occurrence sets generated are simply the "raw" TECs computed by the algorithm (see Fig. 13.19(a)). In "BB" mode, each raw pattern occurrence is replaced in the output with the pattern containing all the points in the bounding-box of the pattern (Fig. 13.19(b)). In "Segment" mode, each raw pattern occurrence is replaced with the pattern containing all the points in the temporal segment spanning the pattern (Fig. 13.19(c)). The results of this task are given in Table 13.2. The values in this table are *three-layer F_1 scores* (TLF1), as defined by Meredith (2013). Each value gives the harmonic mean of the precision and recall of the algorithm on a given piece. Three-layer F_1 score is a modification of the standard F_1 score that gives credit to an algorithm for discovering a pattern or an occurrence that is very similar but not identical to a ground-truth pattern. If the standard definition of F_1 score is used in this task, then an algorithm may score 0 even if each pattern that it generates differs from a ground-truth pattern by only one note. Three-layer F_1 score overcomes

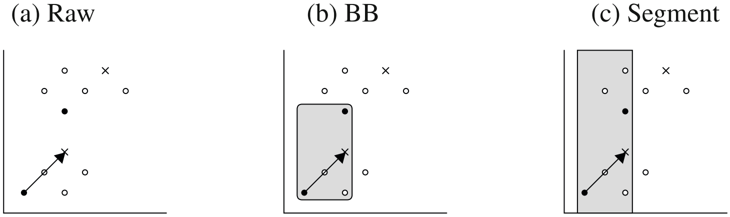

Fig. 13.19 (a) The pattern consisting of two black dots is a "raw" pattern, as might be output by one of the SIA-based algorithms. This pattern is the MTP for the vector indicated by the arrow. The pattern of crosses is the image of the black dot pattern after translation by this vector. (b) The shaded area indicates the corresponding bounding-box pattern output by the algorithm when operating in "BB" mode. (c) The shaded area indicates the corresponding segment pattern output when the algorithm operates in "Segment" mode

this problem by using F_1 score (or, equivalently, the Sørensen–Dice index (Dice, 1945; Sørensen, 1948)) to measure similarity on three levels of structure: between individual occurrences, between occurrence sets and between sets of occurrence sets. In Table 13.2, the highest values for each piece are in bold. The last column gives the mean TLF1 value over all the pieces for a given algorithm. The algorithms are named using the same convention as used in Table 13.1, except that, in addition, the mode (Raw, Segment or BB) is appended to the name.

Overall, the best performing algorithms on this task were COSIATEC and SIATECCompress in "Segment" mode. These were the best-performing algorithms on the Chopin, Gibbons and Beethoven pieces. Using SIAR instead of SIA in these algorithms in "Segment" mode did not change the overall mean performance, but

Table 13.2 Results on Task 2. Values are "three-layer F1" values, as defined by Meredith (2013)

Algorithm	Chopin	Gibbons	Beethoven	Mozart	Bach	Mean
COSIACTTEC	0.09	0.16	0.22	0.29	0.23	0.20
COSIACTTECBB	0.18	0.24	0.42	0.40	0.22	0.29
COSIACTTECSegment	0.25	0.31	0.56	0.45	0.19	0.35
COSIARCTTEC	0.09	0.16	0.22	0.29	0.23	0.20
COSIARCTTECBB	0.18	0.24	0.42	0.40	0.22	0.29
COSIARCTTECSegment	0.25	0.31	0.56	0.45	0.19	0.35
COSIARTEC	0.05	0.12	0.14	0.18	0.20	0.14
COSIARTECBB	0.22	0.31	0.49	0.38	0.22	0.32
COSIARTECSegment	**0.44**	**0.39**	**0.69**	0.55	0.19	**0.45**
COSIATEC	0.05	0.11	0.18	0.23	0.19	0.15
COSIATECBB	0.17	0.23	0.51	0.46	0.21	0.32
COSIATECSegment	0.37	0.37	**0.71**	**0.60**	0.18	**0.45**
Forth	0.12	0.33	0.32	0.21	0.17	0.23
ForthBB	0.18	0.27	0.32	0.27	0.17	0.24
ForthCT	0.23	0.35	0.56	0.56	**0.40**	0.42
ForthCTBB	0.27	0.34	0.57	0.58	**0.39**	0.43
ForthCTSegment	0.29	0.35	0.58	**0.59**	0.31	0.42
ForthR	0.12	0.30	0.20	0.25	0.30	0.23
ForthRBB	0.18	0.25	0.29	0.31	0.28	0.26
ForthRCT	0.23	0.35	0.56	0.56	**0.40**	0.42
ForthRCTBB	0.27	0.34	0.57	0.58	**0.39**	0.43
ForthRCTSegment	0.29	0.35	0.58	**0.59**	0.31	0.42
ForthRSegment	0.28	0.25	0.35	0.38	0.27	0.31
ForthSegment	0.33	0.26	0.35	0.33	0.19	0.29
SIACTTECCompress	0.12	0.20	0.23	0.30	0.27	0.22
SIACTTECCompressBB	0.17	0.27	0.30	0.35	0.27	0.27
SIACTTECCompressSegment	0.20	0.26	0.33	0.38	0.22	0.28
SIARCTTECCompress	0.12	0.20	0.23	0.30	0.27	0.22
SIARCTTECCompressBB	0.17	0.27	0.30	0.35	0.27	0.27
SIARCTTECCompressSegment	0.20	0.26	0.33	0.38	0.22	0.28
SIARTECCompress	0.10	0.18	0.19	0.18	0.25	0.18
SIARTECCompressBB	0.39	0.32	0.53	0.45	0.26	0.39
SIARTECCompressSegment	**0.60**	**0.39**	**0.65**	0.57	0.25	**0.49**
SIATECCompress	0.11	0.16	0.19	0.25	0.26	0.19
SIATECCompressBB	0.37	0.30	0.51	0.51	0.29	0.40
SIATECCompressSegment	**0.56**	**0.40**	0.63	**0.59**	0.29	**0.49**

did change the performance on individual pieces. The highest score obtained on any single piece was 0.71 by COSIATECSEGMENT on the Beethoven sonata movement. Interestingly, Forth's algorithm was the best-performing algorithm on the Bach fugue by a considerable margin. This suggests that there may be some feature of this algorithm that makes it particularly suited to analysing imitative contrapuntal music.

13.5 Conclusions

Each of the algorithms considered in this study takes a point-set representation of a piece of music as input and computes a set of TECs that collectively cover (or almost cover) this point set. All the algorithms attempt to select TECs in a way that maximizes compression factor and compactness. The results obtained on two quite different evaluation tasks suggest that this geometric, compression-based approach has the potential to lead to versatile algorithms that derive analyses from in extenso music representations that can profitably be used in a variety of musicological tasks. However, the results also indicate that certain variants of the algorithms may be more suited to some tasks than to others. The results do not unambiguously support the hypothesis that the best analyses of a piece correspond to the shortest possible descriptions of it. However, COSIATEC, the SIA-based algorithm that achieves the best compression in general, was the best-performing algorithm on Task 1 and achieved the second-best score overall on Task 2.

Acknowledgements The work reported in this chapter was partly carried out while working on a collaborative European project entitled "Learning to Create" (Lrn2Cre8). The project Lrn2Cre8 acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET grant number 610859.

The author would also like to thank Tom Collins and two anonymous reviewers for their helpful and constructive feedback on an earlier version of this chapter.

Supplementary Material The source code of the Java implementations of the algorithms described in this chapter that were used in the evaluation tasks are freely available at http://chromamorph.googlecode.com. For further information or advice on running or modifying these implementations, readers should contact the author.

References

Bayard, S. (1950). Prolegomena to a study of the principal melodic families of British-American folk song. *Journal of American Folklore*, 63(247):1–44.

Bent, I. (1987). *Analysis*. The New Grove Handbooks in Music. Macmillan. (Glossary by W. Drabkin).

Chaitin, G. J. (1966). On the length of programs for computing finite binary sequences. *Journal of the Association for Computing Machinery*, 13(4):547–569.

Collins, T. (2011). *Improved methods for pattern discovery in music, with applications in automated stylistic composition*. PhD thesis, Faculty of Mathematics, Computing and Technology, The Open University, Milton Keynes.

Collins, T. (2013a). JKU Patterns Development Database. Available at https://dl.dropbox.com/u/11997856/JKU/JKUPDD-Aug2013.zip.

Collins, T. (2013b). MIREX 2013 Competition: Discovery of Repeated Themes and Sections. http://tinyurl.com/o9227qg. Accessed on 5 January 2015.

Collins, T. (2013c). PattDisc-Jul2013. Available online at http://www.tomcollinsresearch.net/publications.html. Accessed 29 December 2013.

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

Collins, T., Laney, R., Willis, A., and Garthwaite, P. H. (2011). Modeling pattern importance in Chopin's Mazurkas. *Music Perception*, 28(4):387–414.

Collins, T., Thurlow, J., Laney, R., Willis, A., and Garthwaite, P. H. (2010). 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)*, pages 3–8, Utrecht, The Netherlands.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, 3rd edition.

Dice, L. R. (1945). Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302.

Forth, J. and Wiggins, G. A. (2009). An approach for identifying salient repetition in multidimensional representations of polyphonic music. In Chan, J., Daykin, J. W., and Rahman, M. S., editors, *London Algorithmics 2008: Theory and Practice*, pages 44–58. College Publications.

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

Grijp, L. P. (2008). Introduction. In Grijp, L. P. and van Beersum, I., editors, *Under the Green Linden—163 Dutch Ballads from the Oral Tradition*, pages 18–27. Meertens Institute/Music & Words.

Kolmogorov, A. N. (1965). Three approaches to the quantitative definition of information. *Problems of Information Transmission*, 1(1):1–7.

Lerdahl, F. and Jackendoff, R. S. (1983). *A Generative Theory of Tonal Music*. MIT Press.

Li, M., Chen, X., Li, X., Ma, B., and Vitányi, P. M. B. (2004). The similarity metric. *IEEE Transactions on Information Theory*, 50(12):3250–3264.

Li, M. and Vitányi, P. (2008). *An Introduction to Kolmogorov Complexity and Its Applications*. Springer, third edition.

Meredith, D. (2006a). Point-set algorithms for pattern discovery and pattern matching in music. In *Proceedings of the Dagstuhl Seminar on Content-based Retrieval (No. 06171, 23–28 April, 2006)*, Schloss Dagstuhl, Germany. Available online at http://drops.dagstuhl.de/opus/volltexte/2006/652.

Meredith, D. (2006b). The *ps13* pitch spelling algorithm. *Journal of New Music Research*, 35(2):121–159.

Meredith, D. (2007). *Computing pitch names in tonal music: A comparative analysis of pitch spelling algorithms*. PhD thesis, Faculty of Music, University of Oxford.

Meredith, D. (2013). Three-layer precision, three-layer recall, and three-layer F1 score. http://tinyurl.com/owtz79v.

Meredith, D., Lemström, K., and Wiggins, G. A. (2002). Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music. *Journal of New Music Research*, 31(4):321–345.

Meredith, D., Lemström, K., and Wiggins, G. A. (2003). Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music. In *Cambridge Music Processing Colloquium*, Department of Engineering, University of Cambridge. Available online at http://www.titanmusic.com/papers.php.

Meredith, D., Wiggins, G. A., and Lemström, K. (2001). Pattern induction and matching in polyphonic music and other multi-dimensional datasets. In Callaos, N., Zong, X., Verges, C., and Pelaez, J. R., editors, *Proceedings of the 5th World Multiconference on Systemics, Cybernetics and Informatics (SCI2001)*, volume X, pages 61–66.

Rissanen, J. (1978). Modeling by shortest data description. *Automatica*, 14(5):465–471.

Salomon, D. and Motta, G. (2010). *Handbook of Data Compression*. Springer, fifth edition.

Scheurleer, D. (1900). Preisfrage. *Zeitschrift der Internationalen Musikgesellschaft*, 1(7):219–220.

Simon, H. A. and Sumner, R. K. (1968). Pattern in music. In Kleinmuntz, B., editor, *Formal representation of human judgment*. Wiley.

Simon, H. A. and Sumner, R. K. (1993). Pattern in music. In Schwanauer, S. M. and Levitt, D. A., editors, *Machine Models of Music*, pages 83–110. MIT Press.

Solomonoff, R. J. (1964a). A formal theory of inductive inference (Part I). *Information and Control*, 7(1):1–22.

Solomonoff, R. J. (1964b). A formal theory of inductive inference (Part II). *Information and Control*, 7(2):224–254.

Sørensen, T. (1948). A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on Danish commons. *Kongelige Danske Videnskabernes Selskab*, 5(4):1–34.

van Kranenburg, P., Volk, A., and Wiering, F. (2013). A comparison between global and local features for computational classification of folk song melodies. *Journal of New Music Research*, 42(1):1–18.

Velarde, G., Weyde, T., and Meredith, D. (2013). An approach to melodic segmentation and classification based on filtering with the Haar-wavelet. *Journal of New Music Research*, 42(4):325–345.

Volk, A. and van Kranenburg, P. (2012). Melodic similarity among folk songs: An annotation study on similarity-based categorization in music. *Musicae Scientiae*, 16(3):317–339.