Empirical Kolmogorov Complexity

Ari Trachtenberg
Boston University,
8 St. Mary's St.,
Boston, MA 02215, USA,
trachten@bu.edu
February 1, 2018

***Abstract*—The Kolmogorov complexity of a string is the shortest program that outputs that string, and, as such, it provides a deterministic measure of the amount of information within the string that is related, but independent of, Shannon entropy. In practice, this complexity measure is uncomputable and mainly useful for deriving theoretical bounds.**

In this work, we consider the task of empirically computing Kolmogorov complexity for short strings. We design and build a virtual One Instruction Set Computer that we use to empirically compute the Kolomogorov complexity of short strings (within this computing model). It is hoped that this approach could inform novel cloud-based data compression methods that more effectively digest human data than standard Lempel-Ziv based compression.

The ability to compress data, both at rest on storage systems and in transit over communications lines, has taken on increasing significance in our age of data abundance and sharing, where it is often much easier and more efficient to create data than to distribute it to all interested parties. Most of the common lossless compression schemes are ultimately based on the venerable Lempel-Ziv compression algorithm [1], which asymptotically compresses a stationary, ergodic source to its entropy rate [2]. The slow convergence of this algorithm to the entropy rate has led to a number of variants [3] aimed at speeding up the convergence.

The key point, however, is that these approaches view the data to be compressed as the result of a random statistical source, whose complexity can be measured in terms of Shannon entropy [4]. As a result, data that is produced by a simple, predictable process may still provide very poor compression. For example, the string of increasing base-10 integers:

$$1\,2\,3\,4\,5\,\ldots\,1000$$

is compressed into 1850 bytes with the standard `gzip` compressor, even though the pattern is produced losslessly by only 42 bytes of Perl code:

```
foreach $ii (0..1000) {
  print "$ii";
}
```

Indeed, the divergence between the length of the source code producing an output and the Lempel-Ziv compression of that output can be arbitrary large. Replacing 1000 in the example above with $10,000$ adds one byte to the source code length, but $20,352$ bytes to the `gzip`'ed compression. Adding more zeroes widens this gap as far as desired.

Kolmogorov complexity attempts to address this deficiency by defining the complexity of a string not in terms of the statistical properties of its source, but in terms of the shortest program needed to produce the string. Although the expected Kolmogorov complexity of uniformly random length n strings is asymptotically equal to their entropy [5], this is heavily influenced by the large number of strings that are not compressible. Many strings produced and used by humans are, in fact, the result of natural, determinative processes and, it is hoped, may actually be compressible.

Unfortunately, determining the Kolmogorov complexity of a string is not generally possible with Turing-based computers [4], and this has hampered practical progress on this front thus far. However, the increased computing and memory power available today does leave room for empirically computing Kolmogorov complexity of short strings, based on a chosen reference language. This, in turn, leads to the hope of better compression of some human generated texts.

Ultimately, the goal of this work is to provide a means of compressing text closer to its Kolmogorov complexity. We do this by empirically running many different programs and recording their outputs, in an attempt to identify the shortest program for producing each output. For efficiency and compactness, our programs are written in a customized byte-level One-Instruction Set Computer, and they exhibit several desirable properties:

- *Compression of non-statistical sources* - they may be able compress some data that can be generated from simple code even if it does not exhibit obvious textual patterns.
- *Efficient compression and decompression* - online aspects for both processes are linear-time, with the bulk of the computational effort being off-line.
- *Contained decompression* - although compression requires significant resources and offline processing, decompression can be accomplished on devices with limited computation and communication resources.

I. Background

Our results are based on two well-studied ideas in the literature, for which we provide a brief background herein,

many without explicit proofs (which are available in the corresponding references).

1) Kolmogorov complexity: Intuitively, the Kolmogorov complexity of a string is the length of the shortest program needed to produce the string. In this vein, a binary string of l ones, for example, has a lower Kolmogorov complexity than a binary string of l random bits: the former can be expressed in terms of a short program that writes a binary one l times, whereas the latter, presumably, would suffer from a lack of randomness if it were produced by a simple program.

Ultimately, the Kolmogorov complexity $C(x)$ of a string x depends on the language of the program that is producing it, and all of this can be formalized in terms of a "universal" partially recursive function; the interested reader is referred to the seminal work on the topic by Ming and Vitányi [6] or the more accessible summary by Kouckỳ [7]. The main theorems from these works of relevance to us are as follows.

First of all, the Kolmogorov complexity of a string cannot be much larger than the length of the string itself, since one way of computing a string is simply to print it out.

Theorem I.1 (Trivial upper bound)**.** *There exists a constant c such that, for all strings x,*

$$C(x) \leq |x| + c,$$

where $|x|$ denotes the length of x.

A simple pigeon-hole counting argument also shows that there exists strings that do not do much better than this upper bound.

Theorem I.2. *There exists a string x such that*

$$C(x) \geq |x|.$$

These strings are called *Kolmogorov Random*. Indeed, a simple generalization of this argument shows that most strings have high Kolmogorov complexity [6, 8]:

Theorem I.3. *For any constant $c \geq 0$, there are at least $2^n - 2^{n-c} + 1$ strings x of length n for which*

$$C(x) \geq |x| - c.$$

In other words, at least half of all strings compress by at most one character, and three-fourths compress by at most two characters.

Unfortunately, no program can generally compute the Kolmogorov Complexity of a string [6][Theorem 2.3.2], or even whether a string is Kolmogorov Random [8]. In more technical terms, these problems are both not *recursive*.

The silver lining is that we can *enumerate* the set of strings with a given Kolmogorov complexity: [6][Theorem 2.7.1]

Theorem I.4. *The set $A = \{(x, a) : C(x) \leq a\}$ is recursively enumerable.*

Indeed, it is this last point that we attempt to exploit by empirically and explicitly computing the Kolmogorov complexity of short strings. To do this concretely, we fix a very specific programming language, described in the next section.

2) One Instruction Set Computer: A One Instruction Set Computer (OISC) is a Turing-complete machine based on a single machine instruction. Created initially as an educational tool [9], various OISC machines have been proposed since in the literature (see [10] and the citations therein). For example, the SBN instruction - Subtract and Branch if Negative - accepts three operands: one operand is subtracted from a second, with a branch to a third operand address if the result is negative.

The OISC closest to our approach is the SUBLEQ machine, which operates according to the following C-style pseudocode provided in [11] (together with a statement, but not a proof, of its Turing completeness):

Algorithm 1 SUBLEQ

```
procedure SUBLEQ(a,b,c)
    *b -= *a
    if *b ≤ 0 then
        goto c
    end if
end procedure
```

II. Design

Our main goal in this work is to design and implement a One Instruction Set Computer (OISC) that will allow empirical evaluations of the Kolmogorov complexity of various short strings.

We will then use these evaluations to produce a more general compressor, and corresponding decompressor. We will be specifically interested in several specific properties.

- **Universality of input** - it should be possible to run any binary sequence through the compressor and produce an output, though, of course, the output might not be shorter than the input. Under this regime, it is possible to run the output of a compressor through the very same compressor, in the hopes of getting a shorter result.
- **Universality of output** - it should be possible to run any binary sequence through the decompresser to produce an input.
- **Contained decompression** - the algorithm for decompression must require as little resources as possible, so that it can be effectively placed, for example, on devices with limited computation, storage, or networking capabilities. Compression, on the other hand, could require significant network or computing resources.

We next detail the technical specifications of the OISC that we implement.

A. The OISC

Our OISC, called `SUBLEQ_M` , is based on the SUBLEQ machine described in Section I-2

TAPE									
Index:	0	1	2	3	4	5	...		
Tape (hex):	90	91	92	93	94	input	...	0	...
ACC = 0									

Fig. 1. Initial state of the SUBLEQ_M machine.

a) Initialization.: Our machine runs on a semi-infinite tape TAPE of unsigned 8-bit bytes, starting at memory index 5, to which the read head points at the beginning of a run; indices 0, through 4 of the tape head have, respectively, bytes 0x90 through 0x94 pre-initialized so as to allow some initial backward movement of the read head from the first instruction.

The machine input is placed on the tape, byte by byte, starting at index 5, and it is zero-extended through the remainder of the tape. In addition the machine has one accumulator register ACC , which is initialized to the zero byte. The state of the machine at startup is thus summarized by Figure 1.

b) Execution.: At each step, our machine reads one unsigned byte B from the tape read head, and logically divides it into two four-bit nybbles $B1$ (low-order bits) and $B2$ (high-order bits), so that $B = B2||B1$ and $||$ denotes concatenation. The machine then executes according to Algorithm 2.

Algorithm 2 SUBLEQ_M

```
procedure SUBLEQ_M(B1, B2)
    TAPE [B1] -= ACC
    ACC=TAPE [B1]
    if TAPE [B1] % 2 == 0 then
        goto B2
    end if
end procedure
```

Note that the addresses represented by $B1$ and $B2$ are interpreted *relative* to the current read head, and shifted by -8, so as to allow head movement backwards (up to 8 steps) or forwards (up to 7 steps). In addition, the subtraction operation is implemented modulo 256, since the result must be an unsigned byte.

c) Termination: Execution proceeds until one of the following terminating conditions:

- Memory is accessed at index $i < 0$. *(hang)*
- The machine attempts to evaluate a zero byte. *(overt termination)*
- More than 1000 computation steps have been completed. *(time bound)*

The output of the machine is the contents of all tape memory after the initialization segment (i.e. symbols at indices 0 through 4) and until the last byte accessed during the machine computation.

d) Example: Figure 2 demonstrates the evolution of the computation of the machine on the input 0x88acd4, expressed in hex, where the tape byte in **bold** is the byte currently under the read head.

Step	Head	ACC	Tape (hex)
0	5	0	90 91 92 93 94 **78** ac d4
1	4	120	90 91 92 93 **94** 78 ac d4
2	5	24	18 91 92 93 94 **78** ac d4
3	4	96	18 91 92 93 **94** 60 ac d4
4	5	184	b8 91 92 93 94 **60** ac d4

Fig. 2. Sample run of our machine.

The machine hangs after iteration number 4, because it tries to access the memory at address -3. The result of the calculation is the hex string 0x60acd4, which appears on the tape after the initialization segment 0x9091929394.

III. Compression

The SUBLEQ_M variant described in Section II-A produces a matching from its input to its Machine-executed output. We can use this matching to provide a general purpose data compressor, and decompressor, based on an estimated (or, in some cases, exact) Kolmogorov complexity of short substrings.

A. Encoding

Compression proceeds in two phases: off-line data collection, and on-line compression.

1) Off-line data gathering: Initially, we exhaustively enumerate a number of short strings and evaluate our SUBLEQ_M machine on each putative input, recording the resulting (input,output) pair in a database. The database is uniquely keyed against the output, so that if two inputs produce the same output, only the shorter of the inputs is maintained. In this way, the database entry provides an upper bound on the Kolmogorov complexity of an output string within the context of our SUBLEQ_M machine. The bound is tight if all shorter inputs have been processed. We will denote the mapping of output to input strings by $\delta : \Sigma^* \longrightarrow \Sigma^*$, where Σ denotes the alphabet over which strings are chosen, and $*$ does the Kleene closure.

2) Online compression: When provided a specific plaintext string s to compress, we iteratively apply Algorithm 3 until the entire string has been processed.

Algorithm 3 Compression framework

- **Identify** a substring s_x of s that appears as an output in our database.
- **Record** the input $\delta(s_x)$ corresponding to this output as part of the compressed version of the string.
- **Remove** s_x from s, leaving the rest of the string for processing.

Several important technical details figure prominently in the effectiveness of this compression, as described herein.

a) Multiple substrings: There may be several substrings that appear as outputs in the database that has been compiled off-line. In this case, it may be difficult to make the choices

`INSTANCE:`
A string s and a collection of substrings s_i of s, $i = 0 \ldots n-1$, each with a compressed length $c(s_i)$.
`SOLUTION:`
A set $I \subseteq 0 \ldots n-1$ such that $s = ||_{iinI} s(i)$.
`MEASURE:`
Minimize $\sum_{iinI} c(s_i)$.

Fig. 3. Substring Compression Problem.

that produce the optimal compression. The formal problem is stated in Figure 3, using the standard notation $||$ to denote string concatenation.

Theorem III.1. *`Substring Compression` is NP-hard.*

Proof: The proof follows from a transformation from the NP-complete Knapsack problem [12]. In the decision version of that problem, one is given a set U and function $s(u)$ and $v(u)$ for each element $u \in U$, with the task of deciding whether there is a subset $U' \subseteq U$ such that $\sum_{uinU'} s(u) \leq B$ and $\sum_{u \in U'} v(u) \geq K$. The functions $s(\cdot)$ and $v(\cdot)$ represent, respectively, the *size* and *value* of an element in U, in the language of the problem. We will be interested in a variant of the optimization version of the problem, where K is not given (but rather the sum $\sum_{u \in U'} v(u)$ needs to be maximized) and the bound B needs to be met exactly (i.e., $\sum_{uinU'} s(u) = B$).

To relate a given Knapsack instance to a Substring Compression problem, we:

- Number the elements of $U = u_0, u_1, u_2, \ldots, u_{n-1}$.
- Set our string to be $s = 1^B$ (i.e. B ones in a row).
- Make our substrings $s_i = 1^{s(u_i)}$.
- Define our costs $c(u_i) = -v(u_i)$.

Since all substrings are over the same character space, a solution to the Substring Compression problem involves finding substrings whose lengths sum to B, corresponding to Knapsack elements whose sizes sum to B. The optimization involves minimizing sum of $c(u) = -v(u)$ for elements u corresponding to chosen substrings; this corresponds directly to maximizing the sum of $v(u)$ for the same elements, thereby solving the Knapsack problem. ∎

One standard approach to such a computational challenge is to use a greedy algorithm, which, at each iteration, picks the substring the produces the greatest local compression ratio. In other words, we iteratively (until exhaustion) pick s_x such that $\frac{|s_x|}{|\delta(s_x)|}$ is maximized.

b) Non-existence of substring: There are no guarantees that our set of outputs is complete, in that it will cover any possible plaintexts. In other words, it is possible that $\delta(\sigma)$ is not defined in our database for *any* prefix σ of s, in which case our iterative compressor will fail. Such cases can be coded with a flag and the relevant prefix of the string encoded in plain text.

B. Decoding

The different inputs that are recorded by Algorithm 3 need to be collected in an unambiguous manner into a single file. The decoder reads this file and extracts, one by one, the input sequences, which are processed through our `SUBLEQ_M` machine to produce decompressed outputs. The outputs are stitched together to produce the original plain text file.

We next detail several approaches to the question of how best to unambiguously encode inputs so that they may be properly decoded.

1) Size-delimiting: One of the easiest methods of unambiguously collecting the inputs is to reserve one byte at the beginning of each input sequence to mark its length. This approach fixes the overhead of encoding based on the shortest input length, which may be significant. For example, if compression is restricted to input lengths that are three bytes or more, then the overhead of size-delimited encoding is capped at one of every four bytes (i.e. 25%).

On the other hand, size-delimited encoding has several useful properties:

- **Parallelization** - Because the starting point of each input is known at decoding time, they can each be decoded in parallel on their own version of our `SUBLEQ_M` machine.
- **Databaseless decoding** - The decoder does not need access to the large database that is used in encoding, since it is just running the `SUBLEQ_M` machine on each input instance. This means that decoding can take place on computationally limited devices without the need for a network connection, or, for that matter, any knowledge of the original database.
- **Re-encoding** - The output of the encoder can be re-encoded many times with the very same encoder, and peeled apart, layer by layer, by the decoder.

A variant the size-delimited encoder is the space-delimited encoder, which uses the all zero byte (treated as a "halt" instruction by our machine) to delimit input sequences.

2) No delimiting: An alternate approach to input encoding involves the use of *no* delimiters. The preliminary results in Section IV show a coverage area of roughly 41%, meaning that about 59% of plain texts will not match an output sequence in our database. This non-match can then be used as an oracle for correct decoding.

Given an encoding e, the decoding process accumulates a result res as follows:

In this way, finding a plaintext that does not match our database δ indicates that decoding has failed. To avoid restricting plaintexts to only those strings that match our database, we can add a control bit to indicate whether an upcoming text represents (a) an input, or (b) a 16-byte plaintext sequence.

The benefit of this approach is that it incurs a single bit of overhead per input encoding (or none, if the plaintexts are restricted to those in the database). The deficits are that:

- Decoding may produce an incorrect result, if incorrect

```
procedure DECODE(e, res)
    if e is empty then
        return res
    end if
    for l ← 16 down to 1 do
        σ ← the l-character prefix of e
        α ← SUBLEQ_M applied to σ
        if δ(α) == σ then
            Decode(e − σ, res + α);
        end if
    end for
end procedure
```

encoding subsequences match our database nevertheless.
- Decoding requires access to the database δ and can no longer proceed on devices that do not have network access.

3) Prefix trees: A final method for encoding is to enforce prefix-freedom on all inputs in our database, by throwing out those inputs that are prefixes of other inputs. This combines some of the benefits and deficits of the previous approaches, in that:

- Encoding overhead is now zero.
- Decoding can proceed in parallel, as all delimits can be identified easily.
- One may encode the result of an encoder iteratively.
- Decoding requires access to the (potentially large) prefix-tree.

In all these approaches, the advantage of a Kolmogorov-based compression is that it might be possible to compress strings that are not typically compressible by Lempel-Ziv approaches, such as code binaries or, possibly, encrypted text.

IV. Results

Our initial evaluations involved evaluating SUBLEQ_M on all one, two, and three character ASCII inputs, and roughly a quarter of four character ASCII inputs.

a) Statistics: Our evaluation resulted in a database of roughly $100GB$ that contained $981,438,660$ entries. Of these:

1) 47.19% represented entries where the input and output differ.
2) 97.99% of inputs were subsequences of their output.
3) The average output to input length ratio was 1.33.
4) 41.33% of plaintexts have a prefix that matches a database key.

The first statistic above denotes the number of non-trivial compressions in the database. Roughly 53% of the strings evaluated do not change at all under the transformation of SUBLEQ_M . Indeed, the second statistic shows that most input sequences simply add some bytes when evaluated under our machine. Still, there is a small, but non-trivial compression ratio among the database entries, as denoted by the third statistic. The final statistic is calculated by producing the subset S of database entries whose outputs form a prefix-free set, and evaluating the Kraft-McMillan fraction [2] on the lengths of this set:

$$\sum_{s \in S} \frac{1}{256^{|s|}}.$$

A. Lengths

We also produced a matrix of the input-output lengths in our database:

$$\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 100 & 0 & 0 & 0 & 0 \\
0 & 16 & 27070 & 0 & 0 & 0 \\
0 & 18 & 4421 & 7309384 & 0 & 0 \\
0 & 20 & 4987 & 1223847 & 521346969 & 0 \\
0 & 22 & 5548 & 1384579 & 88166245 & 0 \\
0 & 24 & 6127 & 1545181 & 99405600 & 0 \\
0 & 26 & 6698 & 1712531 & 110654422 & 0 \\
0 & 27 & 7024 & 1810602 & 118406082 & 0 \\
0 & 0 & 346 & 172584 & 13748939 & 0 \\
0 & 0 & 4 & 91781 & 9385823 & 0 \\
0 & 0 & 3 & 1969 & 3424496 & 0 \\
0 & 0 & 9 & 4589 & 520467 & 0 \\
0 & 0 & 3 & 1844 & 154244 & 0 \\
0 & 0 & 1 & 1073 & 296472 & 0 \\
0 & 1 & 264 & 69725 & 487013 & 0 \\
0 & 0 & 0 & 362 & 34671 & 0 \\
0 & 0 & 0 & 66 & 9906 & 0 \\
0 & 0 & 0 & 3 & 603 & 0 \\
0 & 0 & 0 & 2 & 2642 & 0 \\
0 & 0 & 0 & 0 & 1149 & 0 \\
0 & 0 & 0 & 0 & 35 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

In this matrix, the value of cell $[i, j]$ is the number of database entries where the input has length i and the output has length j. For example, there are 4421 entries where the input has length 2 and the output has length 3.

Several interesting artifacts arise from this table. Since we have evaluated all strings of length 1, 2, and 3 in our machine, the matrix diagonal revealse the fraction of Kolmogorov random strings we have encountered:

$$\frac{100 + 27070 + 7309384 + 521346969}{981438660} \approx 53.7\%.$$

We can also test Theorem I.3 and various other Kolmogorov properties. Finally, the highest compression ratio we can expect with this table, per iteration, is $\frac{22}{4} \approx 5.5$.

V. Acknowledgment*

The author gratefully acknowledges the feedback of Aryeh Kontorovick on an earlier draft of this work. This work was supported in part by the US National Science Foundation under grants under grant CCF-1563753.

VI. Conclusions

We have outline an approach for empirically computing the Kolmogorov complexities of short strings, and using this to compress data without resorting to statistical evaluations, as is done with Lempel-Ziv-based compressors. Our approach is based on the use of a customized One Instruction Set Computer to build a large database of the shortest programs that produce specific strings. Though we are able to compress some texts, additional efforts are needed to produce practical compression benefits at large scale. We believe that such scale is well within the current state of the art in computing capability, and that further development of this approach could allow for effective compression of some binary input sources without obvious patterns.

References

[1] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[2] T. M. Cover and J. A. Thomas, *Elements of information theory*. John Wiley & Sons, 2012.

[3] T. A. Welch, "A technique for high-performance data compression," *Computer*, vol. 6, no. 17, pp. 8–19, 1984.

[4] P. Grunwald and P. Vitányi, "Shannon information and kolmogorov complexity," *arXiv preprint cs/0410002*, 2004.

[5] L. Ming and P. M. Vitányi, "Kolmogorov complexity and its applications," *Algorithms and Complexity*, vol. 1, p. 187, 2014.

[6] L. Ming and P. Vitányi, *An introduction to Kolmogorov complexity and its applications*. Springer Heidelberg, 1997.

[7] M. Kouckỳ, "A brief introduction to kolmogorov complexity," *MÚ AV ČR, Praha*, p. 4, 2006.

[8] L. Trevisan, "Notes on kolmogorov complexity." [Online]. Available: https://people.eecs.berkeley.edu/~luca/cs172/notek.pdf

[9] F. Mavaddat and B. Parhami, "Urisc: the ultimate reduced instruction set computer," *International Journal of Electrical Engineering Education*, vol. 25, no. 4, pp. 327–334, 1988.

[10] W. F. Gilreath and P. A. Laplante, *Computer architecture: A minimalist perspective*. Springer Science & Business Media, 2003, vol. 730.

[11] P. J. Nürnberg, U. K. Wiil, and D. L. Hicks, "A grand unified theory for structural computing," in *International Symposium on Metainformatics*. Springer, 2003, pp. 1–16.

[12] M. R. Garey and D. S. Johnson, *Computers and intractability*. wh freeman New York, 2002, vol. 29.