CHAPTER 2

Complexity Measures

STEPHEN R. TATE

OVERVIEW

In this chapter, an alternative measure of information, or string complexity, is introduced. The "Kolmogorov complexity" of a string is basically the length of the shortest program that can compute the string, which turns out to be a very fundamental and important way of defining information. We first discuss some problems with traditional information theory, as introduced in the previous chapter, and then give the basic definitions and properties of Kolmogorov complexity and discuss why this theory solves some of the problems inherent in traditional information theory. The chapter next explores the fundamental properties of this complexity measure, including the computational complexity of determining Kolmogorov complexity, showing that unfortunately the Kolmogorov complexity is uncomputable. Finally, it is shown that Kolmogorov complexity and traditional information theory agree in a very strong way in situations where both may be applied.

2.1 INTRODUCTION

A great deal of the progress in data compression owes its existence to the foundation and insights provided by Claude Shannon in his development of information theory, the basics of which were described in the previous chapter. While the very real and practical impact of Shannon's ideas is obvious, there are some fundamental questions and problems that this theory does not adequately address. The material presented in this chapter shows that, in a very strong sense, there is an ultimate answer to questions about compressibility of data. To distinguish between the two fundamentally different approaches, in this chapter classical information theory will be referred to as "Shannon Information Theory," and the approaches presented here will be referred

to as "Algorithmic Information Theory" or, by the more commonly used term, "Kolmogorov complexity."

It should be pointed out here that the motivation for Algorithmic Information Theory is a solid understanding of what information is, and this theory can be used for reasoning about information and incompressibility. However, the practical uses (meaning uses that lead to implementation by programs) of this theory for data compression are somewhat limited. In the practical arena, Shannon Information Theory wins by virtue of its ability to be implemented, despite the less-than-perfect theory of information.

We continue this chapter with an overview of the shortcomings of Shannon Information Theory, followed by an overview of Algorithmic Information Theory and how it addresses these shortcomings, then continue with a more formal treatment of the subject, and finally end with some historical notes on the development of this theory.

Some topics in this chapter assume familiarity with basic computer science concepts and terminology, such as asymptotic notation and computability issues. Of course, this chapter just barely scratches the surface of Algorithmic Information Theory, with a concentration on those issues that relate to data compression. There are an astounding number of applications which are not obvious at all initially. For example, Algorithmic Information Theory can be used to provide an estimate of how many prime numbers are in a particular range, even though at first this question seems to have little to do with compression or even information theory in a standard sense. A book by Li and Vitànyi [10] contains a thorough treatment of Algorithmic Information Theory for those who wish to pursue this area more fully. In this chapter, we use notation that mostly agrees with that in the Li and Vitànyi book, although this chapter was written with a conscious effort to use terminology that is less formal and more familiar for non-experts when possible (e.g., we use "computable" instead of "recursive").

2.1.1 An Aside on Computability

While non-computability is a well-known topic for people who have studied computer science, some results come as a surprise to those who have not studied the area. It is not surprising at all that there are functions which are so complex that it takes a long time to compute them on current computer hardware. For example, factoring the product of two large prime numbers (without knowing the primes, of course!) seems to be extremely difficult, and in fact much of the security of modern on-line commerce depends on this function being very difficult to compute. However, this is very different from saying that the function is non-computable—in fact, factoring *is* computable because there is a very simple algorithm to do so: Simply try dividing all possible divisors into the number in question. The fact that this algorithm, run on numbers used for cryptography today, would take longer than the lifetime of the universe to complete is very important for the efficiency of the procedure, but completely irrelevant for whether the function is *computable*, where only the existence of the algorithm is important.

If computers were suddenly to become 10^{30} times faster, then all of a sudden we could factor the numbers in question, and a lot of "security" would become not so secure. This is in stark contrast to the notion of non-computability, which means that no algorithm exists at all. Computers could become 10^{30} times faster, or even 10^{1000} times faster, and it would have no effect—non-computable is non-computable, regardless of time or speed or memory available. This fact is surprising for many people who seem to have an initial intuition that everything is computable, given enough time.

The fact that there are some functions that simply cannot be expressed by an algorithm is unusual, but has parallels in other areas of mathematics. For example, Gödel's incompleteness theorem says that in any logical system (defined by a basic set of axioms), there are statements

that are true, but cannot be proved true—this is in fact very closely related to non-computability results, where the algorithmic language corresponds to the logical system, and the true statements that can be defined but not proved correspond to functions which can be defined but not computed.

In reading this chapter, keep in mind that when we refer to non-computable functions, we are not talking about simply a lack of power or speed of current machines, nor are we talking about a function which we simply have not been clever enough to invent an algorithm for. We are talking about a fundamental incomputability that exists on all conceivable computers.

2.2 CONCERNS WITH SHANNON INFORMATION THEORY

Shannon Information Theory provides some remarkable insight into the information content of data sources, but fails to satisfactorily answer some natural questions. We outline a few issues with Shannon Information Theory below, all of which will be addressed in a satisfactory way by Algorithmic Information Theory.

2.2.1 Strings versus Sources

The most fundamental question in information theory is, given a string x, what is the information content of that string? Shannon Information Theory does not have the ability to address this question, as it must look at this in terms of a probabilistic data source. In particular, if we are considering only one possible string, you might say the probability of this string is 1, which leads to the result that the entropy is zero and the string contains no information. This contradicts common sense and argues for a theory that can describe the information content of individual strings.

2.2.2 Complex Non-random Sequences

The digits of π form an infinite sequence with many properties in common with truly random sequences, such as equal frequency of digits and strings of digits. By trying to apply Shannon Information Theory and forcing a probabilistic source on top of the stream of digits, the result is a source that offers no compression due to the apparent randomness of the digits. This again contradicts what we know to be true, since the digits of π are *extremely* compressible. In fact n digits can be represented in $O(\log n)$ space, since all we really need is a program that knows how to compute the digits of π and an indication of the length of the string that we need to generate.

This argument extends to a different problem which turns out to be related to the same issue. Consider the output of a good pseudo-random number generator (by "good" we mean one where the output can pass all known statistical tests for randomness—cryptographic pseudo-random number generators certainly meet this requirement). Since the output appears random, Shannon Information Theory offers no help here, but again the data stream can be very highly compressed. The pseudo-random stream can be completely described by simply giving the length of the output and the seed required to start the pseudo-random number generator.

2.2.3 Structured Random Strings

Consider a random binary data source in which each bit is uniformly and independently chosen (so each bit has probability $\frac{1}{2}$ of being 0 and probability $\frac{1}{2}$ of being 1). Shannon Information Theory tells us that data from such a source cannot be compressed on average and, in fact,

compression approaches that are based on Shannon Information Theory (such as Huffman coding or arithmetic coding) will code all data from this source using the same size code and will achieve no compression on any string. But is it accurate to say that no data from this source are compressible?

Consider two strings, "00000000000000000000" and "01010011011110011100." Most people would say the second string is "more random" and "less compressible" even though there is absolutely no justification for these statements in light of the data source, as both strings have exactly the same probability of occurrence (they both occur with probability 2^{-20}). By Shannon's definitions, both strings have the same information content, although most people would disagree with this statement.

Each of the problems outlined above has a distinct flavor, but there is one common underlying theme: An information measure should reflect the difficulty of constructing or computing the data, not some probabilistic model that we try to fit to the data. With that in mind, we consider an information measure that addresses the descriptional complexity of an algorithm for computing the data and refer to this as Algorithmic Information Theory. This approach not only satisfactorily addresses the concerns we outlined above, but also matches up quite nicely with Shannon Information Theory in situations where that theory is applicable.

To this point, we have used the term Algorithmic Information Theory to emphasize that this is, in fact, a theory of information in the same sense as Shannon's theory. However, a more common term for this theory is Kolmogorov complexity, named after the Russian mathematician who was one of the originators of the theory. As will be described in the last section of this chapter, on the history of Kolmogorov complexity, this term is not without controversy since similar ideas were independently discovered at roughly the same time by Solomonoff and by Chaitin. However, to match common usage, in the remainder of this chapter we will use the term Kolmogorov complexity, as this has emerged as the most commonly used name.

2.3 KOLMOGOROV COMPLEXITY

The basic idea of Kolmogorov complexity is to associate information content with the difficulty of describing data, whether or not we can model that data as coming from a probabilistic source as in Shannon Information Theory. The notion of "describing data" can be made more concrete by allowing any *process* which can create the data in question, and a process is simply a computational procedure or algorithm. This then is the rough definition of Kolmogorov complexity: The Kolmogorov complexity of a string x is the length of the shortest program that generates x.

There are two important points to understand from this rough definition. First, only the size of a program is mentioned, but people familiar with compression generally think in terms of two items: a program (the decompressor) and its input (the compressed data). However, this is not really such an odd restriction. Since we are considering only a single instance (i.e., one output string), the input to a decompression program is fixed and can, in fact, be included in the program itself. Second, notice that every string does have a finite program that generates it. While we delve into the question of "what is a program" below, every reasonable programming language could produce any string with a single long "print statement," for example, "Print ⟨01000111011000101010010010101⟩."

While many different types of computing machines are available, the notion of what is computable is extremely robust across all the options, from real physical machines to mathematical models (including more esoteric models such as quantum computers) and even to our understanding of how the human brain processes information. In fact, the "Church–Turing thesis" states that the intuitive notion of computability agrees exactly with what can be computed by current

formal universal models of computation, such as a Turing machine or a random access machine [6]. While the Church–Turing thesis has not been proved (and *cannot* be proved unless something more precise than an "intuitive notion" is involved), it is generally accepted to be true. So in considering a specific model of computation, such as a Turing machine, we do not restrict the power of what we can compute.

Associating the smallest description of data with a metric for the validity or information content of the data is not unique to Kolmogorov complexity. Other areas, including other branches of mathematics and philosophy, also express similar ideas. For instance, in statistics there is a theory known as the "minimum description length principle," or just "MDL," which states that if you are given a collection of theories that describe some data, the most appropriate one is the one that minimizes the sum of the sizes of the theory description and the encoding of the data under that theory. As such theories must be describable, and hence computable, this is clearly just a restatement of Kolmogorov complexity, and in fact Kolmogorov complexity has been put to great use in the area of statistics and inductive reasoning. Furthermore, these concepts correspond very nicely to the philosophical statement known as "Occam's Razor": given competing explanations for an event, the simplest one is usually right.

2.3.1 Basic Definitions

As described above, the difficulty of describing data can be identified with the difficulty of describing an algorithm that produces the data. Note that we are not concerned (at least for now) with the computational complexity of this process, but rather only with the difficulty of describing the algorithm. So how are algorithms expressed, and how can we measure the complexity of an algorithm description?

The choice of measurement is a standard one: We represent data with binary strings, and the complexity of such a string is simply its length. In this chapter, we use the notation $|x|$ to denote the length of a binary string x. While the choice of a binary coding alphabet is straightforward, the semantics of such a string are unclear—in other words, given a binary string x, what does it represent? There are many possible choices: perhaps x is a binary encoding of a Turing machine, or x is the ASCII representation of a LISP program, or x is a binary executable for some specific machine architecture and OS.

We will use the term "descriptional language" to refer to such a language,[1] where the only condition is that it be possible to actually execute programs from this language. In particular, let p be a program written in such a language, and y be an input to p, and let ϕ be the function that maps the pair $\langle p, y \rangle$ to the output when program p is run with input y. Then the function ϕ must be computable (or more precisely, ϕ must be a partial recursive function). As the language and the function that simulates it (ϕ) are really the same, these terms can be used interchangeably.

Definition 2.3.1. *The Kolmogorov complexity of a string x with respect to a descriptional language L is the shortest program in L that produces x as output. We denote this complexity by $C_L(x)$.*

An example should clearly demonstrate that selection of the descriptional language can make a very real difference. Consider the following base 10 number: 51090942171709440000. This number is exactly 21! (21 factorial), but this number requires over 65 bits to represent in binary.

[1] Note that we use the term "language" here in the sense of programming languages, which have both syntax and semantics. This should not be confused with the formal languages sense of a "language" which defines just a set of strings with no particular semantics.

Although you could write a program in C to compute this value, numbers this large cannot be directly represented, at least on current machines. So, in addition to the factorial program, you would have to write routines to do arithmetic on "big numbers." As a result, the size of this program would be substantially larger than just printing out the number. On the other hand, if our descriptional language is the Mathematica programming language, then this value can be produced by the three-character program "`21!`," since Mathematica has intrinsic support for both the factorial function and for working with large numbers. In other words, for this string $C_{Mathematica}(x)$ is much smaller than $C_C(x)$.

Although the choice of representation does indeed make a difference, as just shown, the difference turns out not to be terribly great for an important and common class of languages called "universal languages." In fact, by definition, any universal language (sometimes called a "Turing complete language") can simulate any other universal language, and the size of such a simulation program is a constant that depends only on the two languages and not on the program that is being simulated. For example, we can certainly simulate full Mathematica functionality with a constant-size C program (in fact, Mathematica may very well be written in C). This leads us to what is called the "Invariance Theorem."

Theorem 2.3.1. *If f and g are two universal languages, then for all strings x, $|C_f(x) - C_g(x)| \leq c_{f,g}$, where $c_{f,g}$ is a constant that depends only on f and g.*

At times (such as in the proof of Theorem 2.4.1 later in this chapter), it is convenient to consider arbitrary languages, even if they are not universal. In such a case we can still upper-bound the power of such a language, but since we do not require that all things are computable we cannot lower-bound the power of this language. Thus, the following theorem is a slight variation of the Invariance Theorem.

Theorem 2.3.2. *If f is a universal language, and g is any language defined by a partial recursive function g, then for all strings x, $C_f(x) \leq C_g(x) + c_{f,g}$, where $c_{f,g}$ is a constant that depends only on f and g.*

Theorem 2.3.1 says that as long as the language is universal, it does not matter what it is, up to an additive constant. The example given previously (representing 21!) showed an extreme difference, but this was only because the data being represented were so small. For large strings, the additive constant difference in complexities becomes insignificant. This independence of language is what makes Kolmogorov complexity have a fundamental meaning that transcends issues of whim such as choice of language. Li and Vitànyi [10] stress this point succinctly by noting that "The remarkable usefulness and inherent rightness of the theory of Kolmogorov complexity stems from this independence of the description method."

Since the choice of language does not make a significant difference (as long as it is universal), we will remove the subscript from the complexity notation and simply refer to $C(x)$, understanding that results are significant only up to an additive constant value. There is in fact an area of study known as "concrete Kolmogorov complexity" in which a specific language is chosen (such as an encoding of a Turing machine with a specific, preferably small, number of states, symbols, and tapes), and then the actual constants can be estimated and evaluated.

2.3.2 Incompressibility

It should be clear that *any* lossless compression technique can be viewed in the light of Kolmogorov complexity, as just described, since we must have a program that can reproduce the original data. Unlike Shannon Information Theory, we do not have to refer to a sometimes artificially created

probabilistic model of the input, but rather can handle anything which can be computed by any machine, and in any language. This universality makes Kolmogorov complexity an extremely powerful theory. So, what can Kolmogorov complexity tell us about compression?

We start with a definition that gives terminology for discussing incompressible strings. Recall that because of the default program of simply printing out the desired string, for any string x we can bound $C(x) \leq |x| + c$, where the constant c simply reflects the size of the "print" program. This is an upper bound, so how do we refer to a lower bound?

Definition 2.3.2. *A string x is c-incompressible if $C(x) \geq |x| - c$.*

Intuitively, a c-incompressible string is one that cannot be generated by *any* program significantly shorter than the explicit printing program. The c-incompressible strings are then those that basically cannot be compressed, using any program at all, irrespective of the computational time allowed for the program. To examine how prevalent c-incompressible strings are, we take a brief detour through something called the "counting argument," which is really just an application of the well-known pigeonhole principle.

Theorem 2.3.3. *Let $S \subseteq \{0,1\}^{\leq b}$ be a subset of binary strings of length at most b, and let $f : S \to \{0,1\}^*$ be any function mapping from S to arbitrary binary strings. The image of S under f (sometimes called the "range" of f) contains at most $2^{b+1} - 1$ distinct values.*

Proof. Since there are no restrictions on how f is defined, the number of distinct values produced is limited only by the size of the domain S. Because there are exactly 2^k binary strings of length k, the number of binary strings of length b or less is exactly

$$\left|\{0,1\}^{\leq b}\right| = \sum_{k=0}^{b} 2^k = 2^{b+1} - 1.$$

Since S is a subset of $\{0,1\}^{\leq b}$, this upper-bounds the size of S. ∎

As a direct application of this theorem, consider a fixed lossless compression algorithm $A = (f_c, f_d)$, where f_c is the compression function and f_d is the decompression function. Consider the following question: How many strings of length n can be compressed by A with a savings of at least 1 byte (8 bits)? The decompression function f_d must be able to reconstruct each original binary string given its compressed representation, and the compressed forms of those that are compressed by at least 1 byte form a subset of $\{0,1\}^{\leq b}$ with $b = n - 8$. Theorem 2.3.3 now tells us that f_d can have at most $2^{b+1} - 1 = 2^{n-7} - 1$ possible distinct outputs, so this limits the number of files that can be compressed with 1 byte of savings. Because there are 2^n possible input files, there must be at least $2^n - 2^{n-7} + 1 > \frac{127}{128}2^n$ files that *cannot* be compressed by a byte. Without more knowledge of how A works, we cannot determine which strings can or cannot be compressed. But because there are 2^n possible inputs of length n, a randomly selected file of length n will be incompressible (by a byte or more) with probability $> \frac{127}{128} > 0.99$.

The above argument applies to a single compression system A and can be loosely summarized by saying "every compression algorithm will fail to achieve compression on a significant fraction of possible input files." But with Kolmogorov complexity, each input is allowed to have its own, custom-built decompression program, so can we say anything as strong? It turns out that the argument for Kolmogorov complexity is really the same and simply requires a slightly different perspective.

When we are computing the Kolmogorov complexity of a string, we always do so with respect to a fixed, specific descriptional language, say L. Because it must be possible to interpret any such language using any other universal language, we can consider the function f of Theorem 2.3.3

to be simply an "interpreter" for language L and apply the theorem to lower-bound the number of incompressible strings (even with custom-built decompression programs!). The precise result is stated in the following theorem.

Theorem 2.3.4. *Over $1 - 2^{-c}$ of all strings of length n are c-incompressible.*

Consider once again strings which allow compression by at least 1 byte. Using $c = 7$ in Theorem 2.3.4 shows that at least a fraction $1-2^{-7} = \frac{127}{128}$ of strings of length n is not compressible by even a single byte (note that $c = 7$ since any savings strictly less than 8 bits is not counted)! This is the same fraction we obtained in the earlier argument, but notice how much more powerful our setting has become. The earlier argument says that given a single, fixed compression system, this fraction of files is not compressible in this system. On the other hand, this latest argument says that this fraction ($\frac{127}{128}$) of files is not compressible by *any* compression program, even allowing compression programs to be tailored specifically for the data in question!

Trying an additional value of c is enlightening. With $c = 55$, the c-incompressible strings are those that cannot be compressed, by any program, with a savings of 7 or more bytes. Note that for files 70,000 bits (8750 bytes) or larger, the 55-incompressible strings are those that cannot be compressed by even 0.01%. Theorem 2.3.4 states that at least $1 - 2^{-55}$ strings are 55-incompressible, or, in other words, the probability of a randomly selected file being compressible by 7 or more bytes is less than 2^{-55}. To appreciate how small this number is, consider that Bruce Schneier quotes 2^{-55} as the probability that a person would win a major state lottery and get hit by lightning on the exact same day (Table 1.1 of [11]).

2.3.3 Prefix-free Encoding

A set of binary strings is "prefix-free" if no string in the set is a prefix of any other string in the set. Prefix-free sets, often referred to as "prefix codes," are useful because of their self-delimiting nature: An algorithm reading a binary string can determine when it reaches the end of a codeword, without having to look beyond the end of the codeword. Because of this property, codewords can be concatenated to form longer strings of codewords, which can then be easily parsed into the individual components. In Shannon Information Theory, only prefix codes are typically considered.

In Kolmogorov complexity, as defined and explained in the previous section, the string encodings are not required to be prefix-free. The limits of these strings are typically determined by some "out-of-band" information. For example, a string encoded by a program as required by Kolmogorov complexity might be delimited by placing it on a Turing machine tape surrounded by blank cells.

Unfortunately, this leads to some minor difficulties in the general theory of Kolmogorov complexity. For example, as defined previously, Kolmogorov complexity is not "subadditive." In other words, it is intuitively appealing that the complexity of a pair of strings, x and y, is no more than the sum of the complexities of the individual strings (so $C(x, y) \leq C(x) + C(y)$). However, this is unfortunately not true with the previous definitions.

To correct these problems, Kolmogorov complexity is often studied in a prefix-free form, called "Algorithmic Prefix Complexity." In this case, the language being used to define programs must ensure that the set of all programs is prefix-free. In fact, this is not so unusual. Typical binary executable formats on real machines have this property (since the length of the executable is in the header), as do some high-level programming languages (such as Pascal, where the end of the program is always marked by the string "`end.`").

We use the notation $K(x)$ to refer to the algorithmic prefix complexity of a string x. Clearly, adding the restriction that the set of possible programs be prefix-free cannot *decrease* the

complexity of a string, but how much can it increase? Fortunately, since this is a relatively minor change in the notion of complexity, the difference is not too great. Any set of programs can be converted to a self-delimited, prefix-free set in the following manner. Let p be the shortest program for computing x (so $C(x) = |p|$). First figure out how many bits are in the binary representation of $|p|$, the length of p. Then write out that many zeros, followed by $|p|$ in binary (note that it must start with a 1 bit), followed finally by p itself. This is clearly a self-delimiting description, so the set of all such descriptions is prefix-free. Furthermore, since the number of bits in $|p|$ is at most $\lceil \log_2 |p| \rceil$, we have $K(x) \leq C(x) + 2\lceil \log_2 C(x) \rceil$.

In fact, we can make this a little stronger still by repeating this self-delimiting process. In other words, to encode $|p|$ we really do not need to write it in binary, but rather need only to write the shortest self-delimiting program for generating $|p|$ which, by the above argument, is at most $C(|p|) + 2\lceil \log_2 C(|p|) \rceil = C(C(x)) + 2\lceil \log_2 C(C(x)) \rceil$, and so

$$K(x) \leq C(x) + C(C(x)) + 2\lceil \log_2 C(C(x)) \rceil.$$

This can, in fact, be repeated again to get an even tighter bound. The following theorem, attributed to Solovay [14] in [10] and given here without proof, makes the connection between the "K-complexity" and "C-complexity" quite explicit.

Theorem 2.3.5. *For any string x,*

$$K(x) = C(x) + C(C(x)) + O(C(C(C(x)))),$$

and

$$C(x) = K(x) - K(K(x)) - O(K(K(K(x)))).$$

2.3.3.1 Subtle Problems with Non-prefix Codes

Issues with prefix-free versus non-prefix-free codes can be very subtle and can cause a great deal of confusing and misleading statements. Several people have mistakenly claimed to have techniques for compression of arbitrary data by not understanding this concept fully. Typically, these claims go along the following lines: I can take all strings of length n and represent them with codes whose average length is less than n, so simply repeat this process with an appropriate block size and compression is achieved! In fact, the first part of the claim is true: If the input consists only of strings of length n, then these strings can be encoded with a variable-length code whose average length is approximately $n - 2$. However, the input set is in fact prefix-free (any set of fixed-length strings is prefix-free), and the second set is not. By encoding from a prefix-free set to a non-prefix-free set, there is additional information "hidden" in the output that corresponds roughly to the length of the variable-length output. Although this is simply a curiosity for single blocks, it turns into a problem when this process is repeated (as in the second part of the false "compress anything" claim). Without some sort of external information present, it is impossible to extract these variable-length encoded blocks of data, so the reverse transformation cannot be performed.

Note that an honest accounting for code lengths would require that fixed-size blocks be coded with a prefix-free set of codewords, even if variable length is allowed. If this one restriction is included, then no variable-length code can have average codeword length less than n (the proof uses the Kraft inequality), thus making the first part of the compress anything claim impossible.

The importance of prefix codes to Kolmogorov complexity is highlighted by a recent turn of events. Due to the regular claims of new revolutionary compression algorithms that work for arbitrary data, several "challenges" exist on the `comp.compression` Usenet newsgroup. One such challenge is set up so that a challenger can send $100 to the challenge author, who will

send a data file of any requested length to the challenger. If the challenger can produce a program and data file whose combined length is less than that of the original data and yet manages to reproduce that data, then the challenge author will send $5000 to the challenger. This challenge is motivated exactly by Kolmogorov complexity and the ideas presented in this chapter. Despite the allowance of separate program and data, which allows a little bit of "cheating" on the Kolmogorov complexity, we know by Theorem 2.3.4 that if the challenge data are generated randomly, then there is an extremely low probability that the challenger will succeed.

This challenge was originally designed to dissuade people from making "arbitrary compression" claims without clearly thinking them through, but the challenge was taken up in April 2001 with some interesting results. In a series of e-mails with the challenge author, the challenger managed to slip in a change in the rules that allowed for multiple data files rather than a single file. The "decompressor," subsequently provided by the challenger, then simply concatenated the multiple data files together with a fixed separator byte value. Since the separation of any two pieces saves a byte by not being prefix-free, adding up this savings over several hundred "pieces" results in saving several hundred bytes. This is enough to offset the size of the program to reassemble the pieces. The result is that the apparent total data size (what you get if you simply add up the sizes of all files involved) has in fact decreased. This worked entirely because a very subtle problem in the original challenge statement (allowing non-prefix-free representations) was exaggerated by repeating that problem many times over.[2]

2.4 COMPUTATIONAL ISSUES OF KOLMOGOROV COMPLEXITY

Since $C(x)$ is in a very strong sense the ultimate answer to the information content of the string x, the next natural question is "how can we measure $C(x)$?" The unfortunate answer to this question is that we cannot. There is no method or algorithm for computing $C(x)$ on any infinite set of strings that is correct over its entire domain. In fact, while upper bounds on $C(x)$ are possible in some sense, $C(x)$ cannot be lower-bounded in a reasonable fashion by any algorithm! In this section, these concepts are explained and expanded. We present a sequence of results that are increasingly powerful but require an increasing amount of knowledge of recursion theory (or computability theory).

In the proofs that follow, a specific enumeration of binary strings (which represent programs) is required. Specifically, for two strings x and y, we order $x < y$ if $|x| < |y|$ or if the lengths are the same and the binary number represented by the string x is smaller than the number corresponding to y. Clearly this enumeration can be listed out in order: Simply have a binary counter for strings of a particular length and place that inside a loop over all lengths in increasing order.

The following basic theorem says that no algorithm can correctly compute $C(x)$. The proof is very basic and does not require any knowledge of recursion theory. It is hopefully understandable to anyone who has followed the basic definitions in this chapter.

Theorem 2.4.1. *There is no algorithm that correctly computes $C(x)$ for all strings.*

Proof. Assume for the sake of contradiction that $C(x)$ is a computable function. Consider the following function: $f(m) = \min\{x | C(x) \geq m\}$, where the "min" is with respect to the binary string enumeration described above. Put into words, $f(m)$ is a function which maps m to the first string whose Kolmogorov complexity is at least m. This is well defined because for a given

[2] Note that in addition to the savings allowed by exploiting the non-prefix-free "loophole," there was in fact some information encoded in the *names* of the data files as well, since it was the names that allowed the pieces to be put back together in the proper order.

value of m, only a finite number of strings can have Kolmogorov complexity smaller than m. The important thing to notice is that if $C(x)$ is computable, then $f(m)$ can be computed as well, simply by going through an enumeration of strings x and testing $C(x)$ for each string in order.

Since f is a computable function (a "total recursive function" in recursion theory terms), we can use it as a descriptional language for Kolmogorov complexity and consider $C_f(f(m))$. Clearly, the input to f which produces $f(m)$ can simply be the integer m, which can be encoded with $\lceil \log_2 m \rceil$ bits, and hence $C_f(f(m)) \leq \lceil \log_2 m \rceil$. Furthermore, by the Invariance Theorem (Theorem 2.3.2) we know that $C(f(m)) \leq C_f(f(m)) + c$, where c is a constant that does not depend on m. Hence, $C(f(m)) \leq \lceil \log_2 m \rceil + c$. However, by the definition of $f(m)$ we know that $C(f(m)) \geq m$, which combined with the last observation implies that $m \leq \lceil \log_2 m \rceil + c$. Regardless of the specific value of c, it is always possible to find an m large enough to violate this condition, so we have reached a contradiction. Therefore, our original assumption, that $C(x)$ is computable, must be incorrect, which completes the proof of this theorem. ∎

Since $C(x)$ cannot be computed by an algorithm that is correct all the time, the next question to ask is "Can $C(x)$ be computed correctly for some infinite set of strings?" The answer to this is also "no." The precise statement of this result, as well as its proof, requires a knowledge of basic recursion theory, such as the definition of "recursively enumerable" and basic properties of recursively enumerable sets.

Theorem 2.4.2. *Let $g(x)$ be a partial recursive function with infinite domain A. Then $g(x)$ cannot be equal to $C(x)$ on its entire domain.*

Proof. Assume that $g(x) = C(x)$ over its entire domain. Since $g(x)$ is partial recursive, its domain A is recursively enumerable. Since every infinite recursively enumerable set has an infinite (total) recursive subset, let $B \subseteq A$ be an infinite recursive set. Consider the function $f(m) = \min\{x \in B | C(x) \geq m\}$. Since B is total recursive, we can test for membership in B, and for all $x \in B$ we can use the function $g(x)$ to compute $C(x)$. (Recall that by assumption $g(x) = C(x)$ for all $x \in B$.) This means that $f(m)$ is a total recursive function. Given this function $f(m)$, we can repeat the proof of the preceding theorem in order to reach a contradiction, and so our conclusion is that $g(x)$ cannot agree with $C(x)$ over its entire domain. ∎

We take a short break now from examining things that we *cannot* compute to look at something that we can.

2.4.1 Resource-Bounded Kolmogorov Complexity

In some sense, a good deal of computational complexity theory arose by bounding various resources and considering ideas from recursion theory. For example, placing a polynomial time bound on computations, the set of recursive languages naturally corresponds to the class P, and the set of recursively enumerable languages roughly correspond to NP. We get similarly interesting results by placing bounds on the algorithms described in Kolmogorov complexity. While any measurable resource can be limited, in this presentation we consider only limiting the time complexity of the computations.

Definition 2.4.1. *$C_L^t(x)$ is the time-bounded Kolmogorov complexity of x, which is the length of the shortest program p in descriptional language L such that $L(p) = x$ and $L(p)$ runs in time $\leq t$. There is no guarantee that such a program p even exists and, if there is no such program, we define $C_L^t(x) = \infty$.*

While $C(x)$ is not computable, clearly $C_L^t(x)$ is: Simply enumerate all programs from shortest to longest, simulating each one for t steps, and stop once $L(p) = x$. Since we have an upper

bound on the running time, we have a point where we can "move on" to the next possible program, which removes precisely the uncertainty that made the unbounded Kolmogorov complexity uncomputable.

It is worthwhile to stress again that we are interested in what is computable, not what is efficiently computable. The above argument shows that $C_L^t(x)$ is computable, but this argument is still not particularly useful for practical applications. The algorithm described above is a valid algorithm, but the time required would be astronomical and so could not be used in practice.

One nice consequence of time-bounded Kolmogorov complexity is that it gives us a way to compute an upper bound for the complexity of a string. Noticing that $C_L(x) = \lim_{t\to\infty} C_L^t(x)$, we have a simple way of proving the following theorem (Theorem 2.3.3 from [10]), where the function $g(x, t)$ referred to is just $C_L^t(x)$.

Theorem 2.4.3. *There exists a total recursive function $g(x, t)$, monotonically decreasing in t, such that $\lim_{t\to\infty} g(x, t) = C(x)$.*

2.4.2 Lower-Bounding Kolmogorov Complexity

In the previous section, we saw that we can in some sense upper-bound the Kolmogorov complexity of a string, and so now we turn to lower bounds. Unfortunately, lower bounds are much more difficult. In fact, there does not exist any computable, monotonically non-decreasing and unbounded function which is a lower bound for the Kolmogorov complexity at an infinite number of points.

First consider the Kolmogorov complexity itself. There are a large number of strings with a very small Kolmogorov complexity, and in fact some extremely long strings can be generated by very small programs, so it is natural to ask whether there exists an unbounded monotonically non-decreasing lower bound for the Kolmogorov complexity (note that this would not exist if, for example, there were an infinite set on which the Kolmogorov complexity is bounded by a constant). It turns out that there is such a lower bound.

Consider the function

$$m(x) = \min\{C(y) | y \geq x\}.$$

Clearly $m(x)$ is monotonically non-decreasing and lower-bounds the Kolmogorov complexity (i.e., $m(x) \leq C(x)$ for all x). In fact, $m(x)$ is the greatest monotonically non-decreasing lower bound for the Kolmogorov complexity. The following theorem answers the question from the previous paragraph.

Theorem 2.4.4. *$m(x)$ is unbounded.*

Proof. Assume for the sake of contradiction that $m(x)$ is bounded, so $m(x) \leq b$ for some bound b. Create a sequence of strings by setting $x_1 = \min\{x | C(x) \leq b\}$ and $x_i = \min\{x > x_{i-1} | C(x) \leq b\}$ for $i \geq 2$. Since $m(x)$ is bounded by b, we can always find such an x_i, and so this defines an infinite sequence of distinct strings all with $C(x_i) \leq b$. However, there are only finitely many programs of length $\leq b$, and so only finitely many strings can have Kolmogorov complexity $\leq b$. Thus we have a contradiction, and so $m(x)$ must be unbounded. ∎

So $m(x)$ is unbounded, but at what rate does it grow? It turns out that it grows *extremely* slowly—more slowly in fact than *any* computable unbounded function. This is stated more precisely in the following theorem.

Theorem 2.4.5. *There is no computable function $g(x)$ that is unbounded and lower-bounds $m(x)$.*

Proof. This theorem will be proved by contradiction, and the overall structure of the proof is similar to what was done for Theorem 2.4.1. First assume that such a function $g(x)$ exists. We define a function $f(b) = \min\{x | g(x) > b\}$. Since $g(x)$ is computable (i.e., total recursive) and unbounded, $f(b)$ is also computable: A simple algorithm for $f(b)$ simply enumerates strings in increasing order, checking $g(x)$ for each string and stopping once $g(x) > b$. Therefore, to compute $f(b)$ using recursive function f, it is necessary to supply only the value b, which takes at most $\lceil \log_2 b \rceil$ bits, and so $C_f(f(b)) \leq \lceil \log_2 b \rceil$. By the Invariance Theorem this implies that $C(f(b)) \leq \lceil \log_2 b \rceil + c$.

On the other hand, notice that since $m(x)$ is a lower bound for $C(x)$, and $g(x)$ is a lower bound for $m(x)$, we can bound $C(f(b)) \geq m(f(b)) \geq g(f(b)) > b$ (the last inequality comes from the definition of $f(b)$). Combined with the upper bound on $C(f(b))$ found in the previous paragraph, we get $b < \lceil \log_2 b \rceil + c$, which cannot be true for sufficiently large b. This contradiction proves that no such function $g(x)$ can exist, which completes the proof of the theorem. ∎

We note here that the preceding theorem can be extended to partial recursive functions without too much difficulty, so the final result for attempts at lower-bounding Kolmogorov complexity is that there exists no partial recursive function that is monotonically non-decreasing, is unbounded, and lower-bounds $C(x)$ at infinitely many points. The proof is a slight modification of the preceding proof and can be found as Theorem 2.3.1 in [10].

2.5 RELATION TO SHANNON INFORMATION THEORY

Now that the basics of Kolmogorov complexity have been explained, it is natural to ask what the relationship is between the two main theories of information: Kolmogorov complexity and Shannon Information Theory. Since these two theories were arrived at from completely different starting points, and taking two very distinct approaches, it is perhaps surprising that in the domain where they can both be applied they are in almost complete agreement. On the other hand, as they both aim to answer the same fundamental questions regarding information content, perhaps this should not be so surprising after all. Since Shannon Information Theory requires codings to be prefix-free, in this section we will use the prefix-free variant of Kolmogorov complexity, as discussed and given the notation of $K(x)$ in Section 2.3.3.

We consider strings generated by a probabilistic source $\mathcal{S}$, where the set of possible strings is $x_1, x_2, \ldots, x_m$, and the probability of string x_i being generated is p_i.[3] The entropy of this source can be written as

$$H(\mathcal{S}) = \sum_{i=1}^{m} -p_i \log_2 p_i.$$

This is the basically the expected information content (in the Shannon sense) of strings generated by this source, although note that it does not depend in any way on the strings themselves (only the set of probabilities). We can also consider the expected information content, as measured

[3] Note that this is slightly different from the traditional definition of a source, because of the inclusion of the "strings." The strings are not needed, and are in fact irrelevant, for most of Shannon Information Theory, but are vitally important to Kolmogorov complexity. Therefore, to relate these two measures we must include strings in the definition of a source!

by Kolmogorov complexity, of strings from this source. We define the expected Kolmogorov complexity of a source to be

$$\overline{K}(\mathcal{S}) = \sum_{i=1}^{m} p_i K(x_i).$$

Note that in this measure, the strings themselves (which were ignored in the entropy) are giving the main information content, whereas the probabilities are used only for taking the expected value.

Ideally, we would like to say that $H(\mathcal{S})$ is equal to $\overline{K}(\mathcal{S})$, but unfortunately it is not quite this easy to get a meaningful result. What can be proved without too much difficulty is that these two values are equal up to an additive constant, where the constant depends only on the probability distribution of the source. However, this is a pretty content-free result: Since $H(\mathcal{S})$ itself depends only on the probability distribution, the difference of $\overline{K}(\mathcal{S})$ and $H(\mathcal{S})$ from this argument tells us very little about the actual relationship between these measures. In the following two subsections, two approaches are considered to make the correspondence more meaningful: In the first, we consider a series of sources with increasing entropy, so that functions of the entropy stand out clearly from constants that do not grow with the entropy. In the second approach, we introduce and use conditional complexity measures to reduce the difference between the measures to a small constant that is independent of the probability distribution, showing that these two measures really *are* the same.

2.5.1 Approach 1: An Infinite Sequence of Sources

In this section, let $\mathcal{S}$ be a source as normally considered in Shannon Information Theory: In particular, $\mathcal{S}$ is a stationary, ergodic process (such as an ergodic Markov chain). For the results of this section, we also require that this process have a recursive probability distribution (a recursive probability distribution is simply one in which the probability of any possible output string is computable). Note that models commonly used in Shannon Information Theory fit easily into this setting. For example, if the source is an *iid* (independent, identically distributed) source, then a list of the probabilities of the individual output alphabet symbols describe the source, and string probabilities are obtained by simply multiplying together the individual symbol probabilities. More advanced models, such as Markov chains, also easily work here, but with a slightly more complex method for computing string probabilities.

Let $\mathcal{S}_n$ denote the source restricted to output strings of length n, where probabilities are conditioned on this length. We use $x_{1,n}, x_{2,n}, \ldots, x_{m_n,n}$ to denote the possible outputs from this source and $p_{1,n}, p_{2,n}, \ldots, p_{m_n,n}$ to denote the probabilities. Remember that these are conditioned on the length of the output, so if A denotes the set of all length n strings, and $P(x)$ denotes the probability of a string x in the original source $\mathcal{S}$, and $p(A) = \sum_{x \in A} P(x)$, then $p_{i,n} = (P(x_{i,n}))/(P(A))$.

Theorem 2.5.1. *With the above definitions,* $0 \leq \overline{K}(\mathcal{S}_n) - H(\mathcal{S}_n) \leq K(n) + c$, *where* c *is a constant that depends only on* $\mathcal{S}$ *(and in particular, not on* n*).*

Proof. Lower-bounding $\overline{K}(\mathcal{S}_n)$ turns out to be easy. While Kolmogorov complexity gives much lower complexities for some strings than would be expected from Shannon Information Theory, this cannot allow Kolmogorov complexity (at least in the prefix-free variant) to beat the entropy bound on average! In particular, the minimum-length programs considered in algorithmic prefix complexity form a prefix-free set of codewords, and Shannon's noiseless source-coding theorem states that no uniquely decipherable code (such as a prefix-free code) can beat the entropy on average. Therefore, $\overline{K}(\mathcal{S}_n) \geq H(\mathcal{S}_n)$.

For the upper bound, we can construct a program for all strings generated by $\mathcal{S}_n$ as follows: The program simply takes n as its first input, enumerates all strings of length n, and computes the probabilities of all these strings (recall that we required the probability distribution to be recursive, so this is possible). From these probabilities, the program then constructs a Huffman code that has average code length within 1 bit of entropy [7]. Note that this program does not depend on the actual value of n, so the program has constant size for all sources $\mathcal{S}_n$. Let c' denote the size of this program. Following this program, we encode its input data: a prefix-free encoding of the length n, followed by the Huffman code for the particular string of length n we are encoding. Let $\alpha(x_{i,n})$ be the Huffman code assigned to string $x_{i,n}$ in this manner, so the total size of the program and its input data is $c' + K(n) + |\alpha(x_{i,n})|$. This specific program/input coding strategy upper-bounds the minimum-length program, so

$$\overline{K}(\mathcal{S}_n) \leq \sum_{i=1}^{m_n} p_{i,n}(c' + K(n) + |\alpha(x_{i,n})|) = c' + K(n) + \sum_{i=1}^{m_n} p_{i,n}|\alpha(x_{i,n})| \leq c' + K(n) + H(\mathcal{S}_n) + 1,$$

where the last bound follows from the use of Huffman codes. With $c = c' + 1$, we get exactly the upper bound claimed in the theorem. ∎

The upper bound in the preceding theorem is $K(n) + c$, but notice that there is a simple prefix-free encoding of integers such that $K(n) = O(\log n)$. Since $\mathcal{S}$ is a stationary ergodic process, and any such source has a finite entropy rate (see Theorem 4.2.1 in [5]), we know that $H(\mathcal{S}_n)$ grows linearly with n. In particular, if h is the entropy rate of the source $\mathcal{S}$, then $H(\mathcal{S}_n) \geq (h - \epsilon) \cdot n$ for any $\epsilon > 0$ and sufficiently large n, and so using the upper bound from the theorem above we see that

$$\lim_{n \to \infty} \frac{K(n) + c}{H(\mathcal{S}_n)} = 0.$$

The obvious consequence is that, in the limit, the two notions of information (Shannon entropy and expected Kolmogorov complexity) agree! We make this explicit in the following corollary.

Corollary 2.5.1. *Given the above definitions,*

$$\lim_{n \to \infty} \frac{\overline{K}(\mathcal{S}_n)}{H(\mathcal{S}_n)} = 1.$$

2.5.2 Approach 2: Conditional Complexities

Recall the problem we are trying to solve in comparing Kolmogorov complexity to Shannon entropy: If we allow for a constant difference that depends on the probability distribution, how can we be certain that we are not hiding something arbitrarily worse than the entropy itself? In our second approach we solve this by essentially giving the probability distribution "for free," ensuring that we are examining the information content of the strings generated by the source and not the source itself.

Definition 2.5.1. *The conditional (prefix-free) Kolmogorov complexity of a string x, conditioned on another string y, is denoted $K(x|y)$ and is the minimum-length program that produces x when string y is supplied as an auxiliary input for free.*

Being a little more informal, we can condition on information not expressed as a string, as long as it is clear that this information *can* be encoded in such a fashion. In other words, in the case of interest in this section, we can talk about $K(x|\mathcal{S})$, or the algorithmic prefix complexity of

string x when some description of the source $\mathcal{S}$ (complete with all probabilities) is given for free. To be formal, $\mathcal{S}$ would have to be encoded as a specific string, but we leave such technicalities out of our light coverage in this chapter. We can also define the expected conditional Kolmogorov complexity as

$$\overline{K}(\mathcal{S}|y) = \sum_{i=1}^{m} p_i K(x_i|y),$$

with $\overline{K}(\mathcal{S}|\mathcal{S})$ following from this definition.

Theorem 2.5.2. *Given the above definitions,*

$$0 \leq \overline{K}(\mathcal{S}|\mathcal{S}) - H(\mathcal{S}) \leq c,$$

where c is a constant that depends only on the descriptional language used for the complexity measure (and in particular, c is independent of the probability distribution of $\mathcal{S}$).

Proof. The proof of this theorem is in fact very similar to the proof of Theorem 2.5.1, and so we just roughly outline it here. First note that even though we have supplied additional information (on the source) to the calculation of $\overline{K}(\mathcal{S}|\mathcal{S})$, this is still a prefix code and so Shannon's noiseless source-coding theorem gives the lower bound on $\overline{K}(\mathcal{S}|\mathcal{S}) - H(\mathcal{S})$.

For the upper bound, notice that if the "for free" information is a description of the source as an encoding of all (x_i, p_i) pairs, then a program can use this information to create a Huffman code and can decode these codes into the x_i strings. The size of this program is just a constant amount for the program code, plus the space required for the Huffman code for the desired string. Taking the expected value, and the bound for the efficiency of Huffman codes, we get the upper bound stated in the theorem. ∎

2.5.3 Discussion

The results given in this section, showing the relationship between Kolmogorov complexity and Shannon Information Theory, are roughly what one would hope for from two measures of the same quantity, in this case information. The difference between the two measures, when examined in two different ways, is at most an additive constant, which is the best that can be hoped for. However, since this relationship is desirable it is tempting to not delve deeper and to miss the strength of these results.

In fact, what these results say is almost miraculous. Consider that we have two parts to our source, the strings (the x_i's) and the probabilities (the p_i's), and there is no dependence at all between these two parts. It is even possible to take a source and rearrange the strings so that the strings correspond to entirely different probabilities. Furthermore, our two information measures deal primarily with these two different parts: Kolmogorov complexity is determined primarily from the actual x_i strings, and Shannon's entropy does not even consider the strings! Despite the fact that these two measures look at two different and independent values, they still manage to agree in a very strong way. This goes far beyond "coincidence" and provides additional evidence that we are indeed defining information in a proper and consistent manner.

The existence of a universal information measure, for the most part independent of the probability distribution, is somewhat related to a line of reasoning by Ray Solomonoff, the first inventor of Kolmogorov complexity. In particular, in the area of inductive inference, applying Bayes' Rule one must in some manner decide upon a prior distribution, and Solomonoff demonstrated that, using the ideas of Kolmogorov complexity (although it was not known by this name at the time), there exists a *universal* prior distribution that works as well as the true prior as long as the true

prior is a recursive distribution. This idea of a universal probability distribution is an extremely powerful result of this theory, but is beyond the scope of this chapter—for more information, see the references given in the following section.

2.6 HISTORICAL NOTES

The history of Kolmogorov complexity, or Algorithmic Information Theory, is quite interesting. The same core principles were discovered independently by at least three different people within the span of a few years. The convergence of results and ideas, in areas ranging from computability to probability theory, made the emergence of Kolmogorov complexity inevitable, but the three inventors deserve a great deal of credit for putting the appropriate pieces together.

Historically, the first person to put forward ideas along these lines was Ray Solomonoff in the early 1960s [13]. Solomonoff was studying inductive inference and problems with Bayesian priors in particular. Using the still-new ideas in computability, he developed a theory of "algorithmic probability" to use in his theory of inductive reasoning. While the ideas Solomonoff proposed are essentially the same as what we see in Kolmogorov complexity today, the setting was a bit different (and more specialized) and the results were not widely known at the time.

In the mid-1960s, Kolmogorov complexity as we know it was discovered independently and almost simultaneously by Andrei Kolmogorov and Gregory Chaitin. The setting for Kolmogorov and Chaitin was more similar to the context in which we study Kolmogorov complexity today, with a focus on information and the fundamental question of "what is random data?" Kolmogorov was a very-well-known Russian mathematician with a long and distinguished career, which explains to some extent why his name is associated with the theory. His basic paper [8], while brief, outlines the basic ideas and proves the Invariance Theorem.

Chaitin was the last in the time sequence of inventors, but his story is nonetheless fascinating. Chief among the interesting facts is that Chaitin was only 18 years old at the time of his discoveries, which he submitted as a pair of papers to the prestigious *Journal of the ACM* [1, 2]. Chaitin has continued his career with, among other significant work, a series of book publications exploring the nature of information (and in particular algorithmic information theory), randomness, and incompleteness.

2.7 FURTHER READING

There is a great deal published on Kolmogorov complexity, but the best place to start for more information would be either Li and Vitányi's book [10] or, for a different take on the issues (albeit with a less broad-ranging coverage), any of Chaitin's books in this area, such as [3, 4]. The application of these ideas goes far beyond data compression issues, as the notion of "information" is fundamental to many different areas from traditional mathematics, to computer science, to physics and other areas.

2.8 REFERENCES

1. Chaitin, G. J., 1966. On the length of programs for computing finite binary sequences. *Journal of the ACM*, Vol. 13, No. 4, pp. 547–569.
2. Chaitin, G. J., 1969. On the length of programs for computing finite binary sequences: Statistical considerations. *Journal of the ACM*, Vol. 16, No. 1, pp. 145–159.

3. Chaitin, G. J., 1988. *Algorithmic Information Theory*. Cambridge Univ. Press, Cambridge, UK.
4. Chaitin, G. J., 1992. *Information-Theoretic Incompleteness*. World Scientific, Singapore.
5. Cover, T. M., and J. A. Thomas, 1991. *Elements of Information Theory*. Wiley, New York.
6. Hopcroft, J. E., and J. D. Ullman, 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.
7. Huffman, D. A., 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Electrical and Radio Engineers*, Vol. 40, No. 9, pp. 1098–1101.
8. Kolmogorov, A. N., 1965. Three approaches to the quantitative definition of information. *Problems of Information Transmission [Translated from Problemy Peredachi Informatsii (Russian)]* Vol. 1, pp. 1–7.
9. Li, M., and P. Vitányi, 1990. Kolmogorov complexity and its applications. In *Handbook of Theoretical Computer Science* (J. V. Leeuwen, Ed.), pp. 187–254, Elsevier Science, Amsterdam.
10. Li, M., and P. Vitányi, 1997. *An Introduction to Kolmogorov Complexity and Its Applications*, 2nd ed. Springer-Verlag, New York.
11. Schneier, B., 1996. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd ed. Wiley, New York.
12. Shannon, C., 1948. The mathematical theory of communication. *Bell System Technical Journal*, Vol. 27, pp. 379–423 and 623–565.
13. Solomonoff, R., 1964. A formal theory of inductive inference (I and II). *Information and Control*, Vol. 7, pp. 1–22 and 224–254.
14. Solovay, R. M., 1975. Lecture Notes on Algorithmic Complexity. Unpublished, University of California at Los Angeles.
15. Zvonkin, A., and L. Levin, 1970. The complexity of finite objects and the development of the concepts of information and randomness by means of the theory of algorithms. *Russian Mathematical Surveys*, Vol. 25, pp. 83–124.