USING SIATECCOMPRESS TO DISCOVER REPEATED THEMES AND SECTIONS IN POLYPHONIC MUSIC

David Meredith
Aalborg University
dave@titanmusic.com

ABSTRACT

Three variants of the SIATECCOMPRESS algorithm were submitted to the 2016 MIREX competition on Discovery of Repeated Themes and Sections. The three variants were those that performed best in terms of three-layer F_1 score (TLF1), three-layer precision (TLP) and three-layer recall (TLR), respectively, when a large number of variants of SIATECCOMPRESS, COSIATEC and Forth's algorithm were run on the *polyphonic* version of the JKU Patterns Development Database (JKU-PDD). The variant optimised for TLF1 achieved an average TLF1 score of 0.490 over the five pieces in the database. The variant optimised for TLP achieved an average TLP score of 0.561 over the JKU-PDD. The variant optimised for TLR achieved an average TLR score of 0.583 over the JKU-PDD. The three variants have been implemented in Java.

1. INTRODUCTION

COSIATEC [8, 11–16, 18, 19], SIATECCOMPRESS [12–16], and Forth's algorithm [6, 7] are greedy point-set compression algorithms that have been developed for analysing music. All three algorithms are based on the SIA and SIATEC algorithm described by Meredith, Lemström and Wiggins [17]. Each algorithm takes a point-set representation of a musical piece as input and computes a compact encoding of the piece in the form of a set of *translational equivalence classes* of *maximal translatable patterns*. COSIATEC generates a strict partitioning of the input dataset, whereas the sets of pattern occurrences computed by SIATECCOMPRESS and Forth's algorithm may share points (i.e., notes).

All three algorithms are founded on the hypothesis that the best ways of understanding a piece of music are those that are represented by the shortest descriptions of the piece. In other words, they are designed to explore the notion that music analysis is effectively just music compression.

In previous experiments [12, 15, 16], it was found that, of the three algorithms, versions of SIATECCOMPRESS performed best on the task of discovering repeated themes and sections in both the JKU Patterns Development Database (JKU-PDD) [2] and the test database used in the 2013 and 2014 MIREX competitions [3, 4]. Three variants of the SIATECCOMPRESS algorithm were therefore submitted to the 2016 competition, optimized for precision, recall and F1 score, respectively.

2. USING POINT SETS TO REPRESENT MUSIC

The submitted algorithms assumed that the piece of music to be analysed is represented in the form of a multi-dimensional point set called a *dataset*, as described by Meredith *et al.* [17]. Although these algorithms work with datasets of any dimensionality, it will be assumed here that each dataset is a set of two-dimensional points, $\langle t, p\rangle$, where each point represents a single note or sequence of tied notes whose onset time is t in tatums and whose *morphetic pitch* [9, 10, 17] is p. If morphetic pitch information is not available (e.g., because the data is in MIDI format), then (at least for Western tonal music) it can be very reliably computed from chromatic pitch (i.e., MIDI note number) using an algorithm such as PS13s1 [9, 10].

The three variants of SIATECCOMPRESS submitted to the 2016 MIREX competition on Discovery of Repeated Themes and Sections take Collins' "lisp" format files as input and use morphetic pitch representation.

3. MAXIMAL TRANSLATABLE PATTERNS

I shall use the term *pattern* to refer to any subset of a dataset. Suppose D is a dataset and $D'_1, D'_2 \subseteq D$. The two patterns, D'_1, D'_2, are said to be *translationally equivalent*, denoted by $D'_1 \equiv_{\mathrm{T}} D'_2$, if and only if there exists a vector v, such that D'_1 translated by v is equal to D'_2. That is,

$$D'_1 \equiv_{\mathrm{T}} D'_2 \iff (\exists v \mid D'_2 = D'_1 + v)\,. \qquad (1)$$

Given a vector, v, then the *maximal translatable pattern* (MTP) for v in the dataset, D, is defined and denoted as follows:

$$\mathrm{MTP}(v, D) = \{p \mid p \in D \wedge p + v \in D\} \qquad (2)$$

where $p + v$ is the point that results when one translates p 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.

The notion that COSIATEC, SIATECCOMPRESS and Forth's algorithm can be used to discover the patterns in a piece of music that an analyst or a listener finds important, is founded upon the hypothesis that these patterns correspond in some way to MTPs in the pitch-time dataset representation of the piece. Meredith *et al.* [17] describe an algorithm called SIA for discovering all the MTPs in a dataset.

4. TRANSLATIONAL EQUIVALENCE CLASSES

When analysing a piece of music, we typically want to find *all the occurrences* of an interesting pattern, not just one occurrence. Given a pattern, D', in a dataset, D, the *translational equivalence class* (TEC) of D' in D is defined and denoted as follows:

$$\mathrm{TEC}(D', D) = \{Q \mid Q \equiv_{\mathrm{T}} P \wedge Q \subseteq D\} \,. \qquad (3)$$

We can also define the *covered set* of a TEC, T, denoted by $\mathrm{COV}(T)$, to be the union of the occurrences in the TEC. That is,

$$\mathrm{COV}(T) = \bigcup_{D' \in T} D' \,. \qquad (4)$$

Here we will be particularly concerned with *MTP TECs*—that is, the translational equivalence classes of the maximal translatable patterns in a dataset. Meredith *et al.* [17] describe an algorithm called SIATEC that uses SIA to find all the MTPs and then goes on to find the TEC of each of these MTPs (i.e., it finds all the (exact) occurrences of all the MTPs).

A TEC is a set of patterns that are all translationally equivalent to each other. Suppose a TEC, T, contains n occurrences of a pattern containing m points. There are at least two ways in which one can specify T. First, one can list each of the n occurrences in T explicitly by listing all of the m points in each occurrence. This requires one to write down mn 2-dimensional points or $2mn$ integers. Alternatively, one can explicitly list the m points in just one of the n occurrences, D', and then give the $n-1$ vectors required to map D' onto the other occurrences. This requires one to write down m 2-dimensional points and $n-1$, 2-dimensional vectors—that is, $2(m+n-1)$ integers. If n and m are both greater than one, then $2(m+n-1)$ is less than $2mn$, implying that the second method of specifying a TEC gives us a *compressed* encoding of the TEC (and therefore also of its covered set). Thus, in principle, if a dataset contains repeated (i.e., translationally equivalent) patterns, it may 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, $\langle D', V\rangle$, where D' is one occurrence in T and V is the set of vectors that map D' onto the other occurrences in T. When a TEC, $T = \langle D', V\rangle$, is represented in this way, we call D' the *pattern* and V the *translator set* of the TEC.

```
COSIATEC(D)
1   D' ← COPY(D)
2   T* ← nil
3   T ← ⟨⟩
4   while D' ≠ ∅
5     T* ← GETBESTTEC(D', D)
6     T ← T ⊕ ⟨T*⟩
7     D' ← D' \ COV(T*)
8   return T
```

Figure 1. The COSIATEC algorithm.

5. THE COSIATEC ALGORITHM

COSIATEC [8,18] (see Figure 1) is a greedy compression algorithm, based on SIATEC, that 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

$$D = \bigcup_{T \in \mathbf{T}} \mathrm{COV}(T) \qquad (5)$$

and, for all $T_1, T_2 \in \mathbf{T}, T_1 \neq T_2$,

$$\mathrm{COV}(T_1) \cap \mathrm{COV}(T_2) = \emptyset \,. \qquad (6)$$

In other words, COSIATEC 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 simply listed explicitly.

COSIATEC begins by making a copy of the input dataset which it stores in the variable D' (line 1). Then, on each iteration of the **while** loop (lines 4–7), the algorithm finds the "best" MTP TEC in D', T^*, appends this TEC to $\mathbf{T}$ and then removes the set of points covered by T^* from D'. 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 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.

Given an input dataset, D, and what remains of a copy, D', of this dataset after the removal of zero or more MTP TEC covered sets, the COSIATEC algorithm finds the "best" MTP TEC in D' (line 5), using the GETBESTTEC algorithm shown in Figure 2. In lines 1–2 of GETBESTTEC, the SIA algorithm is used to find all the MTPs in the dataset. The first step in this process is to compute a so-called *vector table*, $\mathbf{V}$, which is a two-dimensional array of ordered triples,

$$\mathbf{V}[i][j] = \langle p_i - p_j, p_j, j\rangle \,,$$

where $p_i - p_j$ is the vector from point p_j to p_i and $p_k = \mathbf{D}'[k]$, where $\mathbf{D}'$ is an ordered set that only contains every element in D', sorted into lexicographical order.

Having computed the vector table, $\mathbf{V}$, the MTPs are found by sorting the triples in $\mathbf{V}$, lexicographically by their vectors (i.e., their first elements), and then scanning this

```
GETBESTTEC(D′, D)
1   V ← COMPUTEVECTORTABLE(D′)
2   MCPs ← COMPUTEMTPCISPAIRS(V)
3   mcp ← nil
4   T* ← nil
5   for i ← 0 to |MCPs| − 1
6     mcp ← MCPs[i]
7     T ← GETTECFORMTP(mcp, V, D′)
8     T′ ← GETCONJUGATETEC(T)
9     T ← REMOVEREDUNDANTTRANSLATORS(T)
10    T′ ← REMOVEREDUNDANTTRANSLATORS(T′)
11    if T* = nil ∨ ISBETTERTEC(T, T*)
12      T* ← T
13    if ISBETTERTEC(T′, T*)
14      T* ← T′
15  return T*
```

Figure 2. The GETBESTTEC algorithm.

```
COMPUTEMTPCISPAIRS(V)
1   W ← SORTBYVECTOR(V)
2   MTPs ← ⟨⟩
3   CISs ← ⟨⟩
4   v ← W[0][0]
5   mtp ← ⟨W[0][1]⟩
6   cis ← ⟨W[0][2]⟩
7   for i ← 1 to |W| − 1
8     vpi ← W[i]
9     if vpi[0] = v
10      mtp ← mtp ⊕ ⟨vpi[1]⟩
11      cis ← cis ⊕ ⟨vpi[2]⟩
12    else
13      MTPs ← MTPs ⊕ ⟨mtp⟩
14      CISs ← CISs ⊕ ⟨cis⟩
15      mtp ← ⟨vpi[1]⟩
16      cis ← ⟨vpi[2]⟩
17      v ← vpi[0]
18  MTPs ← MTPs ⊕ ⟨mtp⟩
19  CISs ← CISs ⊕ ⟨cis⟩
20  MCPs ← ⟨⟩
21  for i ← 0 to |MTPs| − 1
22    MCPs ← MCPs ⊕ ⟨⟨MTPs[i], CISs[i]⟩⟩
23  return MCPs
```

Figure 3. The COMPUTEMTPCISPAIRS algorithm.

sorted list once: each MTP is then equal to the points associated with a run of consecutive triples with the same vector in this sorted list. This is accomplished in line 2 of GETBESTTEC using the COMPUTEMTPCISPAIRS algorithm, which is shown in Figure 3.

The COMPUTEMTPCISPAIRS algorithm (Figure 3) first sorts the triples in the vector table, $\mathbf{V}$, into increasing lexicographical order by their vectors. The resulting ordered set of triples is stored in the variable $\mathbf{W}$ (see line 1). In lines 2–19 of this algorithm, two lists are constructed, $\mathbf{MTPs}$ and $\mathbf{CISs}$. $\mathbf{MTPs}$ contains all the MTPs in the dataset, each MTP being represented as an ordered set of points in lexicographical order. $\mathbf{CISs}$ contains, for each MTP, a list of the indices of the columns in the vector table corresponding to the points in the MTP. In lines 20–22 of COMPUTEMTPCISPAIRS, a list of $\langle \mathbf{mtp}, \mathbf{cis} \rangle$ pairs is constructed by combining corresponding elements in $\mathbf{MTPs}$ and $\mathbf{CISs}$.

In lines 5–14 of GETBESTTEC, the **for** loop iterates over this ordered set of $\langle \mathbf{mtp}, \mathbf{cis} \rangle$ pairs computed by

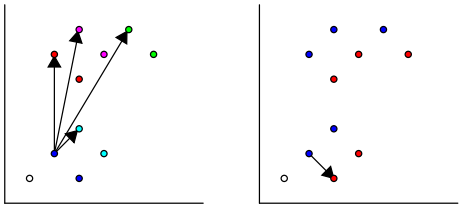

Figure 4. A pair of conjugate TECs. Note that the pattern of blue points in the right-hand figure consists of the upper left point of each pattern in the TEC in the left-hand figure.

COMPUTEMTPCISPAIRS. For each pair, the TEC of the MTP is computed in line 7 using the technique employed in the SIATEC algorithm [17]. Then, in line 8, the *conjugate* TEC [1] is computed for each MTP TEC found in line 7. The concept of a conjugate TEC is illustrated in Figure 4. Given a TEC, $T = \langle D', V \rangle$, the conjugate of T is denoted and defined as follows:

$$\text{GETCONJUGATETEC}(T) = \langle D'', V' \rangle \tag{7}$$

where, if p_0 is the lexicographically first point in D',

$$D'' = \{p_0\} \cup \{p_0 + v \mid v \in V\} , \tag{8}$$

and

$$V' = \{p - p_0 \mid p \in D'\} \setminus \{\langle 0, 0 \rangle\} . \tag{9}$$

Given a pair of conjugate TECs, one may be "better" than the other (e.g., because its pattern might be more compact).

In lines 9 and 10 of GETBESTTEC, redundant translators are removed from both the TEC, T, and its conjugate using the REMOVEREDUNDANTTRANSLATORS algorithm. A translator is defined to be *redundant* if it can be removed from the translator set of a TEC without changing the covered set of the TEC. Ideally, in order to get the most compact description of the covered set of a TEC, one would want to remove as many redundant translators as possible. However, in general, finding the smallest subset of the translator set of a TEC that is sufficient to generate the TEC's covered set is an NP-hard problem. In the implementation of COSIATEC submitted to the MIREX 2013 competition, a greedy approximation algorithm is used to remove as many redundant translators as possible from a TEC within a reasonable running time.

Finally, in lines 11–14 of GETBESTTEC, each MTP TEC and its conjugate are compared with the "best" TEC so far and replace it if they are deemed superior to it by the ISBETTERTEC function, defined in Figure 5. This function takes two TECs as its arguments and returns true if the first is "better than" the second. In lines 1–2 of ISBETTERTEC, the compression ratio of the two TECs are compared. If $\text{P}(T)$ and $\text{V}(T)$ are defined to return the pattern and translator set of a TEC, T, respectively, then the compression ratio of a TEC is defined as follows:

$$\text{CR}(T) = \frac{|\text{COV}(T)|}{|\text{P}(T)| + |\text{V}(T)| - 1} . \tag{10}$$

```
IsBetterTEC(T1, T2)
1   if CR(T1) > CR(T2)
2     return true
3   if Compactness(T1) > Compactness(T2)
4     return true
5   if |COV(T1)| > |COV(T2)|
6     return true
7   if PatternSize(T1) > PatternSize(T2)
8     return true
9   if PatternWidth(T1) < PatternWidth(T2)
10    return true
11  if PatternArea(T1) < PatternArea(T2)
12    return true
13  return false
```

Figure 5. The IsBetterTEC function.

If the two TECs to be compared have the same compression ratio, then they are compared for bounding-box *compactness* (lines 3–4 of IsBetterTEC) [17]. The bounding-box compactness of a TEC is the number of points in the TEC's pattern divided by the number of dataset points in the bounding box of this pattern. If the two TECs have the same compression ratio and compactness, the TEC with largest covered set is considered superior (lines 5–6). If the two covered sets are also the same size, then the TEC with the larger pattern is considered superior (lines 7–8). If the patterns are also the same size, then the TEC with the pattern that has the shorter temporal duration is considered superior (lines 9–10). Finally, if the two TECs also have the same temporal duration, then the TEC with the pattern whose bounding box has the smaller area is considered superior (lines 11–12).

6. THE SIATECCompress ALGORITHM

COSIATEC runs SIATEC on each iteration of its **while** loop. Since SIATEC has worst case running time $O(n^3)$ where n is the number of points in the input dataset, running COSIATEC on large datasets can be time-consuming (see Table **??** for some example running times). On the other hand, because COSIATEC strictly partitions the dataset into non-overlapping MTP TEC covered sets, it tends to achieve high compression ratios for many point-set representations of musical pieces (typically between 2 and 4 for a piece of classical or baroque music).

Like COSIATEC, the SIATECCompress algorithm shown in Figure 6 is a greedy compression algorithm based on SIATEC that computes an encoding of a dataset in the form of a union of TECs. SIATECCompress closely resembles the algorithm described by Forth [6, 7], but is simpler and non-parametric. Like Forth's algorithm, but unlike COSIATEC, SIATECCompress runs SIATEC only *once* to get a list of TECs in decreasing order of quality (as defined by the IsBetterTEC function in Figure 5). It then works its way down this list, selecting TECs to include in the encoding, until the input dataset is covered. 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.

```
SIATECCompress(D)
1   V ← ComputeVectorTable(D)
2   MCPs ← ComputeMtpCisPairs(V)
3   MCPs ← RemoveTranEquivMTPs(MCPs)
4   T ← ComputeTECs(D, V, MCPs)
5   T ← AddConjugateTECs(T)
6   T ← RemoveRedundantTranslators(T)
7   T ← SortTECsByQuality(T)
8   return ComputeEncoding(D, T)
```

Figure 6. The SIATECCompress algorithm.

```
ComputeEncoding(D, T)
1   D' ← ∅
2   E ← ⟨⟩
3   for i ← 0 to |T| − 1
4     T ← T[i]
5     S ← COV(T)
6     if |S \ D'| > |P(T)| + |V(T)| − 1
7       E ← E ⊕ ⟨T⟩
8       D' ← D' ∪ S
9       if |D'| = |D|
10        break
11  R ← D \ D'
12  if |R| > 0
13    E ← E ⊕ ⟨AsTEC(R)⟩
14  return E
```

Figure 7. The ComputeEncoding algorithm.

The first steps in SIATECCompress are to compute a vector table and compute MTPs using the SIA algorithm, implemented in ComputeVectorTable and ComputeMtpCisPairs, as in the first two lines of GetBestTEC (see Figure 2). The next step (line 3 in Figure 6) is to remove MTPs from the list, **MCPs**, that are translationally equivalent to MTPs that occur earlier in this list. This eliminates the possibility of the same TEC being computed more than once in line 4. In line 5, the conjugate of each TEC found in line 4 is also added to the list of candidate TECs, **T**. In line 6, redundant translators are removed from the translator set of each TEC in **T** and, in line 7, the resulting list of candidate TECs is sorted into decreasing order of quality using the IsBetterTEC comparator function. This ordered set of TECs is then given to the ComputeEncoding function (Figure 7), which computes a compact encoding of the input dataset.

6.1 Forth's algorithm

Forth [6, 7] presents an algorithm, that, like COSIATEC, computes a set of TECs that collectively cover the input dataset. However, unlike COSIATEC, Forth's algorithm runs SIATEC only once and the covers it generates are not, in general, strict partitions—in the output of Forth's algorithm, the TECs 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, $C_i = \text{COV}(T_i)$, 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" [6, p. 41] of the patterns in the TEC, T_i, and it is defined as follows:

$$W_i = w'_{\text{cr},i} \cdot w'_{\text{compV},i} \ , \tag{11}$$

where $w'_{\text{cr},i}$ is a normalized value representing the compression ratio of T_i and $w'_{\text{compV},i}$ is a normalized value representing the within-voice segment compactness of the patterns in T_i. Forth [6, p. 38] defines the compression ratio of a TEC in the same way as Meredith et al. [18].

The within-voice segment compactness of a pattern, as used in Forth's algorithm, is typically approximately equal to its bounding-box compactness (as used in COSIATEC). However, an advantage of bounding-box compactness over within-voice segment compactness is that the former does not depend on information about voice structure being present in the input data—that is, bounding-box compactness does not require the musical surface to have already been parsed into unambiguous voices. This avoids the need for selecting one, unambiguous voice structure interpretation for a piece in cases where the voice structure is actually ambiguous (as it often is, for example, in keyboard music). Nevertheless, if information about voice structure is available, one might well hypothesize that within-voice segment compactness would give better results than bounding-box compactness (though the results reported in section **??** do not support this hypothesis).

Having computed the sequence of weights, $\mathbf{W}$, Forth's algorithm then attempts to select a subset of $\mathbf{C}$ that covers the input dataset while maximising the weights, W_i, of the TECs used in the cover. Forth's algorithm takes D, $\mathbf{C}$ and $\mathbf{W}$ as parameters, along with a numerical parameter, $c_{\min}$. The algorithm repeatedly selects the "best" remaining TEC covered set, C^*, in $\mathbf{C}$, adds this to the cover, $\mathbf{S}$, and then removes C^* from $\mathbf{C}$. A point set, P, is used to store the set of points covered by the TEC covered sets selected so far. 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 $c_{\min}$. The TEC covered set, C^*, that is added on a particular iteration is then the one for which $c \cdot W_i$ is a maximum. If no TEC covered set is selected on a particular iteration, then the algorithm terminates, even if the dataset has not been completely covered. The algorithm returns the cover, $\mathbf{S}$, containing the "best" TECs selected.

7. EVALUATION

Two experiments were carried out in order to select the specific variants of the algorithms to be submitted to the 2016 MIREX competition. In the first experiment, 96 variants of each of the three basic algorithms (i.e., COSIATEC, SIATECCOMPRESS and Forth's algorithm) were run on the polyphonic version of the JKU-PDD [2]. For each of these three basic algorithms, the following parameter values were used:

1. compactness trawling [5] was either used or not used;
2. SIA was either replaced with SIAR [1] or it was not;
3. redundant translators were either removed or not;
4. minimum permitted compactness of patterns was set to either 0.5 or 0.9;
5. minimum pattern size was set to either 4 or 8;
6. the algorithms were run in either raw, bb or segment mode.

The morphetic pitch representations of the polyphonic versions of the JKU-PDD were used and the algorithms were run on the "lisp" encodings of the pieces.

The performance of the 288 algorithm variants was measured using using Tom Collins' MATLAB implementation of the metrics defined in [3], bundled with the JKU PDD. Particular attention was paid to the values of three-layer precision, three-layer recall and three-layer F_1 measure [15].

The algorithm variants that performed best in this experiment in terms of three-layer F_1 score were versions of SIATECCOMPRESS with the following parameter values:

1. compactness trawling was not used;
2. SIAR was used in place of SIA;
3. redundant translators were not removed;
4. minimum pattern size was set to 4;
5. the algorithms were run in segment mode.

For these parameter values, the choice of minimum compactness threshold made no difference to the F_1 score. With these parameter values, the three-layer F_1 scores achieved over the five pieces in the database were as follows:

- Bach: 0.27952
- Beethoven: 0.63336
- Chopin: 0.60895
- Gibbons: 0.36589
- Mozart: 0.5609
- Mean F_1 score over all five pieces: 0.489724

Table 1 shows the values of three-layer precision for the algorithms that performed best on average over the JKU-PDD in terms of this measure. Note that the top 8 algorithms were versions of either Forth's algorithm or SIATECCOMPRESS, and that the parameter values in these algorithms were as follows:

1. SIAR was used instead of SIA (indicated by "rsd" in the algorithm name);

Algorithm	Bach	Beet	Chop	Gbns	Mzrt	Mean
Forth-d-m-rsd-rrt-minc0.5-min8-bbcomp-segmode	0.20777	0.64637	0.52875	0.56609	0.60826	**0.511448**
Forth-d-m-rsd-rrt-minc0.9-min8-bbcomp-segmode	0.20777	0.64637	0.52875	0.56609	0.60826	**0.511448**
Forth-d-m-rsd-minc0.5-min8-bbcomp-segmode	0.19754	0.64637	0.52842	0.56609	0.60826	**0.509336**
Forth-d-m-rsd-minc0.9-min8-bbcomp-segmode	0.19754	0.64637	0.52842	0.56609	0.60826	**0.509336**
SIATECCompress-d-m-rsd-minc0.5-min8-bbcomp-segmode	0.23153	0.59303	0.48587	0.67418	0.52817	**0.502556**
SIATECCompress-d-m-rsd-minc0.9-min8-bbcomp-segmode	0.23153	0.59303	0.48587	0.67418	0.52817	**0.502556**
SIATECCompress-d-m-rsd-rrt-minc0.5-min8-bbcomp-segmode	0.22396	0.59303	0.47908	0.67418	0.52227	**0.498504**
SIATECCompress-d-m-rsd-rrt-minc0.9-min8-bbcomp-segmode	0.22396	0.59303	0.47908	0.67418	0.52227	**0.498504**

Table 1. Top-scoring algorithms in first experiment in terms of three-layer precision over the JKU-PDD.

2. minimum pattern size was set to 8 (indicated by "min8" in the algorithm name);

3. the algorithms were run in segment mode (indicated by "segmode" in the algorithm name.

Whether redundant translators were removed or not and the choice of minimum compactness made no difference in terms of precision.

The algorithm variants that performed best in this first experiment in terms of three-layer recall were the versions of SIATECCOMPRESS with the following parameter values:

1. compactness trawling was not used;

2. SIA was used instead of SIAR;

3. redundant translators were not removed;

4. minimum pattern size was set to 4;

5. segment mode was used.

For these algorithms, the minimum compactness setting made no difference to the the precision.

This first experiment confirmed previous results that showed that SIATECCOMPRESS appears to be the most successful of the three basic algorithms on this task. A second, more in-depth experiment was therefore carried out to find the variants of SIATECCOMPRESS that perform best on the JKU-PDD in terms of three-layer precision, recall and F_1 score. In this second experiment, 72 variants of SIATECCOMPRESS were run in segment mode on the JKU-PDD, using the following parameter values:

1. either SIA or SIAR was used; if SIAR was used, the r parameter was set to either 1 or 3;

2. redundant translators were either removed or not;

3. minimum compactness was set to either 0.25 or 0.5 (minimum compactness settings has made no difference to the results in the first experiment);

4. minimum pattern size was set to 4, 6 or 8;

5. the number of patterns generated by the algorithms was either unlimited or limited to 10 patterns.

Tables 2 to 4 show, respectively, the three-layer F_1, precision and recall scores for those variants of SIATECCOMPRESS tested that performed best in terms of these measures.

The results in Table 2 indicate that SIATECCOMPRESS performed best in terms of three-layer F_1 score over the JKU-PDD when

- SIAR was used with r set to 1;
- minimum pattern size was set to 4;
- the number of returned patterns was unlimited ("top0").

Removing redundant translators only very marginally reduced performance. Minimum compactness threshold made no difference to these results. The two best variants achieved a three-layer F_1 score of 0.4897. The variant submitted to the MIREX competition as algorithm MeredithTLF1MIREX2016.jar was therefore a version of SIATECCOMPRESS with the following parameter values:

- SIAR used with $r = 1$;
- minimum compactness set to 0.25;
- minimum pattern size set to 4;
- number of patterns generated unlimited;
- segment mode used;
- redundant translators not removed;
- morphetic pitch representation used.

The results in Table 3 indicate that SIATECCOMPRESS performed best in terms of three-layer precision over the JKU-PDD when

- SIAR was used with $r = 1$;
- redundant translators were not removed;
- minimum pattern size was set to 8;
- the number of patterns returned was limited to 10.

Minimum compactness threshold made no difference to these results. The two best variants achieved a three-layer precision of 0.5614. The variant submitted to the MIREX competition as algorithm MeredithTLPMIREX2016.jar

Algorithm	Bach	Beet	Chop	Gbns	Mzrt	Mean
SIATECCompress-d-m-rsd-r1-minc0.25-min4-bbcomp-segmode-top0	0.27952	0.63336	0.60895	0.36589	0.5609	0.489724
SIATECCompress-d-m-rsd-r1-minc0.5-min4-bbcomp-segmode-top0	0.27952	0.63336	0.60895	0.36589	0.5609	0.489724
SIATECCompress-d-m-rsd-r3-rrt-minc0.25-min4-bbcomp-segmode-top0	0.2697	0.61844	0.61695	0.38521	0.55818	0.489696
SIATECCompress-d-m-rsd-r3-rrt-minc0.5-min4-bbcomp-segmode-top0	0.2697	0.61844	0.61695	0.38521	0.55818	0.489696
SIATECCompress-d-m-minc0.25-min4-bbcomp-segmode-top0	0.28792	0.65169	0.552	0.39973	0.54884	0.488036
SIATECCompress-d-m-minc0.5-min4-bbcomp-segmode-top0	0.28792	0.65169	0.552	0.39973	0.54884	0.488036

Table 2. Top-scoring variants of SIATECCOMPRESS in the second experiment in terms of three-layer F_1 score over the JKU-PDD.

Algorithm	Bach	Beet	Chop	Gbns	Mzrt	Mean
SIATECCompress-d-m-rsd-r1-minc0.25-min8-bbcomp-segmode-top10	0.24153	0.6405	0.6147	0.67418	0.6363	0.561442
SIATECCompress-d-m-rsd-r1-minc0.5-min8-bbcomp-segmode-top10	0.24153	0.6405	0.6147	0.67418	0.6363	0.561442
SIATECCompress-d-m-rsd-r3-minc0.25-min8-bbcomp-segmode-top10	0.25614	0.60302	0.60138	0.67418	0.63356	0.553656
SIATECCompress-d-m-rsd-r3-minc0.5-min8-bbcomp-segmode-top10	0.25614	0.60302	0.60138	0.67418	0.63356	0.553656

Table 3. Top-scoring variants of SIATECCOMPRESS in the second experiment in terms of three-layer precision over the JKU-PDD.

Algorithm	Bach	Beet	Chop	Gbns	Mzrt	Average
SIATECCompress-d-m-minc0.25-min4-bbcomp-segmode-top0	0.42892	0.74609	0.82127	0.34474	0.57459	0.583122
SIATECCompress-d-m-minc0.5-min4-bbcomp-segmode-top0	0.42892	0.74609	0.82127	0.34474	0.57459	0.583122
SIATECCompress-d-m-rrt-minc0.25-min4-bbcomp-segmode-top0	0.42892	0.73008	0.83183	0.34474	0.56669	0.580452
SIATECCompress-d-m-rrt-minc0.5-min4-bbcomp-segmode-top0	0.42892	0.73008	0.83183	0.34474	0.56669	0.580452
SIATECCompress-d-m-rsd-r3-rrt-minc0.25-min4-bbcomp-segmode-top0	0.36738	0.68142	0.86171	0.35277	0.6106	0.574776
SIATECCompress-d-m-rsd-r3-rrt-minc0.5-min4-bbcomp-segmode-top0	0.36738	0.68142	0.86171	0.35277	0.6106	0.574776

Table 4. Top-scoring variants of SIATECCOMPRESS in the second experiment in terms of three-layer recall over the JKU-PDD.

was therefore a version of SIATECCOMPRESS with the same parameter values as MeredithTLF1MIREX2016.jar, except the minimum pattern size was increased to 8 and the number of patterns generated was limited to 10.

The results in Table 4 indicate that SIATECCOMPRESS performed best in terms of three-layer recall over the JKU-PDD when

- SIA was used instead of SIAR;
- redundant translators were not removed;
- minimum pattern size was set to 4;
- the number of patterns generated was unlimited.

Minimum compactness threshold made no difference to recall. The two best variants achieved a three-layer recall value of 0.5831. The variant submitted to the MIREX competition as algorithm MeredithTLRMIREX2016.jar was therefore a version of SIATECCOMPRESS in segment mode in which

- SIA was used instead of SIAR;
- redundant translators were not removed;
- minimum pattern size was set to 4;
- the number of patterns generated was unlimited;
- minimum compactness was set to 0.25.

8. REFERENCES

[1] Tom Collins. *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, 2011.

[2] Tom Collins. JKU Patterns Development Database, 2013. Available at `https://dl.dropbox.com/u/11997856/JKU/JKUPDD-Aug2013.zip`.

[3] Tom Collins. MIREX 2013 Competition: Discovery of Repeated Themes and Sections, 2013. `http://tinyurl.com/o9227qg`. Accessed on 5 January 2015.

[4] Tom Collins. MIREX 2014 Competition: Discovery of Repeated Themes and Sections, 2014. `http://tinyurl.com/krnqzn5`. Accessed on 9 April 2015.

[5] Tom Collins, Jeremy Thurlow, Robin Laney, Alistair Willis, and Paul H. Garthwaite. A comparative evaluation of algorithms for discovering translational patterns in baroque keyboard works. In *Proceedings of the 11th International Society for Music Information Retrieval Conference (ISMIR 2010), Utrecht, The Netherlands, 9–13 August 2010*, pages 3–8, 2010.

[6] James C. Forth. *Cognitively-Motivated Geometric Methods of Pattern Discovery and Models of Similarity in Music*. PhD thesis, Department of Computing, Goldsmiths, University of London, 2012.

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

[8] David Meredith. 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, 2006. Available online at `http://drops.dagstuhl.de/opus/volltexte/2006/652`.

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

[10] David Meredith. *Computing Pitch Names in Tonal Music: A Comparative Analysis of Pitch Spelling Algorithms*. PhD thesis, Faculty of Music, University of Oxford, 2007.

[11] David Meredith. Analysis by compression: Automatic generation of compact geometric encodings of musical objects. In *The Music Encoding Conference (MEC 2013)*, 2013. `http://www.titanmusic.com/papers/public/MeredithMEC2013ProceedingsPaper.pdf`.

[12] David Meredith. COSIATEC and SIATECCompress: Pattern discovery by geometric compression. In *MIREX 2013 (Competition on Discovery of Repeated Themes & Sections)*, 2013. Available online at `http://www.titanmusic.com/papers/public/MeredithMIREX2013.pdf`.

[13] David Meredith. Compression-based geometric pattern discovery in music. In *Fourth International Workshop on Cognitive Information Processing (CIP 2014), 26–28 May 2014*, Copenhagen, Denmark, 2014.

[14] David Meredith. Using point-set compression to classify folk songs. In *Fourth International Workshop on Folk Music Analysis (FMA 2014), 12–13 June 2014*, Bogazici University, Istanbul, Turkey, 2014.

[15] David Meredith. Music analysis and point-set compression. *Journal of New Music Research*, 44(3):245–270, 2015. `http://dx.doi.org/10.1080/09298215.2015.1045003`.

[16] David Meredith. Analysing music with point-set compression algorithms. In David Meredith, editor, *Computational Music Analysis*, pages 335–366. Springer, 2016. `http://link.springer.com/chapter/10.1007/978-3-319-25931-4_13`.

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

[18] David Meredith, Kjell Lemström, and Geraint A. Wiggins. Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music. In *Cambridge Music Processing Colloquium*, 2003. http://www.titanmusic.com/papers/public/cmpc2003.pdf.

[19] David Meredith, Geraint A. Wiggins, and Kjell Lemström. Method for pattern discovery. UK Patent GB2379056, granted 29 September 2004. (Priorities: GB0112551 23 May 2001; GB0200203 07 Jan 2002). http://v3.espacenet.com/textdoc?DB=EPODOC&IDX=EP1402400&QPN=EP1402400. Draft available at http://www.titanmusic.com/papers/public/patent2002c_for_FP.pdf.