Compression Depth and the Behavior of Cellular Automata

James I. Lathrop
Iowa State University, jil@iastate.edu

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports
Part of the Theory and Algorithms Commons

Recommended Citation
http://lib.dr.iastate.edu/cs_techreports/62

This Article is brought to you for free and open access by the Computer Science at Iowa State University Digital Repository. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Compression Depth and the Behavior of Cellular Automata

James I. Lathrop
Department of Computer Science
Iowa State University
Ames, IA 50011

Abstract

A computable complexity measure analogous to computational depth is developed using the Lempel-Ziv compression algorithm. This complexity measure, which we call compression depth, is then applied to the computational output of cellular automata. We find that compression depth captures the complexity found in Wolfram Class III cellular automata, and is in good agreement with his classification scheme. We further investigate the rule space of cellular automata using Langton's $\lambda$ parameter.

---

1This research was supported in part by National Science Foundation Grant CCR-9157382, with matching funds from Rockwell, Microwave Systems Corporation, and Amoco Foundation.
1 Introduction

Measures of the complexities of objects are widely used in both theory and applications in order to model, predict, and classify objects. Information theory gives us several methods for measuring the information content of objects. The most widely used of these information measures, entropy and algorithmic information (Kolmogorov complexity), are used to solve problems in several scientific fields, including data compression, data prediction, image processing, and computational complexity.

Even though these two measures of information content are invaluable in many areas of research, they do not capture the essence of what many perceive to be complex. The canonical example of this phenomenon is an object composed entirely of random bits. Under these widely used measures of information content, a string of random bits has maximal information content. However, it seems to lack the intricate structure, found in complex objects, that allows the information to be used efficiently.

Many researchers have defined measures of complexity that attempt to capture specific types of organization or information. Typical proposals for such a measure are specific and designed to measure an object’s complexity under a restricted model or domain. (See [8, 10, 11, 24, 22] for example.) However, Bennett [2, 3] has defined a complexity measure based on programs for universal Turing machines that does capture the desired complexity criteria and is universal for all objects that can be digitally encoded.

Under Bennett’s definition, the computational depth [2, 3] of a binary data string is roughly the amount of time required to generate the string from a description of the string of nearly minimal length. (A parameter $s$ is used to define “nearly minimal” in section 3.) A description of an object contains all the essential information required to algorithmically reproduce the object; a minimal description contains no redundancy or structure. (If a description did contain structure, this structure could be used to compress it.) If an object cannot be quickly derived from its minimal description, then the object is organized (contains redundancy) in an essential way. This organization is quantified by the amount of time required to generate it from the minimal description. Thus, computational depth is the amount of organization embedded in a string by a computation.

Computational depth appears to be an ideal complexity measure for determining whether an object contains intricate structure. For example, Bennett [2, 3] notes that objects with simple structures such as strings consisting of all zeros or strings composed of random bits are not deep. It is easy to show that strings that have
maximal information content, using entropy or Kolmogorov complexity as a measure, are not strongly deep. Bennett also shows that the characteristic sequence of the halting problem is strongly deep, reflecting its very intricate structure. Further evidence that computational depth measures structural organization is given by Juedes, Lathrop, and Lutz [12] who have shown that, if an object can be used to speed up the computations of a significant collection of recursive sequences, then that object must be strongly deep.

Unfortunately, an essential feature in the definition of computational depth is Kolmogorov complexity, an uncomputable quantity. This makes computational depth uncomputable, and although valuable in theoretical contexts, its non-computability renders it useless for actual complexity measurements.

In this paper, we introduce compression depth, a complexity measure that is motivated by Bennett's computational depth, but is feasibly computable, and thus useful for making actual measurements of the complexities of specific objects. We demonstrate this usefulness by investigating the compression depth of cellular automata in classes that have been defined and investigated by Wolfram [25] and Langton [16].

Roughly, the compression depth of a string $x$ is the amount of resource required to compress $x$ to within some number of bits of the smallest compression of $x$. (In the terminology of Bennett [2, 3], a string with high compression depth is said to be cryptic.) Unlike Bennett's notion of computational depth, the resource is not required to be time, but may be any resource whose restriction impairs the performance of a compression algorithm, thereby parameterizing the amount of compression in terms of the resource. Intuitively, a string has a large compression depth if, as more resources are allowed, the compression algorithms utilizes these resources to find more subtle redundancy and further compresses the string. In this paper, we use the well-known Lempel-Ziv (LZ) compression algorithm and restrict the size of the dictionary. By computing the compression of a string $x$ at many different resource levels (dictionary sizes), we thus compute an analogy of computational depth that may be used to measure the "organizational" complexity of $x$.

We demonstrate two applications of compression depth, using cellular automata as a testing ground. First, experiments show that Wolfram Class I and Class II cellular automata (automata that give rise to simple structures) are shallow, having low compression depth. Wolfram class III cellular automata (automata that give rise to "random" structures) are also shallow. Wolfram class IV automata produce patterns that are seen to be complex to the human eye and seem to evolve a rich structure. Our experiments show that many Class IV automata also have large compression
depth, confirming that compression depth appears to measure some type of structure or complexity found in these types of cellular automata.

Further experiments are performed on cellular automata using Langton’s $\lambda$-parameter. Langton [16] defines a method that imposes a structure and ordering on the set of transition functions for all “legal” cellular automata. This allows Langton to define a single parameter, $\lambda$, which he uses to study the behavior of cellular automata. In his experiments, Langton finds that certain ranges of $\lambda$ produce behavior consistent with each of the four classes defined by Wolfram. Even more intriguing, Langton finds a particular value of $\lambda$ around which the entropy of the dynamic system changes from one extreme to the other. Langton also observes that complex automata measured by transient length arise at or near this transition region, leading him to conclude that complex Wolfram Class IV also occur in this region.

We sample cellular automata at various values of $\lambda$ to determine whether the phase transition corresponds to Wolfram Class IV automata as measured by compression depth. Our results differ from Langton’s and show that complex behavior arises in a range of values for $\lambda$. Further inspection shows that the complex automata tend to occupy the entropy phase gap noted by Langton. Our experiments show that automata in this region are rare, corresponding to theoretical results showing that strongly deep objects are also rare.

This paper is divided into five main sections. Section 2 defines basic notations and definitions used throughout the paper, including a basic treatment of Kolmogorov complexity. Section 3 defines the computational depth of finite strings and then describes a method by which compression algorithms can be used to give a depth-like complexity measure. We then describe how to use the Lempel-Ziv compression algorithm to compute the compression depth of strings. Section 4 describes results obtained by applying the compression depth algorithm to cellular automata classified by Wolfram’s scheme. Section 5 describes results pertaining to Langton’s work and his $\lambda$ parameter. Finally, Section 6 presents conclusions drawn from the results in this paper and gives some directions for future research.
2 Preliminaries

This section defines common notations and definitions used throughout this paper. Other notations and definitions used in this paper are defined where they first appear.

A string, usually represented by lower case characters, is a finite sequence of symbols from the set \( \{0, 1\} \). The set of all strings over \( \{0, 1\} \) is denoted by \( \{0, 1\}^* \). For a string \( x \in \{0, 1\}^* \), the length of \( x \) is denoted by \(|x|\). The empty string, \( \lambda \), has length 0.

For the strings \( x, y \in \{0, 1\}^* \), the string \( x \cdot y \) denotes the concatenation of the string \( x \) and the string \( y \) and \( x^n \) denotes the \( n \)-fold concatenation of \( x \) with itself. The substring \( x[i..j] \) denotes a string consisting of the \( i \)th through \( j \)th bits of the string \( x \), where \( 0 \leq i \leq j \leq |x| - 1 \).

We let < be the standard (total) ordering of binary strings, first by length and then lexicographic. Thus \( \lambda < 0 < 1 < 00 < 01 < \cdots \). The string \( s_a \) is the \( a \)th string in the lexicographic ordering of all strings \( x \in \{0, 1\}^* \). For example, \( s_0 = \lambda, s_1 = 0, s_2 = 1, s_3 = 00, s_4 = 01 \), etc.

A string \( x \) is a prefix of a string \( y \), denoted by \( x \sqsubseteq y \), if and only if \(|x| \leq |y|\) and \( x = y[0..|x| - 1] \). A string \( x \) is a proper prefix of \( y \), denoted by \( x \sqsubseteq y \), if and only if \( x \sqsubseteq y \) and \(|x| < |y|\).

Kolmogorov complexity, also called program-size complexity, was discovered independently by Solomonoff [23], Kolmogorov [13], and Chaitin [4]. Self-delimiting Kolmogorov complexity is a technical improvement of the original formulation that was developed independently, in slightly different forms, by Levin [18, 19], Schnorr [20], and Chaitin [5]. The advantage of the self-delimiting version is that it gives precise characterizations of algorithmic probability and randomness. In this paper, in order to simplify the presentation of compression depth, we very briefly develop the elements of Kolmogorov complexity and algorithmic information.

It is well-known that there are Turing machines \( U \) that are universal in the sense that, for every Turing machine \( M \), there exists a program \( \pi_M \in \{0, 1\}^* \) such that, for all \( \pi \in \{0, 1\}^* \),

\[
U(\pi_M \pi) = M(\pi).
\]

(This condition means that \( M(\pi) \downarrow \) if and only if \( U(\pi_M \pi) \downarrow \), in which case \( U(\pi_M \pi) = M(\pi) \).) Furthermore, there are universal Turing machines \( U \) that are efficient, in the sense that, for each Turing machine \( M \) there is a constant \( c \in \mathbb{N} \) (which depends on
such that, for all \( \pi \in \{0, 1\}^* \),

\[
time_U(\pi M \pi) \leq c(1 + \time_M(\pi) \log \time_M(\pi)).
\]

Fixing a universal Turing machine \( U \), the Kolmogorov complexity of a string \( x \) is

\[
K(x) = \min \left\{ |\pi| \mid U(\pi) = x \right\}.
\]

(Here we use the convention that \( \min \emptyset = \infty \).

The quantity \( K(x) \) is also called the \textit{algorithmic entropy}, or \textit{algorithmic information content}, of \( x \).

Kolmogorov complexity may be generalized by bounding the amount of time that may be used to generate the string \( x \). The \textit{\( t \)-time-bounded Kolmogorov complexity} of \( x \) is defined

\[
K^t(x) = \min \left\{ |\pi| \mid U(\pi) = x \text{ in time } t \right\}.
\]

Note that as \( t \to \infty \), \( K^t(x) = K(x) \).

Using Kolmogorov complexity, it is possible to define a notion of randomness. Intuitively, if a string \( x \) has small Kolmogorov complexity, i.e., \( K(x) \) is much less than \( |x| \), then there is a short program \( \pi \) such that \( |\pi| = K(x) \) and \( U(\pi) \) outputs the string \( x \). Thus \( x \) must contain some redundancy, or pattern, that is exploited by the program \( \pi \) to generate \( x \). Since a random string contains no such pattern, it must have a Kolmogorov complexity that is essentially as large as its length.
3 Compression Depth

Compression depth is a computable complexity measure that attempts to measure the amount of structure (organization) in (finite) binary strings. Motivated by Bennett’s notion of computational depth [2, 3], compression depth is based on well-known compression algorithms that quickly compress data. While any lossless compression algorithm with the property that it can be parameterized may be used to define a compression depth complexity measure, this section focuses on the well-understood Lempel-Ziv compression algorithm, and thereby defines the LZ-compression depth of strings.

3.1 Computational Depth

Computational depth and compression depth both attempt to measure the organization, and therefore usefulness, of a finite binary string. Because compression depth is motivated by computational depth, we give a brief description of computational depth here. The interested reader may read the papers by Bennett [2, 3] or Juedes, Lathrop, and Lutz [12] for more in-depth and detailed analyses of computational depth and its properties.

Roughly speaking, the computational depth (called “logical depth” by Bennett [2, 3]) of an object is the amount of time required for an algorithm to derive the object from its shortest description. (Precise definitions appear below.) Since this shortest description contains all the information in the object, the depth thus represents the amount of “computational work” that has been “added” to this information and “stored in the organization” of the object. (Depth is closely related to Adleman’s notion of “potential” [1] and Koppel’s notion of “sophistication” [14].)

Definition (Bennett [2, 3]) Let \( x \in \{0,1\}^* \) be a string, and let \( s \in \mathbb{N} \) be a significance parameter. The depth of the string \( x \) at significance level \( s \), is the number

\[
\text{depth}_s(x) = \max \left\{ t | K(x) \leq K^t(x) - s \right\},
\]

where we use the convention that \( \max \emptyset = 0 \). For any given significance level \( s \), a string \( x \) is called \( t \)-deep if \( \text{depth}_s(x) \geq t \), and \( t \)-shallow otherwise. Figure 1 shows the relationship between depth and \( K^t(x) \) for a hypothetical string \( x \).

It is easy to see that the above definition gives us a complexity measure with the property that simple and random strings are shallow. For example, consider a string
Figure 1: Graph of $K'(x)$ and its relationship to computational depth for a hypothetical string $x$.

$x$ such that $K(x) \geq |x|$. (A simple counting argument shows that, for all $n$, at least one string of length $n$ has this property. In fact, a number of researchers [15] have independently shown that, for all sufficiently large $n$, at least $2^{n-c}$ of the strings of length $n$ have this property, where $c$ is a constant that does not depend on $n$.) Since there is a very fast program of length $|x| + 2\log|x| + C$ that simply prints the string $x$, and since $K(x) \leq K'(x)$ for all $t$, the depth of $x$ cannot be any greater than the time it takes to print $x$ at any significance level greater than $2\log|x|$. On the other hand, if $x$ is simply $0^n$, then $x$ contains at most $\log|x|$ bits of information, and hence $K(x) \leq 2\log|x|$. Since there is a fast (linear time) program that contains the binary encoding of the length of $x$ that simply loops and outputs $0^n$, and since $K(x) \leq K'(x)$ for all $t$, the depth of $x$ can be no greater than the time it takes for the program to output $0^n$ at significance levels greater than $2\log|x|$.

In contrast with the two examples described above, the characteristic sequence of the halting language, denoted $\chi_H$, is an example of a sequence that has a high depth
measure. (This was proven by Bennett [2, 3] and generalized by Juedes, Lathrop, and Lutz [12].) Consider the first \( n \) bits of this sequence, namely the string \( \chi_H[0..n-1] \). This string can be recovered exactly from a program that encodes the length of the string and the number of ones contained in the string. Such a program can easily be written with length at most \( 2\log \log n + 2\log n \). Thus, \( \chi_H[0..n-1] \) contains roughly the same amount of algorithmic information as the string 0\(^n\); however, the high depth of \( \chi_H[0..n-1] \) implies that its information is “buried,” or stored more “deeply” in the string, thereby requiring much more computation time to produce it from its minimal description. In effect, the time-bounded Kolmogorov complexity of \( \chi_H[0..n-1] \) drops as \( t \) is increased, but it does not drop quickly.

The main problem with using computational depth as a complexity measure is that \( K(x) \) is not computable, and therefore not useful for computing the complexity of strings or objects. However, \( K(x) \) can be approximated by \( K^t(X) \) by using a sufficiently large value of \( t \), say \( T \). Thus, depth can be approximated by substituting \( K^T(x) \) for \( K(x) \) in the definition of \( depth_s(x) \), giving the following approximation for depth:

\[
depth^T_s(x) = \max \left\{ t | K^T(x) \leq K^t(x) - s \right\}.
\]

Here as before, we use the convention that \( \max \emptyset = 0 \).

Since \( K^t(x) \) is computable, \( depth^T_s(x) \) is computable. However, this is still not a measure that can practically be used to measure the complexity and structure of strings or the dynamics of cellular automata. The computation of \( depth^T_s(x) \) requires that all possible short programs that output \( x \) be simulated for \( T \) steps. Since the number of candidate programs is on the order of \( 2^{|x|} \), \( depth^T_s(x) \) is not feasibly computable and is thus not much more useful than \( depth_s(x) \) for the purpose of actually computing the depth.

### Compression and Depth

The definition of computational depth uses time-bounded Kolmogorov complexity to measure the time required to compute \( x \) from its smallest representation. However, \( K(x) \) is not computable, and the obvious approximation to \( K(x) \) requires so much computation time as to render it unusable. One way to proceed is to consider the “reverse” of time-bounded Kolmogorov complexity by formulating a complexity measure based on the time required to compress \( x \) to its shortest description. (This is similar to Bennett’s notion of *cryptic* [2, 3].) Using compression as the basis for a depth-like measurement gives the following approach to defining the *compression*
depth of a string \( x \).

**Definition.** A *compression algorithm* is an algorithm \( A \) that maps \( \{0, 1\}^* \) into \( \{0, 1\}^* \). (In cases of interest, \( |A(x)| \) will never be much larger than \( |x| \), and will be less than \( |x| \) when \( x \) contains redundancy that is “recognized” by \( A \).

**Definition.** A *parameterized compression algorithm* is a compression algorithm with compression parameter \( t \), denoted \( A^t \), where \( t \) specifies the amount of resources available to the parameterized compression algorithm. (Note that the resource is not necessarily time.)

**Definition.** The *t-resource compression complexity* of a string \( x \in \{0, 1\}^* \) given compression algorithm \( A \) and parameter \( t \in \mathbb{N} \) is

\[
C^t_A(x) = \min \{|A^q(x)| \mid 0 \leq q \leq t\}.
\]

(The minimum is taken in order to force \( C^t_A(x) \) to be nonincreasing in \( t \).)

**Definition.** The *compression complexity* of a string \( x \in \{0, 1\}^* \) relative to a parameterized compression algorithm \( A \) is

\[
C_A(x) = \min \{|A^q(x)| \mid q \geq 0\}.
\]

**Definition.** Let \( A \) be a parameterized compression algorithm, and let \( s \in \mathbb{N} \). The *compression depth* of the string \( x \) at *significance level* \( s \) is

\[
C_{\text{depth}}^s_A(x) = \max \{t \mid C_A(x) \leq C^t_A(x) - s\},
\]

Where \( \max \emptyset = 0 \).

For any given significance level \( s \) and parameterized compression algorithm \( A \), a string \( x \) is called *t-compression-deep relative to A* if \( C_{\text{depth}}^s_A(x) \geq t \) at significance level \( s \). Otherwise, \( x \) is *t-compression-shallow relative to A* at significance level \( s \).

Since the compression algorithm is parameterized by \( t \), the compression depth can be viewed graphically in the same manner as the computational depth. In Figure 2, the relationship between the compression depth of a string \( x \) and the significance parameter is shown by plotting \( C^t_A(x) \) versus \( t \). Intuitively, a string has a large compression depth if, as more resources are allowed, the compression algorithms utilizes these resources to find more subtle redundancy and further compress the string.

There are many compression algorithms used to compress data. However, not all of them are suitable for use as a method for computing a compression depth. Particular
properties must be present in a compression algorithm in order for it to be useful for computing compression depth. For the properties listed below, let $x$ be a string and let $A$ be a compression algorithm.

1. There must be a useful parameterized version of $A$.

2. The compression must be lossless. That is, there must exist a decompression algorithm $B$, such that, for all $t$, $B(A_t(x), t) = x$.

3. For all $t$, $A_t(x)$ must be feasibly computable.

There are a variety of compression algorithms that are used for many purposes. By evaluating these algorithms in terms of the requirements stated above, a suitable compression algorithm may be found that can be used to measure the compression depth.
of strings. Note that requirement (2) above eliminates many sound and video compression algorithms. These algorithms often discard information in order to achieve more compression, resulting in a somewhat degraded image after decompression. Unfortunately, the information ignored in the compression process for these algorithms often forms the very structure that makes strings deep. Thus, these types of algorithms are unsuitable for generating a depth measurement as described here.

Run-length encoding and Huffman encoding are both compression algorithms that do not yield a depth-like measurement, each for a different reason. Run-length encoding is a simple compression algorithm designed to compress picture data by encoding a long string of zeros or ones as a special code followed by the number of zeros or ones. However, this algorithm does not compress simple strings such as (01)^n. Therefore any parameterization of this compression algorithm is inadequate for the purpose of depth measurement.

Huffman encoding compresses data by using either the probability distribution or an approximation to the probability distribution over a fixed block size, and then exploiting strings with high probability to achieve compression. This technique does not yield a good depth measure for two reasons. First, the natural parameterization of the Huffman compression algorithm is block size. However, the string (01101)^n will achieve much better compression with block sizes that are multiples of 10. It is desirable that compression not fluctuate greatly with small increments of resource. Secondly, unless the probability is agreed upon in advance, the encoder must also store the string substitution table with the compressed data in order for it to be decompressed. This can be very large, obscuring any compression of the string.

\subsection{Lempel-Ziv Compression and LZ Depth}

Lempel-Ziv compression, first introduced by Lempel and Ziv \cite{LempelZiv77}, provides a good and efficient compression algorithm that can be parameterized without suffering from the blocking effects associated with Huffman encoding. Many variations of this original algorithm have since been introduced that run faster and with better compression. However, these improvements are small and the asymptotic performance of these algorithms is no better than the original Lempel-Ziv algorithm \cite{LempelZiv77}.

There are many variations of the Lempel-Ziv algorithm and an even wider variety of implementations. However, this paper utilizes the original Lempel-Ziv (LZ) algorithm for simplicity. This section describes the original algorithm and gives two examples. A careful description of a new parameterized compression algorithm based on the
original Lempel-Ziv algorithm that yields a good notion of compression depth follows. Finally, examples of compression depth using the modified Lempel-Ziv algorithm are illustrated using binary strings of various depths.

The following definitions are useful for defining the original Lempel-Ziv algorithm, as well as the parameterized version defined later in this section.

**Definition.** The prefix set of a string \( x \in \{0,1\}^* \) is the set \( X = \{ y \mid y \subseteq x \} \).

**Definition.** A valid code is a set \( X \subseteq \{0,1\}^* \) such that, for all \( x \in X \), the prefix set of \( x \) is a subset of \( X \).

**Definition.** A parsing of a string \( x \in \{0,1\}^* \) is a partition of the string \( x \) into phrases \( x_1, x_2, x_3, \ldots x_n \) such that \( x_1 \cdot x_2 \cdot x_3 \cdot \ldots x_n = x \).

**Definition.** [6] A distinct parsing of a string \( x \in \{0,1\}^* \) is a parsing of \( x \) such that no phrase, except possibly the last phrase, is the same as an earlier phrase.

**Definition.** A valid distinct parsing of a string \( x \in \{0,1\}^* \) is a distinct parsing of \( x \) such that if \( x_i \) is a phrase in the string \( x \), then every prefix \( y \) of \( x_i \) appears before \( x_i \) in the distinct parsing.

It is clear that every string \( x \) has a unique valid distinct parsing and that the set of phrases in this valid distinct parsing is a valid code. This is also illustrated graphically in Figure 3.

\[
1 \quad 11 \quad 0 \quad 10 \quad 00 \quad 111 \quad 001 \quad 1110
\]

**Figure 3:** The valid distinct parsing of the string \( x = 1110100111001110 \).

The Lempel-Ziv compression algorithm uses the valid distinct parsing of a string to encode it by replacing each phrase with a code word representing a pointer and a bit. In this scheme, the pointer indicates the longest proper prefix of the phrase, and the bit is simply the last bit of the phrase. Together, these completely specify the phrase being encoded. Because every prefix word of a phrase must also be a phrase that occurs earlier in the distinct parsing, the distinct parsing shown in Figure 3 can be augmented with arrows to show these pointer-pair codes as depicted in figure 4. By assigning an address to each parse phrase, beginning at address 1, the pairs of pointers and bits are coded in binary to yield a final compressed string as illustrated.
A graph of the Lempel-Ziv compression lengths for $0^n$, for various values of $n$, is shown in Figure 6. This figure also shows that the strings $(0000000)^n$, $(0000001)^n$ and $(10101010)^n$ are also highly compressible. On the other hand, this figure also shows that a string chosen randomly according to the uniform distribution is not compressible. Thus, the Lempel-Ziv compression algorithm exhibits all the key properties.
required for defining a compression depth algorithm, providing it can be parameterized.

Figure 6: Lempel-Ziv compression of periodic and random strings.

A Parameterized Lempel-Ziv compression algorithm

The Lempel-Ziv algorithm described above provides reasonable compression with modest computational requirements, but it also offers a natural parameterization. By restricting the number of phrases used from the distinct valid parsing, we can “cripple” the Lempel-Ziv algorithm, limiting its ability to compress data. If this restriction of the valid distinct parsing is performed properly, then simple strings such as $0^n$ compress to near-maximal even when the valid distinct parsing is so severely limited. Thus, the size of the limited valid distinct parsing forms the basis for a parameterized Lempel-Ziv compression algorithm and, ultimately, a measure of compression depth.

To simplify the exposition (and implementation) of the parameterized Lempel-Ziv algorithm, we define a *dictionary* as a rooted binary tree used to define a set of
strings. Each non-root node represents a nonempty string, corresponding to the path from the root node to that node. A left branch represents a zero bit and right branch represents a one bit. Figure 7 shows an example of a dictionary and the set of strings it represents.

![Image of a dictionary and the set of strings it represents.](image)

**Figure 7:** A dictionary and the set of strings it represents

The tree structure of the dictionary can be used to implement both the original Lempel-Ziv algorithm and a parameterized version. In the original algorithm, the dictionary represents the phrases seen so far as the input is scanned. By traversing the tree as each bit of the input is read, the next phrase in the string is determined. When this traversal leads to a leaf node, the next bit determines the parse point and a new leaf is appropriately added. This process is illustrated in Figure 8.

![Image of a tree (dictionary) to generate the distinct parsing of a string.](image)

**Figure 8:** Using a tree (dictionary) to generate the distinct parsing of a string
We parameterize the Lempel-Ziv algorithm by restricting the size of the dictionary. This is accomplished by only allowing the parameterized algorithm to add new strings to the dictionary when they are also in a master dictionary. Since the parameterized Lempel-Ziv algorithm may only add strings that are also in the master dictionary, the dictionary built by the parameterized Lempel-Ziv algorithm is bounded in size and structure by the master dictionary. Thus, by adding strings to the master dictionary, we increase a resource for compression, thereby giving a method for computing compression depth based on the Lempel-Ziv algorithm.

The process of parsing a string given a master dictionary is illustrated in Figure 9. The parse tree is obtained by labeling the node of the master dictionary with non-negative integers. Initially, all nodes are labeled 0. This label is then used to indicate whether the string represented by the node has been used in the parse. A non-zero label indicates which phrase in the parse the node represents. The label associated with the root node is always zero and meaningless. The parsing is performed in the same manner as the normal Lempel-Ziv algorithm except that only strings in the master dictionary may be added to the parse tree. (Note that the master dictionary must be at least of size 3 containing at least strings “0” and “1”. This is the smallest resource bound possible.)

![Parse Tree](image)

**Figure 9**: Process for parsing a string given a master dictionary

In the example shown in Figure 9, the first bit (a one) is read and the right branch (corresponding to reading a one) of the root node is examined. If there is no right branch, or the right branch is labeled with a zero (as in this example), then a phrase has been found. The node corresponding to the phrase found (in this case the phrase is the single bit 1) is then labeled with a 1 to indicate that it is the first phrase found. The process then repeats, starting with the next bit of input and at the root of the tree. The next bit is read (a one), and again the right branch of the root node is
examined. However, in this case the node is now labeled with a 1, indicating that the string it represents occurred earlier in the parse. Thus, the next bit of the input is read (a one), and the right branch of this node is now examined. This node is labeled 0, and thus the input is parsed with the phrase “11.” This new node is then marked with a 2, indicating it is the second new phrase in the parsing. The process continues until the entire input is consumed as shown in Figure 10.

In the above example, a key situation occurs on the fifth, seventh and eighth strings parsed. These phrases are parsed because there were no left or right nodes to examine in the master dictionary. For example, in the fifth phrase, a zero bit is read and the node labeled 3 is examined. The next bit is read (a zero) and the node labeled 3 does not have a left branch. At this point the phrase is parsed as the string 00, but is not added to the parse dictionary since there is no node to label. This is exactly the mechanism by which we “cripple” the original Lempel-Ziv algorithm to yield a parameterized version. Note that this procedure no longer parses the input into distinct phrases; however, the same Lempel-Ziv decompression algorithm may be used to retrieve the original string from the compressed string.

If the master dictionary is the same as the dictionary produced when the string is parsed with the original Lempel-Ziv algorithm, it is easy to see that the parameterized Lempel-Ziv algorithm gives the same parsing and compression as the original algorithm. In addition, any extension of a master tree of this form will also give an identical parse to the original Lempel-Ziv algorithm. Thus, as the resource is level is increased, the compression of the string tends towards the original Lempel-Ziv compression. This is shown in Figure 11.

In order to compute a compression depth measurement, several compression values must be computed with various amounts of resource. Since the compression depth measurement requires that the resource be measured by a number, we define the amount of resource to be the size (number of nodes) of the master dictionary. However, an efficient method for determining the structure of the tree at each size remains to be addressed. Ideally, the algorithm to compute the Lempel-Ziv compression depth at resource level \( n \) would evaluate the compression of the string for every master dictionary of size \( n \). However, this is computationally infeasible. Here, we use a recursive algorithm based on the master tree of size \( n - 1 \) to compute the master tree of size \( n \).

As shown in Figure 12, the master tree of size \( n - 1 \) is extended at each node having fewer than two successors by adding each possible successor, one at a time for the entire tree. The parsing algorithm defined above is executed, and the number of times the new node is referenced in the parse is counted. This is computed for each possible
new node, corresponding to each possible single legal phrase that could be added to the master dictionary. The master dictionary is then extended by the node that is referenced the maximum number of times among the candidate new nodes. Roughly, this procedure chooses to extend the master dictionary by a string which extends one of the strings currently in the dictionary by one bit and occurs the most frequently in the string to be parsed. This gives a very fast computation of the entire compression depth graph since each time the master dictionary is increased by one, only a linear number of new strings (nodes) require their frequencies to be computed.
LZ depth and examples

Now that a viable method has been found to parameterize the Lempel-Ziv compression algorithm, we may use the parameterized version to define a measure of compression depth.

Definition. The LZ depth of a string $x \in \{0, 1\}^*$ at significance level $s$, denoted by $D_s^{LZ}(x)$, is defined to be $C_{depth}^{LZ}(x)$. That is,

$$D_s^{LZ}(x) = \max \left\{ t \mid C_{LZ}(x) \leq C_{LZ}^t(x) - s \right\}.$$ 

Several strings are used to verify that the parameterized Lempel-Ziv algorithm yields a complexity measure with properties similar to computational depth. First, the LZ compression depths of simple strings such as $0^n$, $(01)^n$ and $0^n1^n$ are computed for $n = 50,000$ and these strings are shown to be shallow in Figures 13. Second, the LZ
Figure 12: Increasing the size of the master dictionary

compression depths of 100 strings of length 100,000 chosen at random according to
the uniform distribution are computed and are also shallow. The outcome for one
such string is shown in Figure 13.

Figure 13: Low LZ compression depth strings

There are two properties of LZ depth worth noting at this time. First, like compu-
tational depth, strings may be shallow for two reasons. As shown in Figure 13, a string may be shallow because it contains very little information in such a way that the LZ compression algorithm does not need a large dictionary in order to achieve near-maximal compression. In the other case, also shown in Figure 13, a string may be shallow because it contains maximal information and thus cannot be compressed, i.e. it lacks structure. Thus, the LZ compression algorithm never compresses the string to any significant degree.

The examples above exhibit strings that are shallow, but what properties of strings imply depth? In the case of computational depth, a deep string contains redundant information (structure) that is hard to compute from its smallest description. With compression depth, deep strings have buried redundancy not found quickly by the parameterized compression algorithm. For Lempel-Ziv compression depth we look at cellular automata to find examples of strings that are “LZ deep.”
Cellular Automata: Compression Depth and the Wolfram Classification

Cellular automata are massively parallel computing systems containing, in theory, an infinite number of finite-state machines, each interconnected with a small set of neighboring machines. (In practice, the number of machines is finite so that the system may be implemented.) It is standard to configure the machines in Euclidean space with a low number of dimensions, either as a line (one dimension) or as an array (two dimensions). Every processor contains the exact same program and may only communicate with its local neighbors.

In this paper, we use cellular automata as a testing ground for evaluating Lempel-Ziv compression depth. For this purpose, we restrict our use of cellular automata to one dimension and a neighborhood size of three cells. Formally, each cellular automaton in this paper is defined by a one-dimensional infinite lattice of cells, with a finite-state machine $M$ at each cell. The neighborhood of a cell consists of the cell itself and its two adjacent neighbors. The finite-state machine $M$ is defined by a set $S$ of states and a transition function $\delta$ mapping $S \times S \times S$ into $S$. Note that this is a single transition function that governs the behavior of all the finite-state machines in the cellular automaton.

A computation of a cellular automaton is specified by initializing the state of each finite-state machine at time $t_0$. At each subsequent time step $t_i$, the next state of each finite-state machine is determined by the transition function and the states of the finite-state machines covered by the neighborhood template. In order to make the computation time feasible, the experiments described here limit the number of cells to 50,000, and in some cases 30,000. For one-dimensional cellular automata, the boundary conditions at the first and last cells are resolved by connecting the last cell to the first cell.

In order to visualize the trajectory of the computation of a one-dimensional cellular automaton, a second dimension (time) is introduced. By stacking successive “pictures” of the state of each cell at successive time steps, a “waterfall-like” picture is formed. These pictures are then easily viewed by assigning colors to the different states of the transition function. In this paper, we assign state 0 to be white and all other states to be black.

Using these pictures, Wolfram [25] defined four classes of cellular automata based on the patterns they produced. While his paper contained mostly conjectures, qual-
Wolfram divided the types of patterns that evolve in cellular automata into four basic classes. In Wolfram's own words, these four classes are described qualitatively as follows.

I) Evolution leads to a homogeneous state.

II) Evolution leads to a set of separated, simple, stable or periodic structures.

III) Evolution leads to a chaotic ("random") pattern.

IV) Evolution leads to complex localized structures, sometimes long-lived.

The classification scheme described above is not well-defined, especially with respect to the definition of Class IV behavior; however, the intent of the scheme is clear. Given a random initial configuration, the cells of a Class I cellular automaton all evolve to the same state. The cells of a Class II cellular automaton evolve to simple stable or short periodic patterns, and Class III cellular automata do not evolve any patterns. The cells of Class III cellular automata remain randomly distributed. Class IV automata neither exhibit the simple structures found in Class I and Class II, nor do they behave chaotically ("randomly") like cellular automata found in Class III. These cellular automata produce complex interactions and structures that continually evolve over time. It is conjectured that cellular automata in this class support information storage, transmission, and modification. Hence, it is conjectured that universal computation could take place inside this class. Thus, they should contain "deep" (organized) information.

We examine the complexity of the patterns produced by these cellular automata by applying our Lempel-Ziv compression depth algorithm to the output they produce. Since the patterns are two dimensional and the Lempel-Ziv depth algorithm requires a single string of bits, we adopt the convention that the display bits are concatenated by using successive columns, and then processed by the Lempel-Ziv depth algorithm. Figure 14 graphs the result of Class I, Class II and Class III after they have been processed in this way. Note that the results are similar to the shallow strings displayed in Figure 13 of section 3, verifying that cellular automata in these classes are indeed
Figure 14: Compression depth of Class I, II, and III cellular automata

Figure 15: Compression depth of a Class IV cellular automaton

...compression-shallow. Compare this with the graph shown in Figure 15. This figure shows the result produced when a Class IV cellular automaton is processed in the same manner. Notice that the graph produced indicates that this cellular automaton evolved to a state with far more depth than that produced by the cellular automata in the other classes, and thus confirms that compression depth does capture and measure the complexity found in this class IV cellular automaton.
5 Cellular Automata: Compression Depth, Entropy, and Langton’s Parameter \( \lambda \)

Ordering the rule space of cellular automata

The set of all transition functions for a cellular automaton with \( S \) states and \( N \) neighbors is very large. For \( S = 8 \) and \( N = 5 \) there are \( 2^{2708} \) different transition functions. Langton [16] asked whether there was a way to partition this set so that transition functions in the same partition supported the same type of dynamic behavior in cellular automata. One obvious set of partitions would classify the four groups of dynamic behavior defined and observed by Wolfram.

Langton [16] defined a parameter \( \lambda \) as one possible method for ordering the rule space of a large class of cellular automata. With this ordering, Langton found that transition functions with lower values of \( \lambda \) evolved patterns that belonged to Wolfram’s Class I and Class II cellular automata. Higher values of \( \lambda \) produced transition functions that evolved patterns indicative of Class III cellular automata. Furthermore, Langton observed a phase transition where the dynamics of cellular automata changed from structured to chaotic over a small interval around a specific value of \( \lambda \). He denoted this “critical value” of \( \lambda \) as \( \lambda_c \) and conjectured that Class IV cellular automata reside at or near this transition.

Formally, \( \lambda \) is defined by choosing a special state \( s_q \) called the quiescent state, or ground state. Let \( n \) be the number of transitions to this special state. Then for a cellular automaton with \( |S| \) states and neighborhood size \( N \), \( \lambda \) is the ratio of the number of transitions that do \textit{not} map to \( s_q \) to the total number of transitions in the transition function \( \delta \). In terms of \( N \), \( |S| \) and \( n \),

\[
\lambda = \frac{|S|^N - n}{|S|^N}.
\]

If the \( n \) transitions to \( s_q \) are fixed, and the remaining \( |S|^N - n \) transitions are chosen randomly according to the uniform distribution from the states in \( S - s_q \), then \( \lambda \) roughly corresponds to the degree of randomness or the “temperature” of the transition function. For example, at \( \lambda = 0.0 \), all transitions map to the ground state. In this case, the “temperature” of the transition function is “absolute zero” and does not support any dynamic activity. In contrast to the above example, when \( \lambda = 1 - \frac{1}{|S|} \) then all states in \( S \) are represented equally in the transition function. This corresponds to a transition function with high temperature and produces dynamic activity similar
to Class III cellular automata.

**Qualitative Observations**

Langton performed several experiments using one-dimensional cellular automata. By viewing the behavior of several cellular automata with varying values of $\lambda$, Langton described several apparent attributes common to evolution at particular values of $\lambda$. Langton performed these experiments with a one-dimensional cellular automaton with 128 cells. The two end cells were considered neighbors for the purpose of computing the next state in the transition function. The transition function used for his experiment depended on of the cell itself, together with the two cells to the left and right, and $S$ contained 4 states. The following observations describing Langton’s experiment are paraphrased from his paper [16].

$$0.00 \leq \lambda < 0.15$$

All dynamic activity dies after at most 7 time steps. The cells of the cellular automata all enter the same state.

$$0.20 \leq \lambda < 0.40$$

Cellular automata can support simple periodic structures. The higher $\lambda$ values in this range support periodic structures of length 40 time steps.

$$0.45 \leq \lambda < 0.50$$

Cellular automata can support complicated structures with long periodic structures and long initial transient lengths.

$$0.55 \leq \lambda < 0.75$$

Cellular automata produce chaotic structures with no discernible patterns.

It is clear from the above qualitative descriptions that $\lambda$ appears to divide the space of all cellular automata rules into four distinct categories corresponding to Wolfram’s four classes. However, the picture is not this simple. Langton’s own experiments reveal that the transition from simple to chaotic occurs in a range of values of $\lambda$. In addition, the transition is not always sharp. However, with the use of other complexity measures, we may evaluate the roll of $\lambda$ more precisely.
Complexity Measures and Phase Transitions

There are many ways to measure the complexity of the resultant computation of a cellular automaton. Langton used transient length, entropy, and mutual information for this purpose. In the present paper, Langton’s results using transient length and entropy as measures of complexity are reviewed. This helps relate the work of Langton and others with the research presented here.

One simple complexity measure already mentioned defines the amount of time before a periodic structure evolves. Langton called this the transient length. However, this has little meaning in the chaotic regime where transients are essentially infinite. The ideal measure of transient length would measure the number of time steps before the cellular automata maintains a “constant” behavior. Several statistical measurements suffice to define this notion. One possible definition which can be defined for both chaotic and structured regimes follows.

**Definition:** Transient length is the time (number of time steps) required for all cells to settle (with high probability) to within one percent of their long term cell occupation probability. The cell occupation probability is defined as the probability that the cell is in the state $s_q$.

With this definition, the transient length is well-defined over the entire range of $\lambda$. Langton observed that the transient length increased for values of $\lambda$ between 0.00 and 0.50 and decreased for values between 0.50 and 0.75. This gives supporting evidence that the transition point could support universal computation, since universal computation requires arbitrarily long transients. Langton also observed that the sizes of the cellular automata did not influence the transient lengths except in the region of the transition. In this region the transient lengths grew at exponential rates with respect to the number of cells in the cellular automata, giving more evidence that universal computation could spontaneously emerge in the transition region.

Another complexity measurement uses the sampled entropy of cells to approximate the average information stored per cell in the cellular automata at a time after most transients have disappeared. This measure illustrates the sharp transition found near $\lambda_c$ with great clarity. The existence of a phase transition was shown by Langton for two-dimensional cellular automata. His results are verified in the present paper using one-dimensional cellular automata without any restrictions on the transition function.

**Definition** The Shannon entropy [21], or the information content, of a cell with state
set $S$ is defined as

$$H(S) = - \sum_{s \in S} \Pr(s) \log \Pr(s),$$

where $\Pr(s)$ is the probability that at any time $t$, the cell is in state $s$.

The procedure used to verify Langton’s results samples the state of each cell over time and then averages this result to give the sample average cell entropy. This procedure is defined as follows.

1. Initialize the cells of a one-dimensional cellular automaton to random states.
2. Run the cellular automaton until for each cell, the estimated long-term cell occupation probability changes by less than 0.0001.
3. Use the histogram to compute an estimate of the probability distribution over the state set $S$.
4. Compute the estimated entropy for each state using these probability distributions.
5. Average the entropy values to compute the value of the average entropy per cell.

It is assumed that Langton’s method was similar, although it was not explicitly stated in his paper [16].

Experiments by the author to verify Langton’s results were performed using one-dimensional cellular automata with 10,000 cells. The average entropy per cell versus $\lambda$ is shown in Figure 16 with 1,000 simulations per value of $\lambda$ for a total of 20,000 simulations. Again, the simulations show a gap between entropy values of 0.60 and 1.40 with only relatively few instances inside this gap. These simulations also exhibit the complete absence of low-entropy values at values of $\lambda$ greater than 0.5. The data from these simulations may also be converted to a series of histograms for each value of $\lambda$ by dividing the entropy into ranges and counting the number of simulations that produce average entropy values that fall into each range. By viewing this series of histograms as approximations to probability distributions and plotting them against $\lambda$ on a two-dimensional grid, the data produce the striking histogram shown in Figure 17. Notice that the transition from low to high entropy values is clearly seen as the valley between the two extremes and is located between $\lambda$ values of 0.30 and 0.40.
While Langton conjectured that universal computation and complex structures emerge at the transition point where transient length is longest, Figure 18 suggests another \( \lambda \) region, the region that gives the highest probability of producing a cellular automaton with average entropy in the transition gap. As shown in Figure 18, this is also the region where cellular automata have the largest spectrum of entropy values statistically. This figure shows the average cell entropy and its variance for each \( \lambda \) value. Notice that the variance peaks in the region between \( \lambda \) values of 0.3 and 0.4. It can be argued that universal computation also requires that the cellular automata produce both simple (trivial and chaotic) and complex structures, i.e., diversity as well as long transients.

This notion is further enforced by computing the LZ compression depth of the computations of the same set of cellular automata used in the previous experiment. In this case, we view the average depth of the cellular automata computation for various values of \( \lambda \) and entropy. As shown in Figure 19, we see that the computations that produce higher depth have \( \lambda \) values between the range 0.2 and 0.45 with entropy values in the range of 0.1 and 0.5. This roughly corresponds to the region of phase transition, where few cellular automata produce output.
Figure 17: Histogram of entropy-λ space.
Figure 18: Average and variance of entropy values versus $\lambda$.

Figure 19: LZ depth in $\lambda$-entropy space
6 Conclusion

In this paper we have defined a new, feasibly computable complexity measure motivated by Bennett’s notion of computational depth. By using a modified version of the Lempel-Ziv compression algorithm, we measure the amount of organization stored in a string or computation. We find that there is good agreement between high compression depth and Wolfram Class III cellular automata.

Of particular interest is the region of cellular automata rule space in which highly complex (perhaps universal) computation can spontaneously emerge. Langton argued that this occurs at the phase transition and \( \lambda \) value of 0.5. Simulations performed here show that this may not be the case. We show that there is a region of rule space with lambda values ranging between 0.25 and 0.5 where cellular automata produce a wide range of behavior. It is arguable that diversity is also a necessary condition for universality as well as arbitrarily long transient lengths.

Even though the results here differ from Langton’s, results by Mitchell, Hraber, and Crutchfield [7] agree with the results presented here. Crutchfield, Hraber, and Mitchell used genetic algorithms to evolve rules for cellular automata with fitness functions that encourage the attributes required for universal computation. The genetic algorithm evolved transition functions with \( \lambda \) parameters both above and below Langton’s phase transition.

Of course with compression depth, as with any computable measure, there will always be some “deep” objects whose complexities are too subtle to be detected by the algorithm, and which thus appear to be shallow. However, this does not prevent the measure from being useful in a wide variety of contexts.

In any case, compression depth, and in particular LZ depth, offers a new approach to measure the complexity of dynamical systems. While this paper has concentrated on applying LZ depth to cellular automata and the complexities generated by them, there is no reason why it could not be applied to any system that exhibits the ability to organize and structure information.
Acknowledgments

I thank Jack Lutz for suggesting this line of research and for useful discussions of these results. I also thank John Walker for finding several typographical errors.
References


