Internat. J. Math. & Math. Sci.
VOL. 15 NO. 1 (1992) 161-174

MEASURING STATIC COMPLEXITY

BEN GOERTZEL

Department of Mathematics
University of Nevada, Las Vegas
Las Vegas NV 89154

(Received August 17, 1990 and in revised form March 1, 1991)

ABSTRACT: *The concept of "pattern" is introduced, formally defined, and used to analyze various measures of the complexity of finite binary sequences and other objects. The standard Kolmogoroff-Chaitin-Solomonoff complexity measure is considered, along with Bennett's 'logical depth', Koppel's 'sophistication', and Chaitin's analysis of the complexity of geometric objects. The pattern-theoretic point of view illuminates the shortcomings of these measures and leads to specific improvements. It gives rise to two novel mathematical concepts -- "orders" of complexity and "levels" of pattern, and it yields a new measure of complexity, the "structural complexity", which measures the total amount of structure an entity possesses.*

KEY WORDS AND PHRASES. *Kolmogorov complexity, algorithmic information, pattern, sophistication, structure, depth*
AMS SUBJECT CLASSIFICATION CODE. *68Q30*

1. INTRODUCTION

Different contexts require different concepts of "complexity". In the theory of computational complexity, as outlined for instance by Kronsjo [1], one deals with the complexity of problems. And the complexity of evolving systems falls under the aegis of dynamical systems theory, as considered for example by Bowen [2]. The present paper, however, is concerned with the complexity of static objects, a subject which has received rather little attention. Although most of the discussion focuses on binary sequences, the implications are much more general.

The first mathematically precise measure of the complexity of static objects was invented simultaneously by Kolmogorov [3], Chaitin [4] and Solomonoff [5].

DEFINITION 1. Let M be a universal Turing machine. Let us say that a program for M is self-delimiting if it contains a segment telling M its total length in bits. Then, the KCS complexity of a finite binary sequence x is the length of the shortest program which computes x on M.

In the decades since its conception, this definition has led to a number of interesting developments. Chaitin [4] has used it to provide an interesting new proof of Godel's theorem; and Bennett [6], Zurek [7] and others have applied it to problems in thermodynamics. However, it has increasingly been realized that the concept of KCS complexity fails to capture the intuitive meaning of "complexity."

The problem is that, according to the KCS definition, "random", structureless sequences are judged the most complex. The least complex sequences are those like 000000000...000, 010101010101...010101, and 10100100010000100000...0, which can be computed by very short programs. And the most complex sequences x are those which cannot be computed by any program shorter than "print x". There is a sense in which this is not a desirable property for a definition of complexity to have -- in which a human or a tree or the sequence of prime numbers is more "complex" than a random sequence. Over the past decade, there have been two noteworthy attempts to remedy this deficiency: Bennett's [6] "logical depth", and Koppel's [8] "sophistication."

We outline a general mathematical framework within which various measures of complexity may be formulated, analyzed and compared. This approach yields significant modifications of these measures, as well as several novel, general concepts for the analysis of complexity. Furthermore, it gives rise to an entirely new complexity measure, the "structural complexity", which measures the total amount of structure an entity possesses. Intuitively, this tells one "how much there is to say" about a given object.

2. PATTERN

DEFINITION 2. A pattern space is a set (S,,| |), where S is a set, * is a binary operation defined on some subset of SxS, and | | is a map from S into the nonnegative real numbers.*

Let us consider a simple example: Turing machines and finite binary sequences.

*DEFINITION 3. Let y be a program for a universal Turing machine; let z be a finite binary sequence. Define y*z to be the binary sequence which appears on the memory tape of the Turing machine after, having been started with z on its input tape beginning directly under the tape head and extending to the right, program y finishes running. If y never stops running, then let y*z be undefined. Let $|z|_T$ denote the length of z as its length, and let $|y|_T$ denote the length of the program y.*

Now we are ready to give a general definition of pattern.

*DEFINITION 4. Let a, b, and c denote constant, nonnegative numbers. Then an ordered pair (y,z) is a pattern in x if x=y*z and $a|y| + b|z| + cC(y,z) < |x|$, where C(y,z) denotes the complexity of obtaining x from (y,z).*

DEFINITION 6. If y is a Turing machine program and z is a finite binary sequence, $C_T(y,z)$ denotes the number of time steps which the Turing machine takes to stop when

equipped with program y and given z as initial input.

For many purposes, the numbers a, b and c are not important. Often they can all be taken to equal 1, so that they do not appear in the formula at all. But in some cases it may be useful to, for instance, set a=b=1 and c=0. Then the formula reads $|y| + |z| < |x|$. *The constants could be dispensed with, but then it would be necessary to redefine* $|\ |$ *and C more often.*

Intuitively, an ordered pair (y,z) is a pattern in x if the complexity of y, plus the complexity of z, plus the complexity of getting x out of y and z, is less than the complexity of x. In other words, an ordered pair (y,z) is a pattern in x if it is simpler to represent x in terms of y and z than it is to say "x". The constants a, b and c are, of course, weights: if a=3/4 and b=5/4, for example, then the complexity of a is counted less than the complexity of b.

The definition of pattern can be generalized to ordered n-tuples, and to take into account the possibility of different kinds of combination, say $*_1$ *and* $*_2$.

DEFINITION 7: An ordered set of n entities $(x_1, x_2, ..., x_n)$ *is a pattern in x if* $x = x_1 *_1 x_2 *_2 ... *_n x_n$ *and* $a_1|x_1| + a_2|x_2| + ... + a_n|x_n| + a_{n+1} < |x|$, *where* $C(x_1, ..., x_n)$ *is the complexity of computing* $x_1 *_1 x_2 *_2 ... *_n x_n$ *and* $a_1, ..., a_{n+1}$ *are nonnegative numbers.*

Also, the following concept will be of use:

DEFINITION 8: The intensity in x of a ordered pair (y,z) such that $y*z = x$ *may be defined as* $IN[(y,z)|x] = (|x| - [a|y| + b|z| + cC(y,z)])/|x|$.

Obviously, this quantity is positive whenever (y,z) is a pattern in x, and negative or zero whenever it is not; and its maximum value is 1.

3. AN EXAMPLE: GEOMETRIC PATTERN

Most of the present paper is devoted to Turing machines and binary sequences. However, the definition of pattern does not involve the theory of computation; essentially, a pattern is a "representation as something simpler". Instead of Turing machines and binary sequences let us now consider pictures. Suppose that A is a one inch square picture, and B is a five inch square picture made up of twenty-five non-overlapping one-inch pictures identical to A. Intuitively, it is simpler to represent B as an arrangement of copies of A, than it is to simply consider B as a "thing in itself". Very roughly speaking, it would seem likely that part of the process of remembering what B looks like consists of representing B as an arrangement of copies of A.

This intuition may be expressed in terms of the definition of pattern. Where x and y are square regions, let:

$y *_1 z$ *denote the region obtained by placing y to the right of z*
$y *_2 z$ *denote the region obtained by placing y to the left of z*
$y *_3 z$ *denote the region obtained by placing y below z*
$y *_4 z$ *denote the region obtained by placing y above z*

And, although this is obviously a very crude measure, let us define the complexity $|x|$ *of a square region with a black-and-white picture drawn in it as the proportion of the region covered with black. Also, let us assume that two pictures are identical if one can be obtained by a rigid motion of the other.*

The operations $*_1$, $*_2$, $*_3$ and $*_4$ may be called simple operations. Compound operations are, then, compositions of simple operations, such as the operation $(x*_1 w)*_2 x)*_4 w$. If y is a compound operation, let us define its complexity $|y|$ to be the length of the shortest program which computes the actual statement of the compound operation. For instance, $|(x*_1 w*_2 x)*_4 w|$ is defined to be the length of the shortest program which outputs the sequence of symbols "$(x*_1 w*_2 x)*_4 w$".

*Where y is a simple operation and z is a square region, let y*z denote the region that results from applying y to z. A compound operation acts on a number of square regions. For instance, $(x*_1 w*_2 x)*_4 w$ acts on w and x both. We may consider it to act on the ordered pair (x,w). In general, we may consider a compound operation y to act on an ordered set of square regions $(x_1, x_2, ..., x_n)$, where x_1 is the letter that occurs first in the statement of y, x_2 is the letter that occurs second, etc. And we may define $y*(x_1, ..., x_n)$ to be the region that results from applying the compound operation y to the ordered set of regions $(x_1, ..., x_n)$.*

Let us return to the two pictures, A and B, discussed above. Let $q=A*_1 A*_2 A*_3 A*_4 A$. Then, it is easy to see that $B=q*_4 q*_4 q*_4 q*_4 q$. In other words, $B = (A*_1 A*_2 A*_3 A*_4 A)*_4 (A*_1 A*_2 A*_3 A*_4 A)*_4 (A*_1 A*_2 A*_3 A*_4 A)*_4 (A*_1 A*_2 A*_3 A*_4 A)*_4 (A*_1 A*_2 A*_3 A*_4 A)$. Where y is the compound operation given in the previous sentence, we have $B=y*A$. The complexity of that compound operation, $|y|$, is certainly very close to the length of the program "Let $q=A*_1 A*_2 A*_3 A*_4 A$; print $q*_4 q*_4 q*_4 q*_4 q$". Note that this program is shorter than the program "Print$(A*_1 A*_2 A*_3 A*_4 A)*_4 (A*_1 A*_2 A*_3 A*_4 A)*_4 (A*_1 A*_2 A*_3 A*_4 A)*(A*_1 A*_2 A*_3 A*_4 A)*_4 (A*_1 A*_2 A*_3 A*_4 A)$", so it is clear that the latter should not be used in the computation of $|y|$.

We have not yet discussed the term $C(y,(B_1, ..., B_n))$, which represents the amount of effort required to execute the compound operation y on the regions $(x_1, ..., x_n)$. For simplicity's sake, we shall simply set it equal to the number of times the symbol "" appears in the statement of y; that is, to the number of simple operations involved in y.*

So, is (y,A) a pattern in B? Let us assume that the constants a, b and c are all equal to 1. We know $y*A=B$; the question is whether $|y|+|A|+C(y,A) < |B|$.

According to the above definitions, $|y|$ is 37 symbols long. Obviously this is a matter of the particular notation being used. For instance, it would be less if only one character were used to denote $*_1$, and it would be more if it were written in binary code.

C(y,z) is even easier to compute: there are 24 simple operations involved in the construction of B from A.

So we have, very roughly speaking, $37 + |z| + 24 < |x|$. This is the inequality that must be satisfied if (y,z) is to be considered a pattern in x. Rearranging, we find: $|z| < |x| - 61$. Recall that we defined the complexity of a region as the proportion of black which it contains. This means that (y,z) is a pattern in x if and only if it the amount of black required to draw B exceeds amount of black required to draw A by at least 62. Obviously, whether or not this is the case depends on the units of measurement.

This is a very simple example, in that the compound operation y involves only one

region. In general, we may define $|(x_1,...,x_n)| = |x_1| + ... + |x_n|$, assuming that the amount of black in a union of disjoint regions is the sum of the amounts of black in the individual regions. From this it follows that $(y,(x_1,...,x_n))$ is a pattern in x if and only if $a|y| + b(|x_1| + ... + |x_n|) + cC(y,(x_1,...,x_n)) < |x|$.

*Results similar to these could also be obtained from a different sort of analysis. In order to deal with regions other than squares, it is desirable to replace $*_1$, $*_2$, $*_3$, $*_4$ with a single "joining" operation *, namely the the set-theoretic union U. Let $z=(x_1,...,x_n)$, let y be a Turing machine, let f be a method for converting a picture into a binary sequence, and let g be a method for converting a binary sequence into a picture. Then we have*

DEFINITION 9: If $x = x_1 \cup x_2 \cup ... \cup x_n$, then (y,z,f,g) is a pattern in x if $a|y| + b|z| + c|f| + d|g| + eC(y,z,f,g) < |x|$.

We have not said how $|f|$ and $|g|$ are to be defined. However, this would require a detailed consideration of the geometric space containing x, and that would take us too far afield. This general approach is somewhat similar to that taken in Chaitin [9].

4. ORDERS OF COMPLEXITY

It should be apparent from the foregoing that complexity and pattern are deeply interrelated. In this and the following sections, we shall explore several different approaches to measuring complexity, all of which seek to go beyond the simplistic KCS approach. According to the KCS approach, complexity means structurelessness. The most "random", least structured sequences are the most complex. The formulation of this approach was a great step forward. But it seems clear that the next step is to give formulas which capture more of the intuitive meaning of the word "complexity".

First, we shall consider the idea that pattern itself may be used to define complexity. Recall the geometric example of the previous section, in which the complexity of a black-and-white picture in a square region was defined as the amount of black required to draw it. This measure did not even presume to gauge the effort required to represent a black-and-white picture in a square region. One way to measure the effort required to represent such a picture, call it x, is to look at all compound operations y, and all sets of square black-and-white pictures $(x_1,...,x_n)$, such that $y*(x_1,...,x_n)=x$. One may then ask which y and $(x_1,...,x_n)$ give the smallest value of $a|y| + b(|x_1| + ... + |x_n|) + cC(y,(x_1,...,x_n))$. This minimal value of $a|y| + b(|x_1| + ... + |x_n|)$ may be defined to be the "second-order" complexity of x. The second-order complexity is then be a measure of how simply x can be represented -- in terms of compound operations on square regions.

In general, given any complexity measure $|\ |$, we may use this sort of reasoning to define a complexity measure $|\ |'$.

DEFINITION 10: If $|\ |$ is a complexity measure, $|\ |'$ is the complexity measure defined so that, for all x, $|x|'$ is the smallest value that the quantity $a|y| + b|z| + cC(y,z)$ takes on, for any (y,z) such that $y*z=x$.

$|x|'$ measures how complex the simplest representation of x is, where complexity is measured by $|\ |$. Sometimes, as in our geometric example, $|\ |$ and $|\ |'$ will measure very different things. But it is not impossible for them to be identical.

Extending this process, one can derive from | |' a measure | |": the smallest value that the quantity

$$a|y|' + b|z|' + cC(y,z) \qquad (4.1)$$

*takes on, for any (y,z) such that y*z=x. |x|" measures the complexity of the simplest representation of x, where complexity is measured by | |'. And from | |", one may obtain a measure | |"' . It is clear that this process may be continued indefinitely.*

It is interesting to ask when | | and | |' are equivalent, or almost equivalent. For instance, assume that y is a Turing machine, and x and z are binary sequences. If, in the notation given above, we let | |=| |$_T$, then |x|' is a natural measure of the complexity of a sequence x. In fact, if a=b=1 and c=0, it is exactly the KCS complexity of x. Without specifying a, b and c, let us nonetheless use Chaitin's [4] notation for this complexity: I(x).

Also, let us adopt Chaitin's notation I(v|w) for the complexity of v relative to w.

DEFINITION 11. Let y be a Turing machine program, v and w binary sequences; then I(v|w) denotes the smallest value the quantity $a|y|_T+cC_T$ (y,w) takes on for any self-delimiting program y that computes v when its input consists of a minimal-length program for computing w.

Intuitively, this measures how hard it is to compute v given complete knowledge of w. Finally, it should be noted that | | and | |' are not always substantially different:

THEOREM 1. If |x|'=I(x), a=b=1, and c=0, then there is some K so that for all x, | |x|' - |x|"| < K.

*PROOF: a|y|' + b|z|' + cC(y,z) = |y|' + |z|'. So, what is the smallest value that |y|' + |z|' assumes for any (y,z) such that y*z=x? Clearly, this smallest value must be either equal to |x|'. For, what if |y|' + |z|' is bigger than |x|'? Then it cannot be the smallest |y|' + |z|', because if one took z to be the "empty sequence" (the sequence consisting of no characters) and then took y to be the shortest program for computing x, one would have |z|'=0 and |y|'=|x|'. And, on the other hand, is it possible for |y|'+|z|' to be smaller than |x|'? If |y|'+|z|' were smaller than x, then one could supply a Turing machine with a program saying "Plug the sequence z into the program y," and the length of this program would be greater than |x|' by no more than the length of the program P(y,z)="Plug the sequence z into the program y". This length is the constant K in the theorem.*

COROLLARY 1. For a Turing machine for which the program P(y,z) mentioned in the proof is a "hardware function" which takes only one unit of length to program, | |"=| |'.

PROOF: Both | |' and | |" are integer valued, and by the theorem, for any x, |x|' ≤ |x|" ≤ |x|'+1.

5. PATTERNS IN PATTERNS; SUBSTITUTION MACHINES

We have discussed pattern in sequences, and patterns in pictures. It is also quite possible to analyze patterns in other patterns. This is interesting for many reasons, one being that when dealing with machines more restricted than Turing machines, it may often be the case that the only way to express an intuitively simple phenomenon is as a pattern in another pattern.

Let us consider a simple example. Suppose that we are dealing not with Turing machines, but rather with "substitution machines" -- machines which are capable of running only programs of the form P(A,B,C)="Wherever sequence B occurs in sequence C, replace it with sequence A". Instead of writing P(A,B,C) each time, we shall denote such a program with the symbol (A,B,C). For instance, (1,10001,1000110001100011000110001) = 11111. (A,B,C) should be read "substitute A for B in C".

We may define the complexity |x| of a sequence x as the length of the sequence, i.e. $|x| = |x|_r$, and the complexity |y| of a substitution program y as the number of symbols required to express y in the form (A,B,C). Then, |1000110001100011000110001|=25, |11111| = 5 and |(10001,1,z)|=11. If z=11111, (10001,1,z)= 1000110001100011000110001.

For example, is ((10001,1,z), 11111) a pattern in 1000110001100011000110001? What is required is that a(11) + b(5) + cC((10001,1,z),11111) < 25. If we take a=b=1 and c=0 (thus ignoring time complexity), this reduces to 11 + 5 < 25, so it is indeed a pattern.

If we take c=1 instead of c=0, and leave a and b equal to one, then this will still be a pattern, as long as the computational complexity of obtaining 1000110001100011000110001 from (10001,1,11111) does not exceed 9. It would seem most intuitive to assume that this computational complexity C((10001,1,z),11111) is equal to 5, since there are 5 ones into which 10001 must be substituted, and there is no effort involved in locating these 1's. In that case the fundamental inequality reads
11 + 5 + 5 < 25, which verifies that a pattern is indeed present.

Now, let us look at the sequence x = 1001001001001001000111001 1001001001001001001011101110100100100100100100111011110010010010010010010.
Remember, we are not dealing with general Turing machines, we are only dealing with substitution machines, and anything which cannot be represented in the form (A,B,C), in the notation given above, is not a substitution machine.

There are two obvious ways to compute this sequence x on a substitution machine. First of all, one can let y=(100100100100100100,B,z), and z= B 0111001 B 1011101110 B 110111 B. This amounts to recognizing that 100100100100100100 is repeated in x. Alternatively, one can let y'=(100,B,z'), and let z'= BBBBBB 0111001 BBBBBB 1011101110 BBBBBB 110111 BBBBBB. This amounts to recognizing that 100 is a pattern in x. Let us assume that a=b=1, and c=0. Then in the first case |y| + |z| = 24 + 27 = 51; and in the second case |y'| + |z'| = 9 + 47 = 56. Since |x| = 95, both (y,z) and (y',z') are patterns in x.

The problem is that, since we are only using substitution machines, there is no way to combine the two patterns. One may say that 100100100100100100 a pattern in x, that 100 is a pattern in x, that 100 is a pattern in 100100100100100100. But, using only substitution machines, there is no way to say that the simplest way to look at x is as "a form involving repetition of 100100100100100100, which is itself a repetition of 100".

Let us first consider |x|'. It is not hard to see that, of all (y,z) such that y is a substitution machine and z is a sequence, the minimum of |y| + |z| is obtained when

y=(100100100100100100,B,z), and z= B 0111001 B 1011101110 B 110111 B. Thus, assuming as we have that a=b=1 and c=0, |x|'=51. This is much less than |x|, which equals 95.

*Now, let us consider this optimal y. It contains the sequence 100100100100100100. If we ignore the fact that y denotes a substitution machine, and simply consider the sequence of characters "(100100100100100100,B,z)", we can search for patterns in this sequence, just as we would in any other sequence. For instance, if we let y_1=(100,C,z_1), and z_1=CCCCCC, then y_1*z_1=y, $|y_1|$=10, and $|z_1|$=6. It is apparent that (y_1, z_1) is a pattern in y, since $|y_1|$ + $|z_1|$ = 10 + 6 = 16, whereas |y| = 18. By recognizing the pattern (y,z) in x, and then recognizing the pattern (y_1, z_1) in y, one may express both the repetition of 100100100100100100 in x and the repetition of 100 in 100100100100100100 as patterns in x, using only substitution machines.*

*Is (y_1, z_1) a pattern in x? Strictly speaking, it is not. But we might call it a **second-level** pattern in x. It is a pattern in a pattern in x. And, if there were a pattern (y_2, z_2) in the sequences of symbols representing y_1 or z_1, we could call that a **third-level** pattern in x, etc. In general, we may make the following definition:*

DEFINITION 12. Let F be a map from S^2 into S. Where a first-level pattern in x is simply a pattern in x, and n is an integer greater than one, we shall say that P is an n'th-level pattern in x if there is some Q so that P is an (n-1) 'th-level pattern in x and P is a pattern in F(Q).

In the examples we have given, the map F has been, implicity, the map from substitution machines into their expression in (A,B,C) notation.

6. APPROXIMATE PATTERN

*Suppose that y_1*z_1=x, whereas y_2*z_2 does not equal x, but is still very close to x. Say |x|=1000. Then, even if $|y_1|$+$|z_1|$=900 and $|y_2|$+$|z_2|$=10, (y_2, z_2) is not a pattern in x, but (y_1, z_1) is. This is not a flaw in the definition of pattern -- after all, computing something near x is not the same as computing x. Indeed, it might seem that if (y_2, z_2) were really so close to computing x, it could be modified into a pattern in x without sacrificing much simplicity. However, the extent to which this is the case is unknown. In order to incorporate pairs like (y_2, z_2), we shall introduce the notion of approximate pattern.*

*In order to deal with approximate pattern, we must assume that it is meaningful to talk about the **distance** d(x,y) between two elements of S. Let (y,z) be any ordered pair for which y*z is defined. Then we have*

*DEFINITION 13. The ordered pair (y,z) is an **approximate pattern** in x if [1 + d(x,y*z)][a|y| + b|z| + cC(y,z)] < |x|, where a, b, c and C are defined as in the ordinary definition of pattern.*

*Obviously, when x=y*z, the distance d(x,y*z) between x and y*z is equal to zero, and the definition of approximate pattern reduces to the normal definition. And the larger d(x,y*z) gets, the smaller a|y|+b|z|+cC(y,z) must be in order for (y,z) to qualify as a pattern in x.*

Of course, if the distance measure d is defined so that d(a,b) is infinite whenever a and b are not the same, then an approximate pattern is an exact pattern. This means that when one speaks of "approximate pattern", one is also speaking of ordinary, exact pattern.

Most concepts involving ordinary or "strict" pattern may be generalized to the case of approximate pattern. For instance, we have:

DEFINITION 14: The intensity of an approximate pattern (y,z) in x is $IN[(y,z)|x] = (|x|-[1+d(x,y*z)][a|y|+b|z|+cC(y,z)])/|x|$.

DEFINITION 15: Where v and w are binary sequences, the approximate complexity of v relative to w, $I_A(v,w)$, *is the smallest value that* $[1+d(v,y*w)][a|y|+cC(y,w)]$ *takes on for any program y with input consisting of a minimal program for w.*

The incorporation of inexactitude permits the definition of pattern to encompass all sorts of interesting practical problems. For example, suppose x is a curve in the plane or some other space, z is a set of points in that space, and y is some interpolation formula which assigns to each set of points a curve passing through those points. Then $I_A[(y,z)|x]$ *is an indicator of how much use it is to approximate the curve x by applying the interpolation formula y to the set of points z.*

7. SOPHISTICATION AND CRUDITY

As indicated above, Koppel [8] has recently proposed an alternative to the KCS complexity measure. According to Koppel's measure, the sequences which are most complex are not the structureless ones. Neither, of course, are they the ones with very simple structures, like 00000000000.... Rather, the more complex sequences are the ones with more "sophisticated" structures.

The basic idea [10] is that a sequence with a sophisticated structure is part of a natural class of sequences, all of which are computed by the same program. The program produces different sequences depending on the data it is given, but these sequences all possess the same underlying structure. Essentially, the program represents the structured part of the sequence, and the data the random part. Therefore, the "sophistication" of a sequence x should be defined as the size of the program defining the "natural class" containing x.

But how is this "natural" program to be found? As above, where y is a program and z is a binary sequence, let $|y|$ *and* $|z|$ *denote the length of y and z respectively. Koppel proposes the following:*

ALGORITHM 1:

1) *search over all pairs of binary sequences (y,z) for which the two-tape Turing machine with program y and data z computes x, and find those pairs for which* $|y| + |z|$ *is smallest.*
2) *search over all pairs found in Step 1, and find the one for which* $|y|$ *is biggest. This value of* $|z|$ *is the "sophistication" of x.*

All the pairs found in Step 1 are "best" representations of x. Step 2 searches all the "best" representations of x, and find the one with the most program (as opposed to data).

This program is assumed to be the natural structure of x, and its length is therefore taken as a measure of the sophistication of the structure of x.

There is no doubt that the decomposition of a sequence into a structured part and a random part is an important and useful idea. But Koppel's algorithm for achieving it is conceptually problematic. Suppose the program/data pairs (y_1, z_1) *and* (y_2, z_2) *both cause a Turing machine to output x, but whereas* $|y_1|=50$ *and* $|z_1|=300$, $|y_2|=250$ *and* $|z_2|=110$. *Since* $|y_1|+|z_1|=350$, *whereas* $|y_2|+|z_2|=360$, (y_2, z_2) *will not be selected in Step 1, which searches for those pairs (y,z) that minimize* $|y|+|z|$. *What if, in Step 2,* (y_1, z_1) *is chosen as the pair with maximum* $|y|$? *Then the sophistication of x will be set at* $|y_1|=50$. *Does it not seem that the intuitively much more sophisticated program* y_2, *which computes x* **almost** *as well as* y_1, *should count toward the sophistication of x?*

In the language of pattern, what Koppel's algorithm does is:

1) *Locate the pairs (y,z) that are the most intense patterns in x according to* $|\ |=|\ |_T$, $a=b=1$, $c=0$.
2) *Among these pairs, select the one which is the most intense pattern in x according to* $|\ |=|\ |_T$, $a=1$, $b=c=0$.

It applies two different special cases of the definition of pattern, one after the other.

How can all this be modified to accomodate examples like the pairs (y_1, z_1), (y_2, z_2) *given above? One approach is to look at some sort of combination of* $|y|+|z|$ *with* $|y|$. $|y|+|z|$ *measures the combined length of program and data, and* $|y|$ *measures the length of the program. What is desired is a small* $|y|+|z|$ *but a large* $|y|$. *This is some motivation for looking at* $(|y|+|z|)/|y|$. *The smaller* $|y|+|z|$ *gets, the smaller this quantity gets; and the bigger* $|y|$ *gets, the smaller it gets. One approach to measuring complexity, then, is to search all (y,z) such that* $x=y*z$, *and pick the one which makes* $(|y|+|z|)/|y|$ *smallest. Of course,* $(|y|+|z|)/|y| = 1 + |z|/|y|$, *so whatever makes* $(|y|+|z|)/|y|$ *smallest also makes* $|z|/|y|$ *smallest. Hence, in this context, the following is natural:*

DEFINITION 16. The crudity of a pattern (y,z) is $|z|/|y|$.

The crudity is simply the ratio of data to program. The cruder a pattern is, the greater the proportion of data to program. A very crude pattern is mostly data; and a pattern which is mostly program is not very crude. Obviously, "crudity" is intended as an intuitive opposite to "sophistication"; however, it is not exactly the opposite of "sophistication" as Koppel defined it.

This approach can also be interpreted to assign each x a "natural program" and hence a "natural class". One must simply look at the pattern (y,z) in x whose crudity is the smallest. The program y associated with this pattern is, in a sense, the most natural program for x.

8. LOGICAL DEPTH

Bennett [9], as mentioned above, has proposed a complexity measure called "logical depth", which incorporates the time factor in an interesting way. The KCS complexity of x measures only the length of the shortest program required for computing x -- it says nothing about how long this program takes to run. Is it really correct to call a sequence of length 1000 simple if it can be computed by a short program which takes a thousand

years to run? Bennett's idea is to look at the running time of the shortest program for computing a sequence x. This quantity he calls the logical depth of the sequence.

One of the motivations for this approach was a desire to capture the sense in which a biological organism is more complex than a random sequence. Indeed, it is easy to see that a sequence x with no patterns in it has the smallest logical depth of any sequence. The shortest program for computing it is "Print x", which obviously runs faster than any other program computing a sequence of the same length as x. And there is no reason to doubt the hypothesis that biological organisms have a high logical depth. But it seems to us that, in some ways, Bennett's definition is nearly as counterintuitive as the KCS approach.

Suppose there are two competing programs for computing x, program y and program y'. What if y has a length of 1000 and a running time of 10 minutes, but y' has a length of 999 and a running time of 10 years. Then if y' is the shortest program for computing x, the logical depth of x is ten years. Intuitively, this doesn't seem quite right: it is not the case that x fundamentally requires ten years to compute.

At the core of Bennett's measure is the idea that the shortest program for computing x is the most natural representation of x. Otherwise why would the running time of this particular program be a meaningful measure of the amount of time x requires to evolve naturally. But one define the "most natural representation" of a given entity in many different ways. Bennett's is only the simplest. For instance, one may study the quantity $dC(y,z) + e|z|/|y| + f(|y|+|z|)$, *where d, e and f are positive constants defined so that* $d+e+f=3$.

The motivation for this is as follows. The smaller $|z|/|y|$ *is, the less crude is the pattern (y,z). And, as indicated above, the crudity of a pattern (y,z) may be interpreted as a measure of how natural a representation it is. The smaller* $C(y,z)$ *is, the less time it takes to get x out of (y,z). And, finally, the smaller* $|y|+|z|$ *is, the more intense a pattern (y,z) is. All these facts suggest the following:*

DEFINITION 17: Let m denote the smallest value that the quantity $dC(y,z) + e|z|/|y| + f(|y|+|z|)$ *assumes for any pair (y,z) such that* $x=y*z$ *(assuming there is such a minimum value). The depth complexity of x may then be defined as the time complexity C(y,z) of the pattern (y,z) at which this minimum m is attained.*

Setting $d=e=0$ *reduces the depth complexity to the logical depth as Bennett defined it. Setting* $e=0$ *means that everything is as Bennett's definition would have it, except that cases such as the patterns* (y_1, z_1), (y_2, z_2) *described above are resolved in a more intuitive matter. Setting* $f=0$ *means that one is considering the time complexity of the most sophisticated -- least crude, most structured -- representation of x, rather than merely the shortest. And keeping all the constants nonzero ensures a balance between time, space, and sophistication.*

Admittedly, this approach is not nearly so tidy as Bennett's. Its key shortcoming is its failure to yield any particular number of crucial significance -- everything depends on various factors which may be given various weights. But there is something to be said

for considering all the relevant factors.

9. STRUCTURE AND STRUCTURAL COMPLEXITY

We have discussed several different measures of static complexity, which measure rather different things. But all these measures have one thing in common: they work by singling out the one pattern which minimizes some quantity. It is equally interesting to study the total amount of structure in an entity.

For instance, suppose x and x_1 both have KCS complexity A, but whereas x can only be computed by one program of length A, x_1 can be computed by a hundred totally different programs of length A. Does it not seem that x_1 is in some sense more complex than x, that there is more to x_1 than to x?

Let us define the structure of x as the set of all (y,z) which are approximate patterns in x (assuming the constants a, b, and c, and the metric d(v,w), have previously been fixed), and denote it P(x). Then the question is: what is a meaningful way to measure the size of P(x)? At first one might think to add up the intensities $[1+d(y^*z,x)][a|y|+b|z|+cC(y,z)]$ of all the elements in P(x). But this approach has one crucial flaw, revealed by the following example.

Say x is a sequence of 10,000 characters, and (y_1, z_1) is a pattern in x with $|z_1|=70$, $|y_1|=1000$, and $C(y_1, z_1)=2000$. Suppose that y_1 computes the first 1000 digits of x from the first 7 digits of z_1, according to a certain algorithm A. And suppose it computes the second 1000 digits of x from the next 7 digits of z_1, according to the same algorithm A. And so on for the third 1000 digits of z_2, etc. -- always using the same algorithm A.

Next, consider the pair (y_1, z_1) which computes the first 9000 digits of x in the same manner as (y_2, z_2), but computes the last 1000 digits of x by storing them in z, and printing them after the rest of its program finishes. We have $|z_2|=1063$, and surely $|y_2|$ is not much larger than $|y_1|$. Let's say $|y_2|=150$. Furthermore, $C(y_2, z_2)$ is certainly no greater than $C(y_1, z_1)$: after all, the change from (y_1, z_1) to (y_2, z_2) involved the replacement of serious computation with simple storage and printing.

The point is that both (y_1, z_1) and (y_2, z_2) are patterns in x, but in computing the total amount of structure in x, it would be foolish to count both of them. In general, the problem is that different patterns may share similar components, and it is unacceptable to count each of these components several times. In the present example the solution is easy: don't count (y_2, z_2). But one may also construct examples of very different patterns which have a significant, sophisticated component in common. Clearly, what is needed is a general method of dealing with similarities between patterns.

Recall that $I_*(v|w)$ was defined as the approximate version of the effort required to compute v from a minimal program for w, so that if v and w have nothing in common, $I_*(v,w)=I_*(v)$. And, on the other hand, if v and w have a large common component, then both $I_*(v,w)$ and $I_*(w,v)$ are very small. $I_*(v|w)$ is defined only when v and w are sequences. But we shall also need to talk about one program being similar to another. In order to do this, it suffices to assume some standard "programming language" L, which assigns to each program y a certain binary sequence L(y). The specifics of L are irrelevant, so long

as it is computable on a Turing machine, and it does not assign the same sequence to any two different programs.

The introduction of a programming language L permits us to define the complexity of a program y as $I_*(L(y))$, and to define the complexity of one program y, relative to another program y_2, as $I_*(L(y_1)|L(y_2))$. As the lengths of the programs involved increase, the differences between programming languages matter less and less. To be precise, let L and L_1 be any two programming languages, computable on Turing machines. Then it can be shown that, as $L(y_1)$ and $L(y_2)$ approach infinity, the ratios $I_*(L(y_1))/I_*(L_1(y_1))$ and $I_*(L(y_1)|L(y_2))/I_*(L_1(y_1)|L_1(y_2))$ both approach 1.

Where z is any binary sequence of length n, let D(z) be the binary sequence of length 2n obtained by replacing each 1 in z with 01, and each 0 in z with 10. Where w and z are any two binary sequences, let wz denote the sequence obtained by placing the sequence 111 at the end of D(w), and placing D(z) at the end of this composite sequence. The point is that 111 cannot occur in either D(z) or D(w), so that wz is essentially w juxtaposed with z, with 111 as a marker inbetween.

Now, we may define the complexity of a program-data pair (y,z) as $I_*(L(y)z)$, and we may define the complexity of (y,z) relative to (y_1, z_1) as $I_*(L(y)z|L(y_1)z_1)$. We may define the complexity of (y,z) relative to a set of pairs $\{(y_1, z_1),(y_2, z_2),...,(y_k, z_k)\}$ to be $I_*(L(y)z|L(y_1)z_1L(y_2)z_2...L(y_k)z_k)$. This is the tool we need to make sense of the phrase "the total amount of structure of x".

Let S be any set of program-data pairs (x,y). Then we may define the size $|S|$ of S as the result of the following process:

ALGORITHM 2:
Step 0. Make a list of all the patterns in S, and label them (y_1, z_1), (y_2, z_2), ..., (y_N, z_N).
Step 1. Let $s_1(x)=I_*(L(y_1)z_1)$
Step 2. Let $s_2(x)=s_1(x)+I_*(L(y_2)z_2)I(L(y_1)z_1)$
Step 3. Let $s_3(x)=s_2(x)+I_*(L(y_3)z_3|L(y_1)z_1L(y_2)z_2))$
Step 4. Let $s_4(x)=s_3(x)+I_*(L(y_4)z_4|L(y_1)z_1L(y_2)z_2)L(y_3)z_3))$...
Step N. Let $|S|=s_N(x)=s_{N-1}(x)+I_*(L(y_N)z_N|L(y_1)z_1L(y_2)z_2)...L(y_{N-1})z_{N-1})$

At the k'th step, only that portion of (y_k, z_k) which is independent of $\{(y_1, z_1),...,(y_{k-1}, z_{k-1})\}$ is added onto the current estimate of $|S|$. For instance, in Step 2, if (y_2, z_2) is independent of (y_1, z_1), then this step increases the initial estimate of $|S|$ by the complexity of (y_2, z_2). But if (y_2, z_2) is highly dependent on (y_1, z_1), not much will be added onto the first estimate. It is not difficult to see that this process will arrive at the same answer regardless of the order in which the (y_i, z_i) appear:

THEOREM 2: The result of Algorithm 2 is invariant under permutation of the (y_i, z_i).

Where P(x) is the set of all patterns in x, we may now define the ***structural complexity*** *of x to be the quantity $|P(x)|$. This, we suggest, is the sense of the word "complexity" that one uses when one says that a person is more complex than a tree, which is more complex than a bacterium. In a way, structural complexity measures how many insightful statements can possibly be made about something. There is much more to say about a person than about a tree, and much more to say about a tree than a*

bacterium.

REFERENCES

1. *KRONSJO, L. Algorithms: Their Complexity and Efficiency, Wiley-Interscience, New York, 1979.*
2. *BOWEN, R. Symbolic Dynamics for Hyperbolic Flows, Am. J. of Math. 95 (1973), 421-460.*
3. *KOLMOGOROV, A.N. Three Approaches to the Quantitative Definition of Information, Prob. Info. Transmission 1 (1965), 1-7.*
4. *CHAITIN, G. Information-Theoretic Computational Complexity, IEEE-TII 20 (1974), 10-15.*
5. *SOLOMONOFF, L. A Formal Theory of Induction, Parts I. and II., Info. and Control 20 (1964), 224-254.*
6. *BENNETT, C.H. The Thermodynamics of Computation -- A Review, Int. J. Theor. Phys.21 (1982), 905-940.*
7. *ZUREK, W.H. Algorithmic Information Content, Church-Turing Thesis,Physical Entropy, and Maxwell's Demon, In Complexity, Entropy and the Physics of Information, (ed. W.H. Zurek), Addison-Wesley, New York, 1990.*
8. *KOPPEL, MOSHE. Complexity, Depth and Sophistication, Complex Systems 1 (1987), 1087-1091.*
9. *CHAITIN, G. Toward a Mathematical Definition of Life, In The Maximum Entropy Formalism (ed. Levine and Tribus), MIT Press, Cambridge MA, 1978.*
10. *ATLAN, H. Measures of Biologically Meaningful Complexity, in Measures of Complexity (ed. Peliti et al) , Springer-Verlag, New York,1988.*