Symbolic Music Analysis with General-Purpose Compression Algorithms

Corentin Louboutin[1] and David Meredith[2]

[1] École normale Supérieure, Rennes, France,
corentin.louboutin@ens-rennes.fr
[2] Aalborg University, Aalborg, Denmark
dave@create.aau.dk

Internship carried out in the *Music Informatics and Cognition Research Group* in the Department of Architecture, Design and Media Technology, Aalborg University from 19 May to 17 August.

Abstract. This paper reviews some compression algorithms designed or modified for analyzing symbolic music data. The algorithms were used in combination with a new method of classification based on multiple viewpoints and the k-nearest-neighbours algorithm to classify the folk tunes in the *Annotated Corpus* of the Dutch Song Database (Nederlandse Liederenbank). The best-performing algorithm achieved a classification success rate 94%.

1 Introduction

Folk music can be used to analyze the cultural development of a geographical region and for studying interactions between cultures [1]. In order to do this, folk songs are typically categorized by tune family, geographic location, genre and so on. Two songs are said to be in the same category if they have a common ancestor in the tree of oral transmission [2]. The possibility of automatic classification of folk songs was first noted as early as the year 1900 by the musicologist Daniel François Scheurleer [25].

Van Kranenbourg et al. [27] show that a classification method based on local features [5, 7, 10, 27], such as pattern similarity, outperforms all methods that focus on global features [8, 10, 27], such as tonality, first and last note of the song, average pitch and so on. Moreover, Conklin et al. [5, 7] extend this analysis with their *multiple viewpoint model* which combines different features, local and global. This method provided good results on prediction and generation of music [6, 22]. Conklin recently showed that applying this model to a classification task outperforms all classification algorithms that use just one feature.

General-purpose compression algorithms such as *Burrows-Wheeler* [3] (BW), *Lempel-Ziv-77* [31] (LZ77) and *Lempel-Ziv-78* [32] (LZ78) are based on the redundancy of the input sequence. This suggested that they might be useful for finding musically relevant patterns. Therefore, our idea was to explore whether these algorithms, perhaps modified for song compression, could be used successfully for music classification. We also investigated the effect that using different representation schemes has on the efficiency and effectiveness of these algorithms.

Meredith [17–20], inspired by the theory of Kolmogorov complexity [13, 28], hypothesized that the simplest and shortest descriptions of any musical object are those that describe the best possible explanations for the structure of that object. He developed an algorithm, COSIATEC ("COmpression with Structure Induction Algorithm, and Translational Equivalence Classes"), which finds maximal repeated patterns in a point-set representation of a piece of music and uses these to produce a losslessly compressed encoding of the input piece. Meredith evaluated COSIATEC [20] by using it to classify the songs in the *Annotated Corpus* of Dutch folk songs [27] by computing *Normalized Compression Distances* (NCDs), the *1-nearest-neighbour* classification algorithm and *leave-one-out* cross-validation (i.e., each song of the corpus is associated with the class of the most similar song in the rest of the corpus).

This paper presents and analyses derivative versions of four compression algorithms: *Burrows-Wheeler* [3], *Lempel-Ziv-77* [31], *Lempel-Ziv-78* [32] and COSIATEC. The first three are general-purpose compression algorithms that were developed for text compression. In this paper they are used on sequences of two-dimensional points, but those sequences will be considered as one-dimensional strings of letters from the alphabet $\mathbb{Z}^2$. For this reason, examples will use letters instead of two-dimensional points. The goal is to preserve the design of the text compression algorithms, but present the musical data in a way that allows them to find repeated patterns.

The main purpose of this paper is to determine how well these algorithms can find musically relevant patterns and detect similarities. Moreover, the paper aims to evaluate the effect of the representations on general-purpose compression algorithms.

Sections 2 to 5 present the four algorithms we used. Then we present the new classification method that combines the *multiple viewpoints* approach [5] and the *k-nearest-neighbours* algorithm. The last section compares all algorithms used with different representations. This evaluation is done on a classification task run on the Dutch folk song dataset, *Onder der Groene linde* [9], with the new classification method.

2 Burrows-Wheeler

There are many lossless compressors used today. One of them is *bzip2* [26] based on the work of Burrows and Wheeler [3, 24]. The algorithm they present uses a transformation on the sequence and entropy coding. The method provides a result at least as good as the one given by *gzip*, and is generally faster. In fact, it outperforms *gzip* both in speed and in compression. It was therefore decided to implement it for note sequences.

The algorithm consists of three parts:

1. The *Burrows-Wheeler transform.* This step executes a permutation of the input sequence that improves the compression effect of the following step.
2. *Move-to-front coding.* This is a transformation that can improve the performance of entropy coding such as Huffman coding. It also has a high compression effect.
3. *Huffman* or *arithmetic coding.*

We implemented each part except the Huffman coder as it improves neither the success rate results nor the compression ratios for the *Annotated Corpus*. Huffman coding fails to improve performance because we use a string representation and there are only a few notes in a melody, then the radix-10 representation is better than radix-2 because it uses fewer characters. The results given by move-to-front coding are presented in section 7.

2.1 Burrows-Wheeler Transform

This transform performs a permutation on the input string. The aim of this permutation is to bring equal elements closer together. This permutation increases the probability of finding a character c at a point in a sequence if c already occurs near this point. Move-to-front coding can then have a better compression effect.

The Burrows-Wheeler transform uses an $n \times n$ matrix where n is the length of the input string S. The elements of this matrix are points in S. Each row is a cyclic shift of S. There is therefore at least one row that is equal to the input. The rows are then sorted into lexicographic order. The output of the algorithm is a pair (T, i), where T is the last column of the matrix and i is the index of a (usually *the*) row corresponding to S.

An example of such a sorted matrix using the input string $S = banana$ is shown in Figure 1. As S appears in the row 3, the output is then the pair formed by the string of the last column and this index: $(nnbaaa, 3)$. In this example, characters that are equal are regrouped together. However, this is not always the case, as can be seen in Burrows' own example, *abraca*, which is transformed into *caraab* [3].

2.2 Move-to-front Coding

This algorithm is used to encode the string returned by the Burrows-Wheeler transform. It takes a string T as input and returns a vector R of integers. This algorithm needs to know the alphabet Y of the input, so the first step consists

row	T
0	$a\ b\ a\ n\ a\ n$
1	$a\ n\ a\ b\ a\ n$
2	$a\ n\ a\ n\ a\ b$
3	$b\ a\ n\ a\ n\ a$
4	$n\ a\ b\ a\ n\ a$
5	$n\ a\ n\ a\ b\ a$

Fig. 1. Example of matrix used by Burrows-Wheeler transform.

```
Data: A sequence of letters $T$
Result: A vector of integers $R$
$Y$ = alphabet of $T$;
construct an empty array $R$ of length $|T|$;
for $i = 0$ to $|T|$ do
    $R(i)$ = index of $T(i)$ in $Y$;
    Move $T(i)$ to the front of $Y$;
end
```

Algorithm 1: Move-to-front coding.

of an iterative algorithm that builds the alphabet by reading the input string from left to right, adding new characters to an initially empty alphabet.

The algorithm then builds R by executing Algorithm 1 (see above). It replaces each character, $T[i]$, by its index in the alphabet, Y, and then places that character at the beginning of Y. Applied on the example *nnbaaa*, it first computes the alphabet $Y = [n, b, a]$ and then returns the integer vector $R = [0, 0, 1, 2, 0, 0]$.

The input of this algorithm is such that when a character appears, the probability that it has already appeared or will appear again is high. Therefore, the integer found by the first instructions of the loop will be lower than without the transform.

To ensure reversibility, the algorithm needs to return the alphabet, Y, as well as the integer vector R returned by the move-to-front coding algorithm and the index i returned by the Burrows-Wheeler transform.

3 Lempel-Ziv-77

In 1977, A. Lempel and J. Ziv introduced a lossless dictionary-based data compression algorithm: LZ77 [31]. There are some improvements for this algorithm such as LZMA, used by the 7zip compressor [11], but some compressors such as ZPAQ, which is one of the best today [15], still continue to use the basic version of LZ77. LZ77 uses some pattern discovery—indeed, it codes repeated substrings by references to their last occurrences [24]. This is why we decided to see what it can do with a string of notes. Are the patterns found by this algorithm musically relevant?

The LZ77 algorithm uses a *sliding window* which consists of two parts: the *dictionary* part and the *look-ahead buffer*. The dictionary contains an already encoded part of the sequence, and the look-ahead buffer contains the next portion of the input to encode. The size of each part is determined by the two parameters n, the size of the window, and L_s, the maximal matching length, that is the size of the look-ahead buffer.

The principle of this algorithm is to find the longest prefix of the *look-ahead buffer* that also begins in the *dictionary*. The output is then a sequence of triples $(p_i, l_i - 1, c)$ where p is a pointer to the first letter of the dictionary occurrence, $l_i - 1$ is the length of the prefix and c is the first character that follows the prefix in the look-ahead buffer.

Let S_1 and S_2 be two strings. $S_1(i)$ denotes the $(i+1)$th element in S_1 (zero-based indexing is used). $S_1(i, j)$ is the substring from $S_1(i)$ to $S_1(j)$. S_1S_2 is the string obtained by concatenating S_1 and S_2. Finally, S_1^n denotes S_1 concatenated n times.

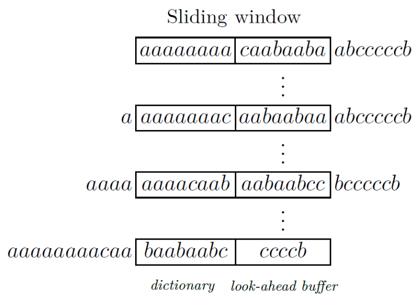
LZ77 is an iterative algorithm. First it initializes the window W by filling the dictionary with a null letter (a in our example, however, in practice, we use the point $(0, 0)$). The look-ahead buffer is then filled with the first L_s notes of the input sequence S to be encoded:

$$W = a^{n-L_s} S(0, L_s - 1)$$

The followings steps are then repeated until the whole sequence S is encoded:

1. Find $S_i = W(n - L_s, n - L_s + l_i - 2)$, the longest prefix of length $l_i - 1$ of the look-ahead buffer that also begin at index p_i in the dictionary. When there is no prefix (i.e. $l_i = 1$), $p_i = 0$, and when there are several possible p_i, the smaller is taken. There can be overlapping if $l_i + p_i > n - L_s$.
2. Add the triple $(p_i, l_i - 1, c)$ to the output string (radix-10 representation is used for p_i and l_i). c is the first character that follows the prefix in the look-ahead buffer: $c = W(n - L_s + l_i - 1)$.
3. Shift the window and fill the end of the look-ahead buffer with the next l_i letters of the input sequence: $W = W(l_i, n)S(h_i + 1, h_i + l_i)$ where h_i is the index in S of the last element of W before the shift operation.

Figure 2 shows LZ77 being used to encode the sequence *caabaabaabcccccb*. It first fills the dictionary with 'a' and the look-ahead buffer with the 8 first characters of the input sequence. Then there is no substring in the dictionary that begins with a c so $l_i = 1$ and $p_i = 0$, the character following the prefix is c. Then, we shift the window by one (value of l_i) and obtain the state given in the second line. Here we found the prefix aa followed by a b so $l_i = 3$ and as p_i can be any integer between 0 and 5, the algorithm returns the lowest one: $p_i = 0$, and then we have to shift the window by 3. The state obtained is shown on


Sliding window
aaaaaaaa | caabaaba | abcccccb
a | aaaaaaac | aabaabaa | abcccccb
aaaa | aaaacaab | aabaabcc | bcccccb
aaaaaaaaacaa | baabaabc | ccccb
dictionary
look-ahead buffer

p_i	l_i	c	S_i
0	0	c	
0	2	b	aa
5	6	c	$aabaab$
7	4	b	$ccccc$

Fig. 2. Sliding window used by the LZ77 algorithm.

line 3. Here there is an overlapping, the algorithm returns $(5, 6, c)$. In fact, the prefix found is *aabaab* so the reproduction of this prefix begins in the dictionary and ends in the look-ahead buffer. The algorithm ends by doing one more step. Finally, the output is:

$$(0, 0, c)(0, 2, b)(5, 6, c)(7, 4, b)$$

4 Lempel-Ziv-78

The Lempel-Ziv-78 (LZ78) algorithm is also a dictionary-based compression algorithm [24, 32]. However, in LZ78, the dictionary is not limited in size. Many algorithms have been developed based on LZ78. The most famous is the Lempel-Ziv-Welch [29] (LZW) algorithm which is used by the basic Linux command *compress*. However, as LZW needs to know the input alphabet which contains $\mathbb{Z}^2$ symbols in our case, we preferred to use the basic LZ78 version.

The principle of this algorithm is to fill an explicit dictionary with substrings of the input. A feature of this algorithm is that the dictionary is the same at encoding and decoding.

LZ78 works in four steps:

1. Create an empty substring B and enlarge it by adding characters of the input S until B does not appear in the dictionary.
2. Add the pair (i, c) to the output, where i is the last index met (i.e., the index corresponding to the longest match of B in the dictionary) and c is the last character added. In practice, when $i = -1$, the algorithm returns $(\texttt{'x'}, c)$. This improves the compression ratio a little because it uses one character instead of two.
3. Add B to the dictionary.
4. Set B to the empty string and repeat the steps until the whole input is encoded.

	Dictionary	
Output	Index	Entry
(x, c)	0	c
(x, a)	1	a
$(1, b)$	2	ab
$(1, a)$	3	aa
(x, b)	4	b
$(3, b)$	5	aab
$(0, c)$	6	cc
$(6, c)$	7	ccc
$(4, \epsilon)$	8	

Fig. 3. Example of sequence encoding with the LZ78 algorithm.

Figure 3 illustrates the encoding of the sequence *caabaabaabcccccb* with LZ78. When the algorithm begins, the dictionary is empty, therefore the two first letters encountered (c and a) are directly added into it and the returned index is -1 (written 'x'). Then a is added to an empty B, but as a is already in the dictionary, the algorithm adds also b, producing $B = ab$ which is not in the dictionary. The output is then $(1, b)$ the index of the longest match (a) in the dictionary and the last character of B. It also adds B in the dictionary as a new substring encountered. The details of the remainder of the encoding process are tabulated in Figure 3.

5 COSIATEC

COSIATEC (COmpression with Structure Induction Algorithm, and Translational Equivalence Classes) is not a general-purpose algorithm. It is an SIA-based algorithm developed by Meredith et al. [16, 19, 21, 30] to find repeated patterns in a musical score. But it can be seen as a compression algorithm for symbolic music [18, 20]. In fact it returns a set of *Translational Equivalence Classes* (TEC), where each TEC is a pair (*pattern*, *translator set*) and this set is typically shorter than (and never longer than) the input set of notes.

The input of COSIATEC is a dataset (no sequence order needed) of two-dimensional points (notes). A *pattern* can be any subset of this dataset. For a dataset D and a vector $\mathbf{v}$, a *Maximal Translatable Pattern* (MTP) is defined by:

$$\mathrm{MTP}(\mathbf{v}, D) = \{p | p \in D \wedge p + \mathbf{v} \in D\}$$

where $p + \mathbf{v}$ is the point obtained by the translation of the point p by the vector $\mathbf{v}$. $\mathrm{MTP}(\mathbf{v}, D)$ is the subset of all points of D that have an image in D when translated by $\mathbf{v}$.

The algorithm used to find MTPs, called SIA, is well described in [21], and will not be described here.

The equivalence relation used to build TECs, denoted $\equiv_T$ is defined between two patterns P_1 and P_2 of a dataset D:

$$P_1 \equiv_T P_2 \iff (\exists \mathbf{v} | P_2 = P_1 + \mathbf{v})$$

where $P_1 + v$ defines the set obtain by translating all points of P_1 by the vector $\mathbf{v}$. The TEC of the pattern $P \subseteq D$ is the equivalence class of P:

$$\mathrm{TEC}(P, D) = \{Q | Q \equiv_T P \wedge Q \subseteq D\}$$

COSIATEC first runs SIATEC to find MTP TECs—that is translational equivalence classes of the maximal translatable patterns in the input dataset. Each TEC is represented by a pair $(\mathit{pattern}, \mathit{translator\ set})$. Moreover the returned TECs have the property that they cover the whole input dataset and that all TEC pairs are disjoint. It means that COSIATEC gives an encoded partition of the input dataset.

The algorithm is quite complex, but fully described by Meredith [19].

6 Combined Representations Classification Method

This section presents the method used in evaluating the compression algorithms described above. This method is based on Conklin et al.'s [6] idea that "no single music representation can be sufficient for music" and that combining several representations can produce a better model. With this method, they achieved good results both in prediction, generation and classification [4, 5, 7, 22, 23]. Our new method combines this multiple viewpoints approach [5] with the well-known k-nearest-neighbours algorithm.

6.1 Definitions

We define a *representation* of a melody to be a reversible function that takes a string of two-dimensional points and returns a string of two-dimensional points. It preserves the size of the string and the sequence of points—that is, a point is replaced in the sequence by its new representation. Each representation used is described in appendix A. We also used composition of transformations, $\circ$, as the composition on functions.

The *onset* of a note is the time of the start of the note and the *pitch* is the MIDI value of the note. In this paper, all representation are applied on a string of $(\mathit{onset}, \mathit{pitch})$ points sorted in the lexicographic order because all melodies are monophonic—that is, all points have different *onset*. However, for polyphonic music, another base-representation could be more appropriate.

Here, a *viewpoint* is a pair (Z, R) where Z is a compression algorithm and R is a representation. It can be seen as a function that takes a song in the *pitch-time* representation and returns a string of characters: $Z \circ R$.

To be able to use 1-nearest-neighbour, Meredith used a distance called *Normalized Compression Distance* [14] defined on a viewpoint Z and two songs s and s' as:

$$\mathrm{NCD}(Z, s, s') = \frac{Z(s+s') - \min(Z(s), Z(s'))}{\max(Z(s), Z(s'))}$$

This distance has two problems. First, the values are not restricted to being in the interval $[0;1]$. Second, for two different compression algorithms on the same corpus, the distances will not be comparable. For example, in our evaluation, one of the algorithms gave values in $[0.5;0.8]$, and another produced values in $[0.8;1.2]$. We therefore define another distance, called *Corpus Compression Distance* (CCD) which depends on the corpus C used for classification. It has the feature that it computes values in $[0;1]$ for all algorithms. The CCD is defined by the following formula:

$$\mathrm{CCD}(s, s', Z, C) = \frac{\mathrm{NCD}(Z, s, s') - \min_{s_1, s_2 \in C \cup \{s\}} \mathrm{NCD}(Z, s_1, s_2)}{\max_{s_1, s_2 \in C \cup \{s\}} \mathrm{NCD}(Z, s_1, s_2) - \min_{s_1, s_2 \in C \cup \{s\}} \mathrm{NCD}(Z, s_1, s_2)}$$

To evaluate the algorithms, we also examined the compression ratios achieved, since these appeared to be related to the classification success rate. The compression ratio *CR* of a viewpoint v on a song s is defined by:

$$CR(v, s) = \frac{|s|}{|v(s)|}$$

where $|x|$ gives the size of the file containing the string x. The success rate is simply defined by the fraction:

$$SR = \frac{\text{number of well classified songs}}{\text{number of songs in the corpus}} \; .$$

6.2 Classification Method

The classification method takes a song and a corpus as input and returns a class which aims to be the real tune family of the song. For this, it computes a matrix M like the one developed by Conklin et al. [5,7]. The matrix is shown in Table 1. To fill this matrix, we use a function f that depends on:

- C, the known corpus (i.e. the labeled songs);
- s, the song to classify (not yet labeled);
- j, the class to evaluate;
- v_i, the viewpoint applied; and
- N, the number of nearest neighbours to consider.

This function f gives a measure of how similar the song, s, is to its nearest neighbours that are in tune family j. The higher the value is, the higher the

$$
\begin{array}{c|ccccc}
M & 1 & \cdots & \text{j} & \cdots & \text{m} \\
\hline
v_1 & & & & & \\
\vdots & & & \vdots & & \\
v_i & & \cdots & f(C,s,j,v_i,N) & \cdots & \\
\vdots & & & \vdots & & \\
v_n & & & & & \\
\hline
 & & \cdots & g(j) & \cdots &
\end{array}
$$

Table 1. Table computed for the song to be classified.

probability that s will be in j. The value of f is given by the following formula:

$$f(C,s,j,v_i) = \sum_{s_{nn} \in C_j^N(s)} \frac{1}{(\mathrm{CCD}(s,s',v_i,C)+\epsilon)^{8*nn}}$$

where ϵ is a constant as low as we want, and:

$$C_j^N(s) \;=\; C_j \cap C^N(s)$$

C_j is the subset of C which contains the song of class j and $C^N(s)$ is the N nearest neighbours of s in C. The primary purpose of the ϵ factor was to avoid divide-by-zero error, but the value and the placement of it under the power has little effect on the results. In practice, we use $\epsilon = 0.1$.

The bottom row in Table 1 is computed by combining the values of each column with the function g which is the geometric mean of the values of f for the class j, weighted by the proportion of corpus songs in class j. g is defined by:

$$g(j) \;=\; \frac{|C_j|}{|C|} * \sqrt[n]{\prod_{i=1}^{n} M_{i,j}}$$

where $|.|$ is used for the cardinality of sets. As this method is used with the leave-one-out strategy, s is not in C so not in C_j. Finally, we choose the class with the maximum value to classify s:

$$c^* \;=\; \operatorname*{argmax}_{c \in [1;m]} g(c)$$

7 Results

This section presents results for the classification on the *Annotated Corpus* of the Dutch folk song melodies, *Onder der Groene linde* [9]. The corpus is available on the website of the Dutch Song Database (`http://www.liederbank.nl`) provided by the Meertens Institute. It consists of 360 melodies such that each of the 26 tune families is represented by at least 8 melodies and not more than 27 melodies. Each melody is labeled with the name of the family it belongs to. The melodies each contain around 50 notes and are monophonic (i.e., at most one note sounds at any one time).

To classify each melody, we use the method described in Section 6 and leave-one-out cross-validation. We first test the method with single viewpoints separately and then we show that combining viewpoints produces better results than when individual viewpoints are used alone. Appendix B describes how LZ77 parameters were chosen

7.1 Single Viewpoint Classification

To evaluate our method, we first used it with single viewpoints separately. The method was used with $N = 8$. That is, the method only considered the first 8 nearest neighbours of the song to classify. The reason for this value is that the smallest tune family has only 8 melodies and so a larger N would increase the error in the method. The success rate is obtained by leave-one-out cross-validation—that is, each song is classified using all the other songs in the corpus. The various representations used are described in appendix A.

As all songs are monophonic (i.e., all onsets are different), general-purpose compression algorithms can only work on representations that transform the onsets. As these algorithms need equality on points in order to compress the strings, if all onsets are distinct, this implies a compression ratio less than 1. Compression ratios less than 1 were associated with poor classification success rates, so all viewpoints with an average compression ratio less than 1 were discarded.

Conversely, COSIATEC cannot use representations that transform the onsets (*ioi*, *ioib* and combined—see Appendix A). Those representations are good for LZ77, LZ78 and BW because they create redundancy, but COSIATEC needs a set of distinct points in order to work. In fact it is the condition of the reversibility of COSIATEC. Therefore, those viewpoints were also discarded.

Table 2 shows the results obtained by using the classification method on each viewpoint separately (i.e., in each case, the table corresponding to Table 1 contained only one row). Only those algorithm-viewpoint combinations are listed that resulted in a success rate higher than 70%. We can see that in terms of success rate, COSIATEC outperforms all of the other compression algorithms with a value of 0.8528 with the basic $(\mathit{onset}, \mathit{pitch})$ representation. The viewpoint $(\text{COSIATEC}, \mathit{int})$ achieves poorer results than the viewpoint $(\text{COSIATEC}, \mathit{basic})$. This implies that the patterns found are not the same with each representation. Therefore, it is very important to find the representation that provides the best success rate for a given algorithm.

LZ77 also produced very good results and we can see that it is good for several representations. In fact, eight of the ten best viewpoints use LZ77. However, this

Viewpoint	*1-NN Leave-one-out SR*	CR_{AC}	CR_{pairs}
(COSIATEC, *basic*)	0.8528	1.5794	1.6670
(LZ77, $int \circ ioi$)	0.8222	1.4597	1.6735
(LZ77, $ioi \circ ioi$)	0.8222	1.2108	1.3547
(LZ77, *ioi*)	0.8194	1.3075	1.4915
(LZ77, $int0 \circ ioi$)	0.8139	1.3769	1.5690
(LZ77, *ioib*)	0.7944	1.1188	1.2629
(LZ77, $int0 \circ ioib$)	0.7861	1.1806	1.3306
(COSIATEC, *int*)	0.7556	1.5266	1.6226
(LZ77, $ioi \circ ioib$)	0.7472	1.0088	1.1127
(LZ77, $int \circ ioib$)	0.7444	1.2389	1.4062
(BW, *ioi*)	0.7333	1.9627	2.2768
(BW, $int0 \circ ioi$)	0.7194	2.0732	2.3853
(BW, $int0 \circ ioib$)	0.7111	1.4192	1.5436
(LZ78, *ioi*)	0.6361	1.7542	1.9292

Table 2. Results of the classification method with each single viewpoint apart. SR is used for Success Rate, CR_{AC} is for mean compression ratio on *Annotated Corpus*, CR_{pairs} is for the average compression ratio on pair files used to compute the NCDs.

algorithm does not compress well for most of the representations. Conversely, the Burrows-Wheeler algorithm achieved good compression but did not perform so well in terms of classification.

The bottom row of Table 2 gives the best result achieved using LZ78. The average compression ratio is similar to that achieved with Burrows-Wheeler, but the success rate is very low. The reason is that the melodies are very short (approximately 50 notes), whereas LZ78 needs a lot of notes to match long patterns. We would expect LZ78 to perform better on longer pieces such as fugues or sonata-form movements, since the patterns it finds in such longer data would be likely to be longer and more relevant (i.e., there would be more long patterns).

7.2 Combined Viewpoints Classification

The last evaluation was used to find the best viewpoints. We chose to use the combined representations method only on viewpoints that gave good results. Then different combinations were tested to see which viewpoint improved the result. We can see in Table 3 all success rates obtained by the combined representations method using the n viewpoints that performed best individually. All results are better than those obtained using single viewpoints (cf. Table 2). But, it seems that some viewpoints have a detrimental effect on success rate, e.g. (LZ77, $int0 \circ ioi$). The last result $10'$ of Table 3 is obtained by removing the two viewpoints (LZ77, $int0 \circ ioi$) and (COSIATEC, *int*) from the ten first viewpoints combination.

First viewpoints	*Leave-one-out SR*
2	0.8833
3	0.9139
4	0.925
5	0.9083
6	0.9083
8	0.9194
10	0.9333
12	0.9139
14	0.9139
10’	0.9444

Table 3. Results for the classification method with the n best viewpoints.

All the above results show that the representation used is an important factor in the classification success rate achieved. It also shows that general-purpose compression algorithms can be used to find musically relevant patterns or at least repetitions in a song. Indeed, the representation has a large effect on both the accuracy of the classification method and the compression ratio. The best success rate obtained with our new method is 0.9444.

Conklin et al. [5, 7] ran his own method on the same corpus and achieved a success rate of 0.967 with the arithmetic fusion function and 0.958 with the geometric one. We speculate that the difference may be due to the fact that he was additionally using duration information to build viewpoints while we only use pitch and note onset information.

8 Conclusions and further work

This paper presents a new classification method applied to the *Annotated Corpus*. This method, based on *normalized compression distance*, the *k-nearest-neighbours* algorithm and the *multiple viewpoints* approach, was run with several compression algorithms and representations. The results show that the representation of the local features has a large effect on the performance of algorithms. COSIATEC outperforms all general-purpose algorithms on the classification task when single viewpoints were used. COSIATEC also performed best when Meredith evaluated it against other SIA-based algorithms [19]. However, it would be interesting to evaluate those other SIA-based algorithms also, using the new representation and classification methods presented in this paper.

The LZ77 algorithm also performed very well with single viewpoints. The results indicate that general-purpose compression algorithms show great promise for use in musical analysis.

The next step of our work will be to try the classification method on other corpora in order to provide a better analysis of the effect of representation method on the algorithms.

LZ78 performed poorly in this study. However, we suspect that it would perform better on longer pieces that can provide the algorithm with enough data to build a dictionary. This possibility will be explored in future work.

In this project, the evaluations were carried out on a corpus of monophonic melodies. A future challenge will be to adapt the general-purpose compression algorithms described above for use on polyphonic music (COSIATEC already works on polyphonic music). We suspect that simply using the lexicographic ordering of points will not be satisfactory, as adjacent points in the encoding will not correspond to adjacent notes in a part or voice. Perhaps a voice-separation algorithm (e.g., [12]) could be used to derive voices before applying our classification method.

References

1. R. Bauman. *Folklore, cultural performances, and popular entertainments: a communications-centered handbook.* Oxford University Press, 1992.
2. S. P. Bayard. Prolegomena to a study of the principal melodic families of british-american folk song. *Journal of American Folklore*, pages 1–44, 1950.
3. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. *SRC Research Report*, 124, 1994.
4. P. Chordia, A. Sastry, T. Mallikarjuna, and A. Albin. Multiple viewpoints modeling of tabla sequences. In *ISMIR*, volume 2010, page 11th, 2010.
5. D. Conklin. Multiple viewpoint systems for music classification. *Journal of New Music Research*, 42(1):19–26, 2013.
6. D. Conklin and I. H. Witten. Multiple viewpoint systems for music prediction. *Journal of New Music Research*, 24(1):51–73, 1995.
7. D. Conklin and I. H. Witten. Fusion functions for multiple viewpoints. In *International Workshop on Machine Learning and Music (MML)*, 2013.
8. L. C. Freeman and A. P. Merriam. Statistical classification in anthropology: An application to ethnomusicology. *American Anthropologist*, 58(3):464–472, 1956.
9. L. Grijp. Introduction. In L.P. Grijp and I. van Beersum, editors. *Under the Green Linden — 163 Dutch Ballads from the Oral Tradition*, pages 18–27, 2008.
10. R. Hillewaere, B. Manderick, and D. Conklin. Global feature versus event models for folk song classification. In *ISMIR*, volume 2009, pages 729–733. ISMIR, 2009.
11. I. Pavlov. LZMA description. `http://www.7-zip.org`. 2013.
12. I. Karydis, A. Nanopoulos, A. Papadopoulos, E. Cambouropoulos, and Y. Manolopoulos. Horizontal and vertical integration/segregation in auditory streaming: a voice separation algorithm for symbolic musical data. In *Proceedings 4th Sound and Music Computing Conference (SMC'2007)*, 2007.
13. A. N. Kolmogorov. Three approaches to the quantitative definition ofinformation'. *Problems of information transmission*, 1(1):1–7, 1965.
14. M. Li, X. Chen, X. Li, B. Ma, and P. M. Vitányi. The similarity metric. *Information Theory, IEEE Transactions on*, 50(12):3250–3264, 2004.

15. M. Mahoney. Large text compression benchmark. *URL: http://www. mattmahoney. net/text/text. html*, 2009.
16. D. Meredith. Point-set algorithms for pattern discovery and pattern matching in music. In *Proceedings of the Dagstuhl Seminar on Content-Based Retrieval*, number 06171, 2006.
17. D. Meredith. A geometric language for representing structure in polyphonic music. In *International Society for Music Information Retrieval Conference*, 2012.
18. D. Meredith. Analysis by compression: Automatic generation of compact geometric encodings of musical objects. In *The Music Encoding Conference 2013*, 2013.
19. D. Meredith. COSIATEC and SIATECCompress: Pattern discovery by geometric compression. In *International Society for Music Information Retrieval Conference*, 2013.
20. D. Meredith. Compression-based geometric pattern discovery in music. In *Fourth International Workshop on Cognitive Information Processing*, 2014.
21. D. Meredith, K. Lemström, and G. A. Wiggins. Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music. *Journal of New Music Research*, 31(4):321–345, 2002.
22. F. Pachet. The continuator: Musical interaction with style. *Journal of New Music Research*, 32(3):333–341, 2003.
23. M. Pearce, D. Conklin, and G. Wiggins. Methods for combining statistical models of music. In *Computer Music Modeling and Retrieval*, pages 295–312. Springer, 2005.
24. K. Sayood. *Introduction to data compression*. Newnes, 2012.
25. D. Scheurleer. Preisfrage. *Zeitschrift der Internationalen Musikgesellschaft*, 1(7):219–220, 1900.
26. J. Seward. The bzip2 and libbzip2 official home page. *Online]. http://www. bzip. org*, 2000.
27. P. van Kranenburg, A. Volk, and F. Wiering. A comparison between global and local features for computational classification of folk song melodies. *Journal of New Music Research*, 42(1):1–18, 2013.
28. P. M. Vitányi and M. Li. Minimum description length induction, bayesianism, and Kolmogorov complexity. *Information Theory, IEEE Transactions on*, 46(2):446–464, 2000.
29. T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.
30. G. A. Wiggins, K. Lemström, and D. Meredith. Sia (m) ese: An algorithm for transposition invariant, polyphonic content-based music retrieval. In *ISMIR*. Citeseer, 2002.
31. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.
32. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536, 1978.

A Representations used

Input is a string $p_0, \cdots, p_l$ of two-dimensional points.

Name	Description
basic	The basic *pitch-time* representation, that is string of (*onset*, *pitch*) points
int	A string of (*onset*, *interval*) points: $$int(p_0) = p_0$$ $$int(p_n) = (p_n.onset,\ p_n.pitch - p_{n-1}.pitch)$$
*int*0	A string of (*onset*, *interval with the first note*) points: $$int0(p_0) = p_0$$ $$int0(p_n) = (p_n.onset,\ p_n.pitch - p_0.pitch)$$
intb	A string of (*onset*, *interval pointer*) points: $$intb(p_0) = p_0$$ $$intb(p_n) = \begin{cases} (p_n.onset,\ p_n.pitch - p_{n-1}.pitch) & \text{if it is the first time the interval occurs,} \\ (p_n.onset,\ n - j) & \text{otherwise.} \end{cases}$$ where j is the index of the last occurrence of the interval in the input.
ioi	For inter-onset interval. $$int(p_0) = p_0$$ $$ioi(p_n) = (p_n.onset - p_{n-1}.onset,\ p_n.pitch)$$
ioib	Same as intb but for inter onset intervals.

Table 4. The viewpoints used in the experiments.

B Parameters of LZ77

The LZ77 algorithm needs two parameters to be set:

– n, the size of the sliding window;

$$\cdots abc\boxed{defghijk}\boxed{abcdefgh}ijk\cdots$$

Fig. 4. Worst case of LZ77 algorithm: periodic input of period p with $p > n - L_S$.

– L_s, the size of the look-ahead buffer, $n - L_s$ is then the size of the dictionary.

The algorithm is based on the hypothesis that patterns will occur close together. Indeed there are a few cases where it does not work well. This can happen when the pattern that occurs again is no longer in the dictionary. The worst case of this is when the input is periodic with a period $p > n - L_S$ and contains a lot of different characters. As is shown in Figure 4, the algorithm cannot find the first letter in the dictionary and consequently cannot find any pattern.

It is therefore important to find good parameters for the analysis of the *Annotated Corpus*. To choose them, we ran the classification method on the *Annotated Corpus* several times with the single viewpoint $(\mathrm{LZ77}_{n,L_s}, \mathit{int} \circ \mathit{ioi})$. Each pair of parameters (n, L_s) produced a success rate, an average compression ratio on song files, and an average compression ratio on pair files. The results are reported in Table 5. This table shows that success rates tend to increase with the dictionary size. The size of the look-ahead buffer seems to have a similar effect on compression ratios. Moreover, the last line shows bad compression ratio but a good success rate. Bad compression ratios are due to the fact that the size of the dictionary is more than 99, and so the returned pointers, p_i, can have three figures instead of two. The good success rate can be explained by the fact that the dictionary is larger.

The best success rate is achieved with the parameters $n = 100$ and $L_s = 10$. Taking these parameter values is a good choice because it increases the success rate and the size of the dictionary allows the algorithm to find similarities between the two songs. Indeed, as the melodies have approximately 50 notes each, when LZ77 compresses the second song, the first one is still in the dictionary. Of course, for another corpus, which contains larger pieces, it would be better to choose a larger value of n.

n	L_S	*1-NN Leave-one-out SR*	CR_{AC}	CR_{pairs}
55	15	0.5472	1.3201	1.4433
70	20	0.7083	1.3376	1.4782
80	20	0.7444	1.3377	1.4875
100	10	$\boxed{0.8222}$	1.3151	1.4718
100	15	0.8056	1.3301	1.4881
100	25	0.7972	1.3378	1.4951
140	40	0.7611	1.4316	1.6710
150	30	0.8000	1.2680	1.4291

Table 5. Results of the classification method with the single viewpoint (LZ77$_{n,L_s}$, $int \circ ioi$). SR is Success Rate, CR_{AC} is the mean compression ratio on *Annotated Corpus*, CR_{pairs} is the average compression ratio on pair files used to compute the CCDs.