MUSIC ANALYSIS AND KOLMOGOROV COMPLEXITY

David Meredith
Aalborg University
dave@create.aau.dk

ABSTRACT

The goal of music analysis is to find the most satisfying explanations for musical works. It is proposed that this can best be achieved by attempting to write computer programs that are as short as possible and that generate representations that are as detailed as possible of the music to be explained. The theory of Kolmogorov complexity suggests that the length of such a program can be used as a measure of the complexity of the analysis that it represents. The analyst therefore needs a way to measure the length of a program so that this length reflects the quality of the analysis that the program represents. If such an effective measure of analysis quality can be found, it could be used in a system that automatically finds the optimal analysis for any passage of music. Measuring program length in terms of number of source-code characters is shown to be problematic and an expression is proposed that overcomes some but not all of these problems. It is suggested that the solutions to the remaining problems may lie either in the field of concrete Kolmogorov complexity or in the design of languages specialized for expressing musical structure.

1. INTRODUCTION

Bent [1, p. 1] defined *music analysis* as the "resolution of a musical structure into relatively simpler constituent elements, and the investigation of the functions of those elements within that structure". When attempting to find explanations for the structures of musical works, music analysts typically have two goals: first, they want to find explanations that are as *simple* as possible; and second, they want to account for as much *detail* as possible. These two goals often conflict: in order to account for more detail, a more complex explanation is usually required. The music analyst must therefore attempt to find an *optimal* explanation that strikes just the right balance between simplicity and level of detail. For some musical passages, there may be two or more distinct, but equally good, explanations. For example, two equally simple explanations might account for different (but equally important) aspects of the music's structure; or there may be two or more equally good ways of explaining the same structural aspects of a musical passage. In such cases, the music may give rise to a *multistable* percept, where it can be interpreted equally

satisfactorily in more than one way [2].

The view adopted here is that a musical analysis can be thought of as being an *algorithm* or *program* that, when executed, generates as output a *representation* of the music being analysed. Such a program therefore embodies an explanation for those structural aspects of the music that are encoded in the output representation. Kolmogorov complexity theory [3–7] suggests that the *length* of such a program can then be used as a measure of the complexity of its corresponding explanation: the shorter the program, the simpler—and, in general, the *better*—the explanation. The level of structural detail that the explanation accounts for corresponds to the level of detail with which the music is encoded in the representation generated by the program. Typically, much of the detailed structure in the music will not be encoded in the output of the program and will therefore go unexplained. It is assumed that the representation generated by the program will be an explicit, *in extenso* description of certain aspects of the structure of the music. The program can therefore be seen as being a compressed or compact encoding of the representation that it generates. On this view, the music analyst's goal is to find the shortest possible programs that generate the most detailed representations of musical passages, works and corpora.

Music analysis is about finding the best ways of interpreting musical works. In other words, it is concerned with finding the most satisfying *perceptual organizations* that are consistent with the musical "surface". This surface could be either the notated score or a particular performance. Of the two, a score will typically permit a higher number of consistent perceptual organizations, since the micro-structure of a performance will usually reflect the particular perceptual organization (i.e., interpretation) of the performer. Most theories of perceptual organization have been based on one of two principles: the *likelihood* principle of preferring the most *probable* interpretations (originally due to Helmholtz [8]); and the *minimum* [9] or *simplicity* [10] principle of preferring the *simplest* interpretations. Typically, statistical approaches to musical structure analysis (e.g., [11]) have applied the likelihood principle, whereas theories in the tradition of Gestalt psychology (e.g., [12]) apply the minimum principle. Indeed, as van der Helm and Leeuwenberg [9, p. 153] point out, the fundamental principle of Gestalt psychology, Koffka's [13] law of *Prägnanz*, which favours the simplest and most stable interpretation, can be seen as an "ancestor" of the minimum principle. For many years, the likelihood and minimum principles were considered by psychologists to be in competition. However, in 1996, Chater [10], drawing on results in Kolmogorov's [3] theory of complexity, pointed

out that the two principles are mathematically identical.

The work presented here relates closely to psychological *coding theories* of perceptual organization that employ the minimum or simplicity principle. In the coding theory approach, a *coding language* [9] or *pattern language* [14] is devised to represent the possible structures of patterns in a particular domain. The preferred organization is then the one that has the shortest encoding in the language. Coding theories of this type have been proposed to explain the perception of serial patterns [14–17], visual patterns [18, 19] and musical patterns [12, 16, 20, 21].

Chater [10] points out that two shortcomings of coding theories are, first, that each domain needs its own pattern language; and, second, that the length of an encoding depends to a certain extent on the specific design of the language used. Chater claims, however, that these problems can be overcome by applying the *invariance theorem* [5, p. 104–107], a central result of Kolmogorov complexity theory. This theorem states that the shortest description of any object is invariant up to a constant between different *universal languages*, a universal language being one that is rich enough to express partial recursive functions. Fortunately, all standard computer programming languages (e.g., C, Lisp, Java) are universal languages. This suggests that one might be able to meaningfully compare analyses of a given musical passage by comparing the lengths of programs representing these analyses, written in the same programming language. Moreover, if an objective, effective method can be found for measuring program length appropriately, then it becomes possible in principle to automate the process of searching for the best analysis of any given passage, piece or corpus of music.

In the remainder of this paper, I introduce the idea of representing a musical analysis (and therefore a particular interpretation of a passage of music) as a computer program. I then address the problem of using the *length* of such a program to evaluate its quality, relative to programs representing alternative analyses of the same musical structure.

2. REPRESENTING A MUSICAL ANALYSIS AS A PROGRAM

Consider Figure 1, which shows the left-hand part of bars 35 to 48 of Chopin's *Étude* in C major, Op. 10, No. 1. Suppose we segment the passage so that each note onset starts a new segment and then label each segment with the pitch class of the notes in it. If we then merge adjacent segments with the same pitch class content, then the resulting structure can be represented by the sequence of pitch classes,

$$9\ 2\ 7\ 0\ 5\ 11\ 4\ 9\ 2\ 7\ 0\ 5\ 11\ 4 \ . \tag{1}$$

One possible way of understanding this sequence is as 14 unrelated numbers. Writing the sequence out *in extenso* (as in (1)) expresses this interpretation. This interpretation requires 14 unrelated pieces of information to be remembered and encoded.

Sequence (1) can also be thought of as being con-

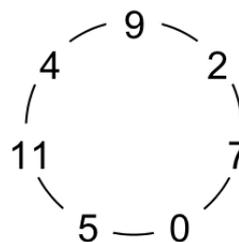**Figure 1**. The bass part of bars 35 to 48 of Chopin's *Étude* in C major, Op. 10, No. 1.

Figure 2. The diatonic fourths pitch class cycle in C major.

structed by repeating the much shorter sequence,

$$9\ 2\ 7\ 0\ 5\ 11\ 4 \ . \tag{2}$$

This seems to be a simpler and more satisfying way of understanding the sequence, since it recognizes some of the structure in it and requires us to remember only 8 pieces of information in order to reproduce it, namely, 7 unrelated numbers and the single operation of repeating this 7-number sequence.

A little more study reveals that sequence (2) is one turn around the diatonic fourths cycle in C major, shown in Figure 2. If p_i is the $(i + 1)$th element in sequence (1), then

$$p_i = (((i + 2) \bmod 7) * 5 + 11) \bmod 12 \ . \tag{3}$$

Sequence (1) could therefore be encoded using just this one formula, together with a specification that i should take values from 0 to 13.

However, Eq. 3 requires us to remember and encode 5 unrelated numbers, 5 unrelated operations and the two boundary values of i, making a total of 12 pieces of information. This interpretation therefore seems to be less parsimonious than expressing the structure as two copies of sequence (2).

Indeed, one could argue that the description in terms of Eq. 3 is even more complex than the literal description in (1). To reconstruct the structure from sequence (1), the only additional piece of procedural information required is that each element in the sequence is to be printed as it is scanned. Whereas, to reconstruct the pitch class structure using Eq. 3, one also needs to have an understanding of precedence rules, the definitions of the various operations used and a definition of some kind of iterating "for" loop construct that allows the formula to be applied to successive values of i between the boundary values.

The advantages of using the formula in Eq. 3 over a literal description like sequence (1) only become apparent if the sequence to be described is longer or if the formula can also be used to describe other musical passages. For

```
#include<stdio.h>
main(){
  printf("9 2 7 0 5 11 4 9 2 7 0 5 11 4");
}
```

Figure 3. C program that literally prints the pitch class structure of the passage in Figure 1.

```
#include<stdio.h>
main(){
  char i;
  for(i=0;i<14;i++)
    printf("%d ",(((i+2)%7)*5+11)%12);
}
```

Figure 4. C program that generates the pitch class structure of the passage in Figure 1 using the diatonic fourths cycle in C major, shown in Figure 2.

example, an *in extenso* description of a sequence consisting of *four* turns around the C major diatonic fourths circle would require encoding 28 pieces of information, whereas a description in terms of Eq. 3 would require no more information than that needed to encode sequence (1). In other words, if the length of such a sequence is n, then the length of an explicit description is $O(n)$ whereas the length of a description using Eq. 3 is constant, i.e., $O(1)$.

Another advantage of a description in terms of Eq. 3 is that it can be parametrized and *reused* (with only minor changes) to describe *any* sequence of pitch classes formed by circulating around a diatonic fourths cycle *of any length and in any key*.

To make the discussion more concrete, consider Figure 3, which shows a C program that literally prints out sequence (1), and Figure 4 which uses Eq. 3 to calculate each value in the sequence. If we format these programs so that they are as short as possible and ignore the initial line that loads stdio.h, then the program in Figure 4 is 66 characters long, whereas that in Figure 3 is only 48 characters long. Kolmogorov complexity theory tells us that the complexity of an object is related to the length of the shortest program that can generate it. The fact that the program in Figure 4 is apparently longer than that in Figure 3 therefore seems to support the suggestion made above that a literal description of sequence (1) is actually more parsimonious than one that employs Eq. 3.

However, if we were to cycle around the C major diatonic fourths circle four times rather than just twice, then we would generate the pitch class sequence

$$9\,2\,7\,0\,5\,11\,4\,9\,2\,7\,0\,5\,11\,4\,9\,2\,7\,0\,5\,11\,4\,9\,2\,7\,0\,5\,11\,4\,. \quad (4)$$

This sequence is literally printed out by the C program in Figure 7 and generated using Eq. 3 by the program in Figure 5.

If the programs in Figure 5 and Figure 7 are formatted so that they are as short as possible and the initial line that loads stdio.h is ignored, then the program in Figure 5 is shorter (66 characters) than that in Figure 7 (78 characters), supporting the point made above that the advantage of using Eq. 3 only becomes apparent when the sequence to be generated is somewhat longer than sequence (1).

As suggested above, we can also refactor Figure 5 so that we define a separate function that encapsulates Eq. 3

```
#include<stdio.h>
main(){
  char i;
  for(i=0;i<28;i++)
    printf("%d ",(((i+2)%7)*5+11)%12);
}
```

Figure 5. C program that generates sequence (4) using the diatonic fourths cycle in C major, shown in Figure 2.

```
#include<stdio.h>

f(char k, char s, char l) {
  char i;
  for(i=0;i<l;i++)
    printf("%d ",(((i+s)%7)*5+k+11)%12);
}

main(){
  f(0,2,14);
}
```

Figure 6. C program that generates sequence (1) using a function, $f(k, s, \ell)$, such that k is the pitch class of the tonic of the major key of the diatonic set, s is the starting index in the cycle (0 being 11 more than the tonic pitch class) and ℓ is the length of the sequence.

and allows us to generate any sequence formed by circulating around a diatonic fourths cycle in any key. This has been done in Figure 6. Note that, *given the function f*, defined in Figure 6, we can now describe sequence (1) using the very short expression $f(0, 2, 14)$. We can also describe any other diatonic fourths cycle sequence using a similarly short expression. For example, the expression $f(5, 4, 12)$ generates the sequence

$$0\,5\,10\,4\,9\,2\,7\,0\,5\,10\,4\,9 \quad (5)$$

which is 12 consecutive steps around the F major diatonic fourths circle, starting on C.

However, the program in Figure 6 is even longer (102 characters) than that in Figure 4. Thus, the ability to encode any member of a whole class of commonly-occurring sequences by a parsimonious expression comes at the price of having to define a custom function that must be included in the "background" knowledge required for these parsimonious expressions to be decoded.

Nevertheless, a music analyst and a software engineer would probably agree that, of the descriptions offered above for sequence (1), the program in Figure 6 is the most elegant, insightful and satisfying, despite being the longest (at least in terms of characters). This seems to be in large part due to the following:

1. the function f provides a very compact description of an infinite set of structures of a type that commonly occurs in music; and

2. the function f allows each member of the set that it defines to be encoded with a very short description, given the definition of f.

In other words, a music analyst would prefer the description in Figure 6 because the function f is a short program that allows *many* musical structures to be described parsimoniously and thus provides a very simple explanation for

```
#include<stdio.h>
main(){
  printf("9 2 7 0 5 11 4 9 2 7 0 5 11 4 9 2 7 0 5 11 4 9 2 7 0 5 11 4");
}
```

Figure 7. C program that literally prints sequence (4).

a large set of musical passages of which the one in Figure 1 is an example.

However, if the pitch class structure of the passage in Figure 1 were the *only* passage in the musical literature that could be generated by a call to f, then we could not reasonably consider the program in Figure 6 to be a good analysis of this structure. And this seems to be reflected in the fact that Figure 6 is longer than, for example, Figure 3. In order to justify the claim that function f is a good analytical idea, we would have to show that it could be used to produce parsimonious descriptions (i.e., good explanations) of some other musical passages. That is, we would have to write another program that uses the function f to generate the pitch class structures of several passages (that actually occur in interesting musical works). This program would demonstrate the value of function f and this new program as a whole would represent a higher quality analysis of the structures that it generates as output than the program in Figure 6. Moreover, the superiority of this new program would be reflected in the fact that the length of its output would be considerably longer than that of Figure 6, while the length of the two programs would be almost the same.

The foregoing discussion suggests that the quality of the analysis represented by a program P that generates an output X, increases with the value ℓ_{P_0}/ℓ_P where ℓ_{P_0} is the length of a program that just literally prints the same output as P (i.e., X), and ℓ_P is the length of P itself. P is clearly an *encoding* of X and ℓ_{P_0}/ℓ_P is the *compression ratio* achieved on X by the encoding P. The music analyst's goal is therefore to losslessly *compress* music as much as possible, or, more simply, *to find the shortest program that generates the music to be explained*. This is a special case of Occam's razor which states that one should prefer the simplest theory that explains the facts, and Kolmogorov complexity explicates the notion of theory complexity as the length in bits of the shortest description of the theory in terms of a binary program on a universal computer [5, pp. 341–343].

3. MEASURING THE LENGTH OF A PROGRAM

If we accept that the goal of music analysis is to find the shortest program that generates the music to be explained, then a music analyst is frequently going to be faced with the problem of deciding which of two programs that generate the same output is the shorter. In cases where the output of one program is a proper subset of the output of another, then the analyst will need to decide which of the two programs (that now generate *different* outputs) achieves the better compression. The analyst therefore needs to be able to measure and compare the lengths of programs.

In the examples above, the length of each program has been given in terms of the number of characters in the most compactly formatted version of the program's source code, written in C. Clearly, one would not count unnecessary whitespace characters or characters within comments; notwithstanding, there are still several reasons why source-code length in characters cannot be used directly to measure the quality of the analysis that a program represents.

One obvious reason for this is that user-defined identifiers for variables, constants and functions can be of arbitrary length. Thus, if program P_1 has fewer characters than P_2, we can easily make P_1 longer than P_2 simply by lengthening the names of the variables in P_1, which clearly does not change the quality of the analysis represented by P_1. Suppose we try to solve this by stipulating that each user-defined identifier only counts as 1 character. This rule has been used by Zenil and Delahaye [22] in their competition to find the shortest universal Turing machine implementation. But what if a program has more identifiers than there are characters in the character set being used? Indeed, not even all the characters in the character set can be made available for use as identifiers—digit characters cannot be used and others must be reserved for operators and reserved words (e.g., '+', '-', '[' etc. in the C programming language). In fact, in ANSI C, there are only 53 permissible 1-character identifiers (the Roman letters and '_') [23, p. 192].

It is actually only justified to count 1 character for each user-defined identifier if the number of distinct tokens used by the program (excluding literal strings and numbers) is no more than the number of characters in the character set. Such tokens include

- the names of functions, constants and variables—both predefined (e.g., 'printf') and user-defined (e.g., 'f', 'i' in Figure 6);

- reserved words for constructs and types (e.g., 'for', 'int');

- required punctuation marks, such as the 'end of statement' character (';' in C) and commas;

- operators (e.g., '=', '+').

C uses seven-bit ASCII [23, p. 229] so the character set in this case contains only 128 characters. It is therefore easy to see how the number of distinct tokens may exceed the number of available characters.

Another way of measuring the length of a token is in terms of the number of bits required to uniquely identify it among the complete set of distinct tokens used by (or available to) the program. We would therefore need to know the minimum value of n such that we can assign a unique bit string of length n to each distinct token (excluding literal numbers and strings). If there are N_{tok} such distinct tokens, then this minimum value of n is given by

$n_{\text{tok}} = \lceil \log_2 N_{\text{tok}} \rceil$, which is the amount of Shannon information in the token, assuming all tokens are equally probable [24]. Similarly, the length of a string can be measured in terms of the amount of information in it. Thus, if N_{chr} is the size of the set of permissible string characters, then the least number of bits capable of encoding a string character is $n_{\text{chr}} = \lceil \log_2 N_{\text{chr}} \rceil$ and the length in bits of (i.e., the information in) a string containing ℓ characters is ℓn_{chr}. The length in bits of a *literal* number can be defined to be the size of the number's type. For example, in C, typically a long uses 8 bytes or 64 bits of memory, an int uses 32 bits and a char uses 8 bits. [1] Putting all this together, we get the following expression for estimating the *length* of a C program, P, in bits:

$$\ell(P) = n_{\text{tok}} N_{\text{tok}}(P) + n_{\text{chr}} \sum_{i=1}^{N_{\text{str}}(P)} \ell_{\text{chr}}(s_i) + \sum_{i=1}^{N_{\text{num}}(P)} \ell(n_i) , \tag{6}$$

where

- $N_{\text{tok}}(P)$ is the number of tokens in P that are neither literal strings nor literal numbers;

- $N_{\text{str}}(P)$ is the number of literal strings in P;

- $\ell_{\text{chr}}(s_i)$ is the length *in characters* of the ith string in P;

- $N_{\text{num}}(P)$ is the number of literal numbers in P; and

- $\ell(n_i)$ is the size in bits of the type of the ith literal number in P.

The total number of distinct tokens, N_{tok}, is the sum of the number of tokens defined within the language itself (including any libraries loaded), which we can denote by $N_{\text{tok}}^{\text{L}}$, and the number of user-defined tokens in the program, $N_{\text{tok}}^{\text{P}}$. Therefore $n_{\text{tok}} = \lceil \log_2(N_{\text{tok}}^{\text{L}} + N_{\text{tok}}^{\text{P}}) \rceil$. If we assume that the program is written in C and that the stdio.h library is loaded, then $N_{\text{tok}}^{\text{L}}$ is about 150. So for all the programs given in this paper, $n_{\text{tok}} = 8$. Table 1 shows the lengths of the programs presented above in terms of number of characters ($\ell_{\text{chr}}(P)$) and in terms of bits using Eq. 6 ($\ell(P)$). Note that the initial line in each program that loads the stdio.h library has been ignored when calculating the values in this table.

As can be seen in the third column of Table 1, measuring program length using Eq. 6 generally produces a lower value than measuring in terms of the number of characters in the most compactly formatted version of the source code. The values in the third column also show that the proportional reduction in the length value is greater for those programs that capture some of the structure in the pitch class sequence (Figures 4, 5 and 6) than for those that literally print the output (Figures 3 and 7). Moreover, measuring in terms of Eq. 6 solves the problem of arbitrary identifier length.

---

[1] These values are actually implementation-dependent. The sizeof function can be used to determine the sizes of these types in bytes on a particular system.

Program, P	$7\ell_{\text{chr}}(P)$	$\ell(P)$	$\ell(P)/(7\ell_{\text{chr}}(P))$
Figure 3	336	291	0.87
Figure 4	462	381	0.82
Figure 5	462	381	0.82
Figure 7	546	501	0.92
Figure 6	714	573	0.80

Table 1. Lengths of programs in this paper. $\ell_{\text{chr}}(P)$ is the length in characters of the most compactly formatted version of the program. This value is multiplied by 7 because C uses the 7-bit ASCII character set. $\ell(P)$ is the length in bits calculated using Eq. 6. Both measures ignore the line that imports the stdio.h library.

4. DEPENDENCY OF PROGRAM LENGTH ON LANGUAGE

Although Eq. 6 overcomes the problem of identifiers being of an arbitrary length, it does not solve a deeper problem caused by the fact that a program's length depends on the syntax of the language in which it is written and the constructs that this language makes available. For example, programs for symbolic manipulation tend to be shorter in Lisp than in Fortran whereas the reverse is true for programs that carry out numerical computations. If every program in Lisp were, say, 70% of the length of the "equivalent" program in Fortran, then this would not be a problem in the present context, because we are here only concerned with the *relative* lengths of programs written in the *same* language. However, the situation is not as simple as this. In fact, if we have two Lisp programs, P_1^L, P_2^L, and two equivalent Fortran programs, P_1^F, P_2^F, then, in general, it is possible for P_1^L to be shorter than P_2^L while P_1^F is longer than P_2^F, even if we measure program length in a way that takes into account the arbitrariness of identifier length. We are also here glossing over the non-trivial problem of defining when two programs in different languages are "equivalent".

We therefore need a method of measuring the complexity of a musical analysis that is either independent of the language in which the analysis is expressed or based on some generally-agreed 'reference' language. An obvious place to look for such a measure is in the theory of Kolmogorov complexity.

5. KOLMOGOROV COMPLEXITY

Let ϕ_0 be an additively optimal universal partial recursive function that we call the *reference function* and let U be a Turing machine that computes ϕ_0 that we call the *reference machine*. The *conditional Kolmogorov complexity* of an object x given an object y is defined to be

$$C_{\phi_0}(x|y) = \min\{l(p) : \phi_0(\langle y, p \rangle) = x\}, \tag{7}$$

where

- $l(p)$ is the length in bits of the program p that computes x on the reference machine U when given input y; and

- ⟨y, p⟩ is a function that maps the input y and the program p onto a single binary string.

x, y and p are understood to be represented as binary strings. The *(unconditional) Kolmogorov complexity* of an object x is then given by

$$C(x) = C_{\phi_0}(x|\epsilon) \qquad (8)$$

where ϵ is the empty string. Note that the Kolmogorov complexity of an object is always defined relative to some specified reference function which defines a particular description method which must be an additively optimal universal partial recursive function [5, pp. 104–106].

The *invariance theorem* [5, pp. 105] tells us that, within the set of partial recursive functions that compute x given y, there exists an additively optimal universal partial recursive function, ϕ_0. It follows that $C_{\phi_0}(x|y) \leq C_\phi(x|y) + c_\phi$ for all partial recursive functions ϕ and all x and y, where c_ϕ is a constant that depends only on ϕ. It can then be shown that, if ϕ and ϕ' are two additively optimal functions, then

$$|C_\phi(x|y) - C_{\phi'}(x|y)| \leq c_{\phi\phi'} \qquad (9)$$

where $c_{\phi\phi'}$ is a constant that depends only the choice of ϕ and ϕ'. In other words, the complexity of x given y is independent of the description method up to a fixed constant for all inputs and outputs. Such a description language could be a standard programming language such as Lisp or C. Thus if $C_{\text{Lisp}}(x|y)$ and $C_{\text{C}}(x|y)$ are the complexities of x given y in Lisp and C, respectively, then the invariance theorem tells us that $|C_{\text{Lisp}}(x|y) - C_{\text{C}}(x|y)| \leq c$ where c is a constant. In other words, the complexity of x given y will be the same, up to a constant, regardless of the language in which we express the program that computes x. However, crucially, *we do not know whether the shortest C program that computes x will be longer or shorter than the shortest Lisp program that computes x*. The invariance theorem therefore helps us if we are interested in knowing approximately the length of the *shortest* program that computes a given object (i.e., its Kolmogorov complexity). However, it does *not* help us directly if we want to be able to meaningfully compare the lengths of a number of different programs that compute some given object x, even if all the programs are written in the same universal programming language. This casts doubt on Chater's [10] claim that the invariance theorem solves the problem of the language-dependency of encoding length.

One way forward might be to adopt the approach taken by Chaitin [25,26] and Tromp [27] who have defined short, concrete universal machines implemented from weak elementary operations (see also [5, pp. 206–211]). However, such an approach would require the expression of musical analyses in a rather low-level language based on lambda calculus or combinatory logic, which a music analyst might find to be too cumbersome. A more practical approach might therefore be to return to the development of coding languages specially designed for expressing musical structure, as has been done recently by Meredith [21].

6. CONCLUSIONS AND FUTURE DIRECTIONS

I have proposed that the goal of music analysis should be to devise the shortest possible programs that generate the music to be explained. The reasoning behind this is that a short program that generates a given passage, gives more insight into the structure of that passage and a more satisfying explanation of it than a longer program. This is just a special case of Occam's razor. If we accept this as the goal of music analysis, then, given two programs that generate the same passage of music, an analyst needs to be able to decide which of the two is shorter. This, in turn, implies that the music analyst must be able to measure the length of a program in such a way that this length accurately reflects the quality of the analysis represented by the program. I have shown that measuring program length in terms of number of source-code characters is problematic and I have tentatively proposed an expression for measuring program length that overcomes the problem of arbitrary identifier length. However, there remains a deeper problem caused by the fact that program length depends on the syntax of the programming language and the constructs that the language makes available. This problem is as follows: if P_1^A, P_2^A are two programs in language A and P_1^A is shorter than P_2^A and P_1^B, P_2^B are two programs in language B equivalent to P_1^A, P_2^A, respectively, then it is possible for P_1^B to be *longer* than P_2^B. This means that whether or not one program is shorter than another depends on the language in which the two programs are written. I have shown that the invariance theorem in Kolmogorov complexity theory does not help to resolve this problem. However, it may be possible to find a solution to the program-length measuring problem in the field of concrete Kolmogorov complexity where the goal is to design concrete universal machines implemented from weak elementary operations. An alternative approach would be to return to the development of coding languages that are specially designed for expressing musical structure.

7. REFERENCES

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

[2] P. Kruse and M. Stadler, eds., *Ambiguity in Mind and Nature: Multistable Cognitive Phenomena*, vol. 64 of *Springer Series in Synergetics*. Berlin: Springer, 1995.

[3] A. N. Kolmogorov, "Three approaches to the quantitative definition of information," *Problems of Information Transmission*, vol. 1, no. 1, pp. 1–7, 1965.

[4] G. J. Chaitin, "On the length of programs for computing finite binary sequences," *Journal of the Association for Computing Machinery*, vol. 13, no. 4, pp. 547–569, 1966.

[5] M. Li and P. Vitányi, *An Introduction to Kolmogorov Complexity and Its Applications*. Berlin: Springer, third ed., 2008.

[6] R. J. Solomonoff, "A formal theory of inductive inference (Part I)," *Information and Control*, vol. 7, no. 1, pp. 1–22, 1964.

[7] R. J. Solomonoff, "A formal theory of inductive inference (Part II)," *Information and Control*, vol. 7, no. 2, pp. 224–254, 1964.

[8] H. L. F. von Helmholtz, *Treatise on Physiological Optics*. New York: Dover, 1910/1962. Trans. and ed. by J. P. Southall. Originally published in 1910.

[9] P. A. van der Helm and E. L. Leeuwenberg, "Accessibility: A criterion for regularity and hierarchy in visual pattern codes," *Journal of Mathematical Psychology*, vol. 35, pp. 151–213, 1991.

[10] N. Chater, "Reconciling simplicity and likelihood principles in perceptual organization," *Psychological Review*, vol. 103, no. 3, pp. 566–581, 1996.

[11] D. Temperley, *Music and Probability*. Cambridge, MA.: MIT Press, 2007.

[12] D. Deutsch and J. Feroe, "The internal representation of pitch sequences in tonal music," *Psychological Review*, vol. 88, no. 6, pp. 503–522, 1981.

[13] K. Koffka, *Principles of Gestalt Psychology*. New York: Harcourt Brace, 1935.

[14] H. A. Simon, "Complexity and the representation of patterned sequences of symbols," *Psychological Review*, vol. 79, no. 5, pp. 369–382, 1972.

[15] E. L. L. Leeuwenberg, "Quantitative specification of information in sequential patterns," *Psychological Review*, vol. 76, no. 2, pp. 216–220, 1969.

[16] F. Restle, "Theory of serial pattern learning: Structural trees," *Psychological Review*, vol. 77, no. 6, pp. 481–495, 1970.

[17] P. C. Vitz and T. C. Todd, "A coded element model of the perceptual processing of sequential stimuli," *Psychological Review*, vol. 76, no. 5, pp. 433–449, 1969.

[18] J. Hochberg and E. McAlister, "A quantitative approach to figural "goodness"," *Journal of Experimental Psychology*, vol. 46, no. 5, pp. 361–364, 1953.

[19] E. L. J. Leeuwenberg, "A perceptual coding language for visual and auditory patterns," *American Journal of Psychology*, vol. 84, no. 3, pp. 307–349, 1971.

[20] H. A. Simon and R. K. Sumner, "Pattern in music," in *Formal representation of human judgment* (B. Kleinmuntz, ed.), New York: Wiley, 1968.

[21] D. Meredith, "A geometric language for representing structure in polyphonic music," in *Proceedings of the 13th International Society for Music Information Retrieval Conference (ISMIR 2012)*, (Porto, Portugal), 2012.

[22] H. Zenil and J.-P. Delahaye, "The shortest universal machine implementation," 2008. `http://www.mathrix.org/experimentalAIT/TuringMachine.html`. Accessed 28 July 2012.

[23] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ.: Prentice Hall, 1988.

[24] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[25] G. J. Chaitin, "A new version of algorithmic information theory," *Complexity*, vol. 1, no. 4, pp. 55–59, 1995/1996.

[26] G. J. Chaitin, "How to run algorithmic information theory on a computer: Studying the limits of mathematical reasoning," *Complexity*, vol. 2, no. 1, pp. 15–21, 1996.

[27] J. Tromp, "Binary lambda calculus and combinatory logic," 2011. Available at <`http://homepages.cwi.nl/~tromp/cl/LC.ps`>. Accessed 30 July 2012.