CDM

Program Size Complexity

Klaus Sutner

Carnegie Mellon University

35-Kolmogorov 2017/12/15 23:17

1. Straight Line Programs

- Program-Size Complexity
- Prefix Complexity
- Incompleteness

So far we have seen two approaches to randomness:

- Density: von Mises, all reasonable subsequences must have density $1/2$.
- Genericity: Martin-Löf, must survive a universal sequential test.

The Auswahlregel method of Mises does not quite work, and both his and Martin-Löf's approach do not really address the question of when a string of, say, 1000 bits is random.

There is an answer to this due to Kolmogorov and Chaitin that also uses computability but focuses on finite strings rather than infinite ones. The key idea is program size complexity: the size of the smallest program that generates the string.

First a harmless example.

We have seen a number of ways to measure the complexity of computational problems, in particular decision problems (aka languages).

One problem with this approach is that it only applies to infinite collections of problems, it fails automatically for finite questions.

So how can we make sense out of the problem of measuring the complexity of a single object, say, a binary string?

First, a harmless example.

Consider an integer polynomial $f \in \mathbb{Z}[x]$.

Usually a polynomial is given as a coefficient list $(a_d, \ldots, a_0)$ as in

$$f(x) = a_d x^d + a_{d-1} x^{d-1} + \ldots + a_1 x + a_0$$

Given $b \in \mathbb{Z}$ it is then easy to evaluate f at b: Horner's rule guarantees that the number of multiplications is no worse than the degree of f:

$$f(b) = ((\ldots(a_d b + a_{d-1})b + a_{d-2})b + \ldots)b + a_0$$

In general, Horner's rule is a perfectly good way to evaluate polynomials. However, for specific polynomials f, there might be a better way to compute $f(b)$ than the general algorithm.

Here is a rudimentary type of program that can be used to evaluate polynomials. We will use the indeterminate x in the examples below, but think of x as a given integer.

Intuitively, a straight-line program is a program containing no conditionals and no loops; only arithmetic operations. Note that in this model we have to compute the coefficients from scratch.

Definition

A straight-line program of length n is a sequence of n assignments of the form

$$\begin{aligned} v_1 &= 1 \\ v_2 &= x \\ v_i &= v_l \text{ op } v_r \qquad \text{where } 0 \leq l, r < i \leq n. \end{aligned}$$

The only allowed operations are $\text{op} = +, -, \times$. The output of the program is the value of v_n.

It is clear that any SLP computes a polynomial function: just substitute $v_l \text{ op } v_r$ for v_i everywhere, and the resulting expression is a polynomial in x.

Also, any polynomial can be computed by a SLP.

$$
\begin{array}{ll}
v_1 = 1 & 1 \\
v_2 = x & x \\
v_3 = v_1 + v_1 & 2 \\
v_4 = v_2 * v_2 & x^2 \\
v_5 = v_4 * v_3 & 2x^2 \\
v_6 = v_5 - v_1 & 2x^2 - 1 \\
v_7 = v_6 * v_2 & 2x^3 - x \\
v_8 = v_7 + v_1 & 2x^3 - x + 1
\end{array}
$$

Sometimes a very short program can compute long polynomials (if they have a lot of structure that can be exploited).

$$1 + 8\,x + 28\,x^2 + 56\,x^3 + 70\,x^4 + 56\,x^5 + 28\,x^6 + 8\,x^7 + x^8$$

$v_1 = 1$	1
$v_2 = x$	x
$v_3 = v_2 + v_1$	$1 + x$
$v_4 = v_3 * v_3$	$(1 + x)^2$
$v_5 = v_4 * v_4$	$(1 + x)^4$
$v_6 = v_5 * v_5$	$(1 + x)^8$

Definition

The SLP-complexity of a polynomial $f \in \mathbb{Z}[x]$ is the least n such that a straight-line program of size n computes f.

Notation: $\mathsf{slc}(f)$.

There is an important open problem about the number of roots of such polynomials (Smale's Fourth Problem).

> Is the number of integer roots of an arbitrary integer polynomial polynomially bounded by its SLP-complexity?

I.e., is there some constant c such that for all $f \in \mathbb{Z}[x]$:

$$\#\text{ roots of } f \leq \mathsf{slc}(f)^c$$

How bad can it be? Roots of real polynomials have been studied to death, there is a rich literature out there.

A classic:

> N. Obreschkoff
>
> Nullstellen Reeller Polynome
>
> VEB Deutscher Verlag der Wissenschaften, 1963

Alas, we are dealing with integer roots, none of the sophisticated results of analysis apply directly to our problem.

Note that the degree of a polynomial with SLP-complexity n can be much larger than n, so a simple degree argument won't work.

$$
\begin{aligned}
v_1 &= 1 && 1 \\
v_2 &= x && x \\
v_3 &= v_2 * v_2 && x^2 \\
v_4 &= v_3 * v_3 && x^4 \\
v_5 &= v_4 * v_4 && x^8 \\
&\vdots \\
v_n &= v_{n-1} * v_{n-1} && x^{2^{n-2}}
\end{aligned}
$$

Remember 15-251?

Generating a term x^r is really the same as the old Addition Chain problem:

Remove x, subtraction and multiplication from our SLPs.

Then we can only compute integers, starting at 1 and using addition.

BTW, many authors do not to charge for the instruction $v_1 = 1$.

Somewhat surprisingly, it is quite difficult to compute the SLP-complexity of a plain integer in this reduced setting.

$$
\begin{aligned}
v_1 &= 1 && 1 \\
v_2 &= v_1 + v_1 && 2 \\
v_3 &= v_2 + v_2 && 4 \\
v_4 &= v_3 + v_3 && 8 \\
v_5 &= v_4 + v_4 && 16 \\
v_6 &= v_5 + v_4 && 24 \\
v_7 &= v_6 + v_3 && 28 \\
v_8 &= v_7 + v_2 && 30
\end{aligned}
$$

This is the "obvious" SLP; it shows that $\mathsf{slc}(30) \leq 8$.

Is this really the shortest program for 30?

$$
\begin{aligned}
v_1 &= 1 && 1\\
v_2 &= v_1 + v_1 && 2\\
v_3 &= v_2 + v_2 && 4\\
v_4 &= v_3 + v_3 && 8\\
v_5 &= v_4 + v_2 && 10\\
v_6 &= v_5 + v_5 && 20\\
v_7 &= v_6 + v_5 && 30
\end{aligned}
$$

So we have $\mathrm{slc}(30) \leq 7$. Is that it?

$$
\begin{aligned}
v_1 &= 1 && 1 \\
v_2 &= v_1 + v_1 && 2 \\
v_3 &= v_1 + v_2 && 3 \\
v_4 &= v_2 + v_3 && 5 \\
v_5 &= v_4 + v_4 && 10 \\
v_6 &= v_4 + v_5 && 15 \\
v_7 &= v_6 + v_6 && 30
\end{aligned}
$$

It is true that $\mathrm{slc}(30) = 7$, but it takes a bit of effort to prove this.

Also note that the solution is not unique.

Developing good intuition for SLP programs is also quite hard, common sense often leads one astray.

E.g., it is clear that $\mathsf{slc}(2n) \leq \mathsf{slc}(n) + 1$.

But the bound cannot be replaced by equality:

$$\mathsf{slc}(2 \times 191) = \mathsf{slc}(191) = 12$$

$$\mathsf{slc}(2 \times 375494703) < \mathsf{slc}(375494703)$$

Developing good intuition for SLP programs is also quite hard, common sense often leads one astray.

For example, we could clean up SLPs a bit by insisting that all instructions looks like so:

$$v_{\mathbf{i}} = v_{\mathbf{i-1}} + v_r$$

These SLPs are called Brauer chains.

It might be tempting to think that slc(n) can always be realized via a Brauer chain.

Wrong!

But the least counterexample is quite large: the conjecture fails first for $n = 12509$.

Exact computation of $\mathsf{slc}(n)$ is difficult; a suitably constructed version of the problem is known to be $\mathbb{NP}$-complete. But even just lower and upper bounds are pretty hard to obtain:

$$\log n + \mathsf{ds}\, n - O(1) \leq \mathsf{slc}(n) \leq \log n + \log n(1 + o(1))/\log\log n$$

Here the logs are base 2 and $\mathsf{ds}\, n$ denotes the binary digit sum of n.

The LHS corresponds vaguely to the "obvious" algorithm based on repeated squaring.

This seems to be state of the art.

It is natural and easy to generalize SLPs a bit:

- Input variables $x_1, \ldots, x_k$ (only on the right hand side).
- Internal variables $v_1, \ldots, v_n$ (both sides).
- Admissible operators op.

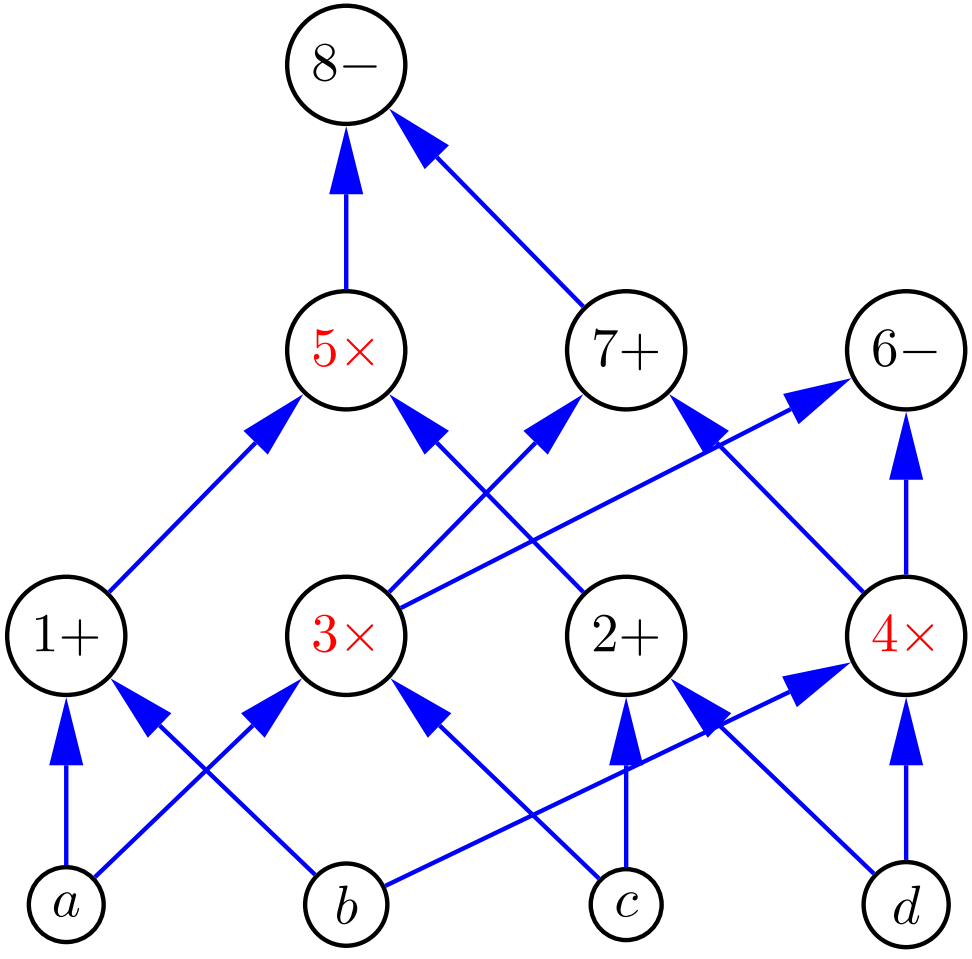
This is exactly the same as feedback-free circuits, where each gate has exactly two inputs; aka fan-in 2. Fan-out is unbounded here.

Constructing small circuits is very closely related to constructing short SLPs.

$$(a + ib) \times (c + id) = (ac - bd) + (ad + bc)i$$

Seems to require 4 real multiplications. But it doesn't.

$$\begin{aligned}
v_1 &= a + b & a + b \\
v_2 &= c + d & c + d \\
v_3 &= a * c & ac \\
v_4 &= b * d & bd \\
v_5 &= v_1 * v_2 & ac + ad + bc + bd \\
v_6 &= v_3 - v_4 & ac - bd \\
v_7 &= v_3 + v_4 & ac + bd \\
v_8 &= v_5 - v_7 & ad + bc
\end{aligned}$$


8−
5×
7+
6−
1+
3×
2+
4×
a
b
c
d

Here is another situation where the size of a program naturally appears as an interesting measure.

Suppose Alice wants to send Bob the first 10000 digits of π. In plain ASCII this requires transmission of 10000 bytes.

We could use some standard compression program to reduce the file size:

ASCII	10000 bytes
`gzip`	5106 bytes
`bzip2`	4369 bytes

Bob can then use the appropriate decompression program to retrieve the digits.

But there is a much better solution: don't send the digits, compressed or otherwise, but send a program that computes them.

```
long a[35014], b, c = 35014, d, e, f = 1e4, g, h;

main()
{
    for( ; b=c-=14; h=printf("%04ld",e+d/f) )
        for( e=d%=f; g=--b*2; d/=g )
            d = d*b + f*( h ? a[b] : f/5 ), a[b] = d%--g;
}
```

After removal of all the superfluous white-space this is only 140 bytes long.

Of course, Bob has to work a bit harder: compile and execute.

- Straight Line Programs

2. Program-Size Complexity

- Prefix Complexity

- Incompleteness

0101010101010101010101010101010101010101010101010101010101010101

0101101110111101111101111110111111101111111101111111101111111111

1011010100000100111001100110011111110011101111001100100100000100

0011100101100001011001010100001110011010111111001010000110010011

Which is the least/most complicated?

A good way to think about this, is to try to compute the first 1000 bits of the "corresponding" infinite bit sequence.

- $(01)^\omega$
- concatenate 01^i, $i \geq 1$
- binary expansion of $\sqrt{2}$
- random bits generated by a measuring decay of a radioactive source `http://www.fourmilab.ch`.

So the last one is a huge can of worms; it looks like we need physics to do this, pure math and logic are not enough.

Examples like these strings and the π program naturally lead to the question:

What is the shortest program that generates some given output?

To obtain a clear quantitative answer, we need to fix a programming language and everything else that pertains to compilation and execution.

Then we can speak of the shortest program (in length-lex order) that generates some fixed output.

Note: This is very different from resource based complexity measures (running time or memory requirement). We are not concerned with the time it takes to execute the program, nor with the memory it might consume during execution.

In the actual theory, one uses universal Turing machines to formalize the notion of a program and its execution, but intuitively it is a good idea to think of

- C programs,
- being compiled on a standard compiler,
- and executed in some standard environment.

So we are interested in the short C program that will produce same particular target output. As the π example shows, these programs might be rather weird.

Needless to say, this is just intuition. If we want to prove theorems, we need a real definition.

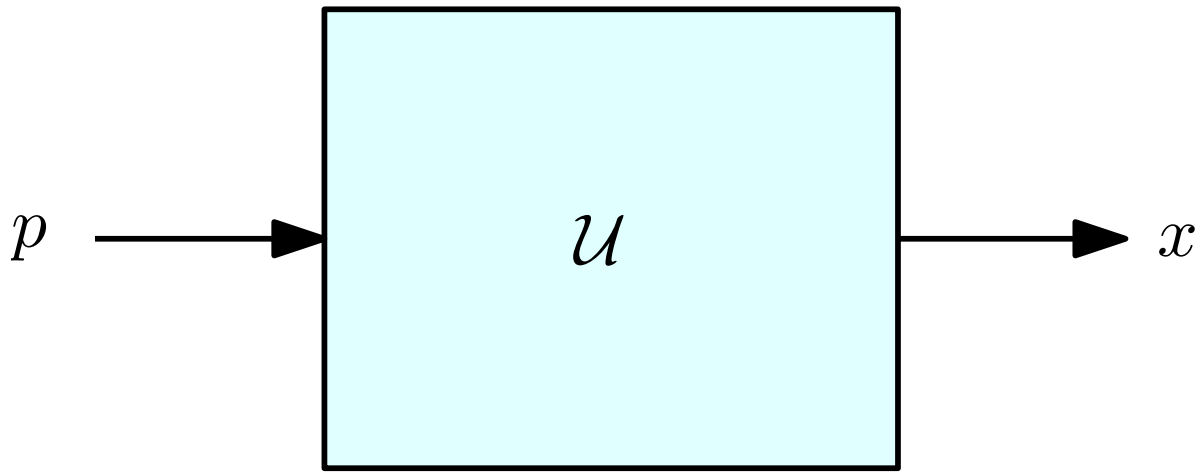
Consider a universal Turing machine $\mathcal{U}$.

For the sake of completeness, suppose $\mathcal{U}$ uses tape alphabet $\mathbf{2} = \{0, 1, b\}$ where we think of b as the blank symbol (so each tape inscription has only finitely many binary digits).

The machine has a single tape for input/work/output.

The machine operates like this: we write a binary string $p \in \mathbf{2}^\star$ on the tape, and place the head at the first bit of p. $\mathcal{U}$ runs and, if it halts, leaves behind a single binary string x on the tape.

We write $\mathcal{U}(p) \simeq x$.


p
$\mathcal{U}$
x

Definition

For any word $x \in \mathbf{2}^*$, denote $\widehat{x}$ the length-lex minimal program that produces x on $\mathcal{U}$: $\mathcal{U}(\widehat{x}) \simeq x$.

The Kolmogorov-Chaitin complexity of x is defined to be the length of the shortest program which generates x:

$$C(x) = |\widehat{x}| = \min\big(|p| \mid \mathcal{U}(p) \simeq x \big)$$

This concept was discovered independently by Solomonov 1960, Kolmogorov 1963 and Chaitin 1965.

Example

Let x be the first 35,014 binary digits of π. Then x has Kolmogorov-Chaitin complexity at most a 980 in the standard C model.

Note that we can always hard-wire a table into the program. It follows that $\widehat{x}$ and therefore $C(x)$ exists for all x. Informally, the program looks like

print "$x_1 x_2 \ldots x_n$"

Moreover, we have a simple bound:

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

But note that running an arbitrary program p on $\mathcal{U}$ may produce no output: the (simulation of the) program may simply fail to halt.

The claim that $C(x) \leq |x| + c$ is obvious in the C model.

But remember, we really need to deal with a universal Turing machine.

The program string there could have the form

$$p = u\, x \in \mathbf{2}^\star$$

where u is the instruction part ("print the following bits"), and x is the desired output.

So the machine actually only needs to erase u in this case. This produces a very interesting problem: how does $\mathcal{U}$ know where u ends and x starts?

We could use a simple coding scheme to distinguish between the program part and the data part of p:

$$p = 0u_1 0u_2 \ldots 0u_r \, 1 \, x_1 x_2 \ldots x_n$$

Obviously, $\mathcal{U}$ could now parse p just fine. This seems to inflate the complexity of the program part by a factor of 2, but that's OK; more on coding issues later.

There are other possibilities like $p = 0^{|u|} 1 \, u \, x$.

Also note: we can cheat and hardwire any specific string x of very high complexity in $\mathcal{U}$ into a modified environment $\mathcal{U}'$.

Let's say

- $\mathcal{U}'$ on input 0 outputs x.
- $\mathcal{U}'$ on input $1p$ runs program $\mathcal{U}(p)$.
- $\mathcal{U}'$ on input $0p$ returns no output.

Then $\mathcal{U}'$ is a perfectly good universal machine that produces good complexity measures, except for x, which gets the fraudulently low complexity of 1. Similarly we could cheat on a finite collection of strings $x_1, \ldots, x_n$.

Fortunately, beyond this minor cheating, the choice of $\mathcal{U}$ doesn't matter much. If we pick another machine $\mathcal{U}'$ and define K' accordingly, we have

$$K'(x) \leq C(x) + c$$

since $\mathcal{U}$ can simulate $\mathcal{U}'$ using some program of constant size. The constant c depends only on $\mathcal{U}$ and $\mathcal{U}'$.

This is actually the critical constraint in an axiomatic approach to KC complexity: we are looking for machines that cannot be beaten by any other machine, except for a constant factor. Without this robustness our definitions would be essentially useless.

It is even true that the additive offset c is typically not very large; something like a few thousand, not $A(100, 100)$.

What we would really like is a natural universal machine $\mathcal{U}$ that just runs the given programs, without any secret tables and other slimy tricks. Think about a real C compiler.

Alas, this notion of "natural" is quite hard to formalize.

One way to avoid cheating, is to insist that $\mathcal{U}$ be tiny: take the smallest universal machine known (for the given tape alphabet). This will drive up execution time, and the programs will likely be rather cryptic, but that is not really our concern.

Greg Chaitin has actually implemented such environments $\mathcal{U}$.

He uses LISP rather than C, but that's just a technical detail (actually, he has written his LISP interpreters in C).

So in some simple cases one can actually determine precisely how many bits are needed for $\widehat{x}$.

Proposition

For any positive integer x: $C(x) \leq \log x + c$.

This is just plain binary expansion: we can write x in

$$n = \lfloor \log_2 x \rfloor + 1$$

bits using standard binary notation.

But note that for some x the complexity $C(x)$ may be much smaller than $\log x$.

For example $x = 2^{2^k}$ or $x = 2^{2^{2^k}}$ requires far fewer than $\log x$ bits.

Exercise

Construct some other numbers with small Kolmogorov-Chaitin complexity.

How about duplicating a string? What is $C(xx)$?

In the C world, it is clear that we can construct a constant size program that will take as input a program for x and produce xx instead. Hence we suspect

$$C(xx) \leq C(x) + O(1).$$

Again, in the Turing machine model this takes a bit of work: we have to separate the program from the data part, and copying requires some kind of marking mechanism (not trivial, since our tape alphabet is fixed).

A very similar argument shows that

$$C(x^{\mathsf{op}}) \leq C(x) + O(1).$$

How about concatenation?

$$C(xy) \leq C(x) + C(y) + O(\log \min(C(x), C(y)))$$

Make sure to check this out in the Turing machine model. Note in particular that it is necessary to sandbox the programs for x and y.

Here is a more surprising fact: we can apply any computable function to x, and increase its complexity by only a constant.

Lemma

Let $f : \mathbf{2}^\star \to \mathbf{2}^\star$ *be computable.*

Then $C(f(x)) \leq C(x) + O(1)$.

Proof.

f is computable, hence has a finite description in terms of a Turing machine program q. Combine q with the program $\widehat{x}$.

□

The last lemma is a bit hard to swallow, but it's quite correct.

Take your favorite exceedingly-fast-growing recursive function, say, the Ackermann function $A(x, x)$.

As we have seen, $A(100, 100)$ is a mind-boggling atrocity; much, much larger than anything we can begin to make sense of.

And yet

$$C(A(100, 100)) \leq \log 100 + \text{ a little } = \text{ a little}$$

Exercise

Prove the complexity bound of a concatenation xy from above.

Exercise

Is it possible to cheat in infinitely many cases? Justify your answer.

Exercise

Use Kolmogorov-Chaitin complexity to show that the language $L = \{ x\, x^{\mathsf{op}} \mid x \in \mathbf{2}^\star \}$ of even length palindromes cannot be accepted by an finite state machine.

Suppose we have a string $x = 0^n$.

In some sense, x is trivial, but $C(x)$ may still be high, simply because $C(n)$ is high.

Definition

Let $x, y \in \mathbf{2}^\star$. The conditional Kolmogorov complexity of x given y is the length of the shortest program p such that $\mathcal{U}$ with input p and y computes x.

Notation: $C(x \,|\, y)$.

Then $C(0^n \,|\, n) = O(1)$, no matter what n is.

And $C(x \,|\, \widehat{x}) = O(1)$.

Lemma

$$C(xy) \leq C(x) + C(y \mid x) + O(\log \min(C(x), C(y)))$$

Proof.

Once we have x, we can try to exploit it in the computation of y.

The log factor in the end comes from the need to separate the shortest programs for x and y.

□

$C(x)/|x|$ is the ultimate compression ratio: there is no way we can express x as anything shorter than $C(x)$ (at least in general; recall the comment about cheating).

An algorithm that takes as input x and returns as output $\widehat{x}$ is the dream of anyone trying to improve `gzip` or `bzip2`.

Well, almost. In a real compression algorithm, the time to compute $\widehat{x}$ and to get back from there to x is also very important. In our setting time complexity is being ignored completely.

As we will see, there is also the slight problem that $C(x)$ is not computable, much less $\widehat{x}$.

As is the case with compression algorithms, even C cannot always succeed in producing a shorter string.

Definition

A string $x \in \mathbf{2}^\star$ is c-incompressible if $C(x) \geq |x| - c$ where $c \geq 0$.

Hence if x is c-incompressible we can only shave off at most c bits when trying to write x in a more compact form: an incompressible string is generic, it has no special properties that one could exploit for compression.

The upside is that we can adopt incompressibility as a definition of randomness for a finite string – though it takes a bit of work to verify that this definition really conforms with our intuition. For example, such a string cannot be too biased.

Having incompressible strings can be very useful in lower bound arguments: there is no way an algorithm could come up with a clever, small data structure that represents these strings.

How do we know that incompressible strings exist? By high school counting: there aren't enough short programs to generate all long strings. Here is a striking result whose proof is also a simple counting argument.

Lemma

Let $S \subseteq \mathbf{2}^\star$ be set of words of cardinality $n \geq 1$. For all $c \geq 0$ there are at least $n(1 - 2^{-c}) + 1$ many words x in S such that

$$C(x) \geq \log n - c.$$

Example

Consider $S = \mathbf{2}^k$ so that $n = 2^k$. Then, by the lemma, most words of length k have complexity at least $k - c$, so they are c-incompressible.

In particular, there is at least one string of length k with complexity at least k.

Example

Pick size s and let $S = \{\, 0^i \mid 0 \leq i < s \,\}$. Specifying $x \in S$ comes down to specifying the length $|x|$. Writing a program to output the length will often require close to $\log s$ bits.

This lemma sounds utterly wrong: why not simply put only simple words (of low Kolmogorov-Chaitin complexity) into S? There is no restriction on the elements of S, just its size.

Since we are dealing with strings, there is a natural, easily computable order: length-lex. Hence there is an enumeration of S:

$$S = w_1, w_2, \ldots, w_{n-1}, w_n$$

Given the enumeration, we need only some $\log n$ bits to specify a particular element. The lemma says that for most elements of S we cannot get away with much less.

Exercise

Try to come up with a few "counterexamples" to the lemma and understand why they fail.

Proof is by very straightforward counting. Let's ignore floors and ceilings.

The number of programs of length less than $\log n - c$ is bounded by

$$2^{\log n - c} - 1 = n2^{-c} - 1.$$

Hence at least

$$n - (n2^{-c} - 1) = n(1 - 2^{-c}) + 1$$

strings in S have complexity at least $\log n - c$.

□

It gets worse: the argument would not change even if we gave the program p access to a database $D \in \mathbf{2}^\star$ as in conditional complexity.

This observation is totally amazing: we could concatenate all the words in S into a single string

$$D = w_1 \dots w_s$$

that is accessible to p.

However, to extract a single string w_i, we still need some $\log s$ bits to describe the first and last position of w_i in D.

A similar counting argument shows that all sufficiently long strings have large complexity:

Lemma

The function $x \mapsto C(x)$ is unbounded.

Actually, even $x \mapsto \min\big(C(z) \mid x \leq_{ll} z \big)$ is unbounded (and monotonic).

Here $x \leq_{ll} z$ refers to length-lex order.

So even a trivial string $000\ldots000$ has high complexity if it's just long enough. Of course, the conditional complexity $C(0^n \mid n)$ is small.

As mentioned, it may happen that $\mathcal{U}(p)$ is undefined simply because the simulation of program p never halts. And, since the Halting Problem is undecidable, there is no systematic way of checking:

Problem:	**Halting Problem for $\mathcal{U}$**
Instance:	Some program $p \in \mathbf{2}^\star$.
Question:	Does p (when executed on $\mathcal{U}$) halt?

Of course, this version of Halting is still semidecidable, but that's all we can hope for.

Theorem

The function $x \mapsto C(x)$ is not computable.

Proof. Suppose otherwise. Consider the following algorithm $\mathcal{A}$ with input n, where the loop is supposed to be in length-lex order.

foreach $x \in \mathbf{2}^\star$ **do**
    let $m = C(x)$;
    if $n \leq m$ **then return** x;

Then $\mathcal{A}$ halts on all inputs n, and returns the length-lex minimal word x of Kolmogorov complexity at least n. But then

$$n \leq C(x) \leq C(n) + c \leq \log n + c',$$

contradiction. □

Let's try to pin down the problem with computing Kolmogorov-Chaitin complexity.

- Given a string x of length n, we would look at all programs $p_1, \ldots, p_N$ of length at most $n + c$.
- We run all these programs on $\mathcal{U}$, in parallel.
- At least one of them, say, p_i, must halt on output x.
- Hence $C(x) \leq |p_i|$.

But unfortunately, this is just an upper bound: later on a shorter program p_j might also output x, leading to a better bound.

But other programs will still be running; as long as at least one program is still running we only have a computable approximation, but we don't know whether it is the actual value.

Consider the following variant of the Halting set K_0, and define the Kolmogorov set K_1:

$$K_0 = \{\, e \mid \{e\}() \downarrow \}$$

$$K_1 = \{\, (x, n) \mid C(x) = n \,\}$$

Theorem

K_0 and K_1 are Turing equivalent.

Proof.

We have just seen that K_1 is K_0-decidable.

This is harder, much harder.

Let $n = |M_e|$ where the Turing machine is encoded in binary.

Use oracle K_1 to filter out the set $S = \{ z \in \mathbf{2}^{2n} \mid C(z) < 2n \}$.

Determine the time τ when all the corresponding programs $\widehat{z}$ halt.

Claim: $\{e\}() \downarrow$ iff $\{e\}_\tau() \downarrow$

Assume otherwise, so $\{e\}() \downarrow$ but $\{e\}_\tau() \uparrow$.

Use M_e as a clock to determine $t > \tau$ such that $\{e\}_t() \downarrow$.

But then we can run all programs of size at most $2n - 1$ for t steps and obtain S, and thus a string $z' \in \mathbf{2}^{2n}$ of complexity at least $2n$.

Alas, the computation shows that $C(z') \leq n$, contradiction.

□

If you don't like oracles, we can also represent $C(x)$ as the limit of a computable function:

$$C(x) = \lim_{\sigma \to \infty} D(x, \sigma)$$

where $D(x, \sigma)$ is the length of the shortest program $p < \sigma$ that generates output x in at most σ steps, σ otherwise. So D is even primitive recursive.

Note that $D(x, \sigma)$ is decreasing in the second argument.

As a consequence, $C(x)$ is a Σ_2 function, just on the other side of computability.

Theorem (Barzdin 1968)

Let A be any recursively enumerable set and denote its characteristic sequence by χ. Then $C(\chi[n] \mid n) \leq \log n + c$.

Proof.

Dovetail the computation of the machine accepting A on inputs less than n.

Terminate as soon as the number of convergent computations is $|A \cap [0, n-1]|$, a number that can be specified in $\log n$ bits.

□

Note, though, that the dovetailing make take more steps than any recursive function in n. E.g., let $A = \emptyset'$ be the jump.

- Straight Line Programs
- Program-Size Complexity

3. Prefix Complexity

- Incompleteness

Kolmogorov-Chaitin algorithmic information theory provides a measure for the "complexity" of a bit string (or any other finite object). This is in contrast to language based models that only differentiate between infinite collections.

Since the definition is closely connected to Halting, the complexity function $C(x)$ fails to be computable, but it provides an elegant theoretical tool and can be used in lower bound arguments.

And it absolutely critical in the context of randomness; more later.

Recall that our model of computation used in Kolmogorov-Chaitin complexity is a universal, one-tape Turing machine over the tape alphabet $\Gamma = \{0, 1, b\}$, with binary input and output.

This causes a number of problems because it is difficult to decode an input string of the form

$$p = q\,z$$

into an instruction part q and a data part z (run program q on input z).

Of course, this kind of problem would not surface if we used real programs instead of just binary strings. We should try to eliminate it in our setting, too.

M. Li, P. Vitányi

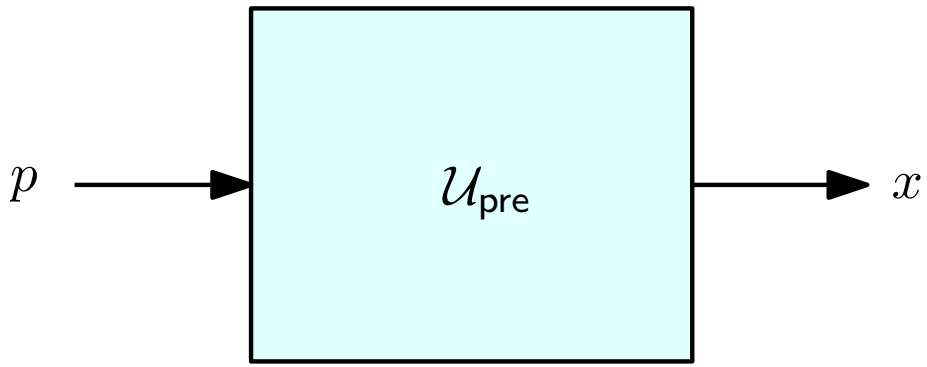
An Introduction to Kolmogorov Complexity and its Applications

Springer, 1993

Encyclopedic treatment.

But note that some things don't quite type-check ($\mathbb{N}$ versus $\mathbf{2}^\star$).

The key idea is to restrict our universal machine a little bit.



We require that $P \subseteq \mathbf{2}^\star$, the collection of all syntactically correct programs for $\mathcal{U}_{\text{pre}}$, is a prefix set: no valid program is a prefix of another.

Note that this condition trivially holds for most ordinary programming languages (at least in spirit).

Throughout we only consider Turing machines with binary input and output (plus a blank symbol). Call a Turing machine M prefix if its halting set $\{\, p \in \mathbf{2}^\star \mid M(p)\downarrow \,\}$ is prefix.

Note that simulation is particularly simple for prefix machines: to simulate M on M' we can set up a header h such that

$$M'(hp) \simeq M(p)$$

for all M-admissible programs p.

Lemma

For any Turing machine M, we can effectively construct a prefix Turing machine M' such that $\forall\, p \in \mathbf{2}^\star \left(M'(p)\downarrow \Rightarrow M(p) \simeq M'(p)\right)$ *and*

$$M \text{ prefix } \Rightarrow \forall\, p \in \mathbf{2}^\star \left(M(p) \simeq M'(p)\right)$$

Of course, in general M' will halt on fewer inputs and the two machines are by no means equivalent (just think what happens to a machine with domain $0^\star$).

Suppose we have an ordinary machine M and some input $p \in \mathbf{2}^\star$. M' computes on p as follows:

- Enumerate the domain of M in some sequence $(q_i)_{i \geq 0}$.
- If $q_i = p$, return $M(p)$.
- If q_i is a proper prefix of p, or conversely, diverge.

It is easy to check that M' is prefix and will define the same function as M, provided M itself is already prefix.

As a consequence, we can enumerate all prefix functions $\{e\}_{\mathsf{pre}}$ just as we can enumerate ordinary computable functions.

The idea of a universal machine that only converges on a prefix set is perfectly well motivated in the world of programming languages, but how about an actual Turing machine?

No problem, we can define $\mathcal{U}'$ so that it checks for inputs of the form

$$p = 0u_1 0u_2 \dots 0u_r\, 1$$

If the input has the right form, $\mathcal{U}'$ computes $\mathcal{U}(u_1 \dots u_r)$.

Otherwise it simply diverges.

Definition

Let $\mathcal{U}_{\mathrm{pre}}$ be a universal prefix Turing machine. Define the prefix Kolmogorov-Chaitin complexity of a string x by

$$K(x) = \min\big(|p| \mid \mathcal{U}_{\mathrm{pre}}(p) \simeq x \big)$$

Note that in general $K(x) > C(x)$: there are fewer programs available, so in general the shortest program for a fixed string will be longer than in the unconstrained case.

Of course, $K(x)$ is again not computable.

The following mutual bounds are due to Solovay:

$$K(x) \leq C(x) + C(C(x)) + O(C(C(C(x))))$$

$$C(x) \leq K(x) - K(K(x)) - O(K(K(K(x))))$$

This pins down the cost of dealing with prefix programs as opposed to arbitrary ones.

Recall that for ordinary Kolmogorov-Chaitin complexity it is easy to get an upper bound for $C(x)$: the program

print "$x_1 x_2 \ldots x_n$"

does the job.

But if we have to deal with the prefix condition, things become a bit more complicated. We could use delimiters around x, but remember that our input and output alphabet is fixed to be $\mathbf{2} = \{0, 1\}$.

We could add symbols, but that does not solve the problem.

In the absence of delimiters, we can return to our old idea of self-delimiting programs. Informally, we could write

print next n **bits** $x_1 x_2 \ldots x_n$

In pseudo-code this is fine, but in our Turing machine model we have to code everything as bits. For this to work we need to be able to distinguish the instruction bits from the bits for n.

Alas, coding details are essential to produce prefix programs and to obtain a bound on $K(x)$.

Here is a simple way to satisfy the prefix condition: code a bit string x as

$$E(x_1 \dots x_n) = 0x_1\, 0x_2\, \dots\, 0x_{n-1}\, 1x_n$$

so that $|E(x)| = 2|x|$.

Of course, there are other obvious solutions such as $0^{|x|}1x$.

Both approaches double the length of the string, which doubling would lead to a rather crude upper bound $2n + O(1)$ for the prefix complexity of a string via the program

print $E(x)$

Can we do better?

How about leaving $x = x_2x_2 \dots x_n$ unchanged, but using E to code $n = |x|$, the length of x:

$$E(|x|)\, x$$

Note that this still is a prefix code and we now only use some $2 \log n + n$ bits to code x.

But why stop here? We can also use

$$E(||x||)\, |x|\, x$$

This requires only some $2 \log \log n + \log n + n$ bits.

In fact, we can iterate this coding operation. Let

$$E_0 := E$$
$$E_{i+1}(x) := E_i(|x|)\, x$$

It is not hard to show that all the E_i are prefix codes.

But there is still a little problem: what is the optimal choice of k so that $E_k(x)$ has minimal length? Clearly k depends on the length of x.

We can handle this nicely by defining an "infinity code" E_∞ that works for all x.

Here it is:

$$E_\infty(x) = \mathrm{len}_k(x)\,0\,\mathrm{len}_{k-1}(x)\,0\,\ldots\,|x|\,0\,x\,1$$

where $k = \mathrm{len}^*(x)$ is just a discrete version of an iterated logarithm:

$$\begin{aligned} \mathrm{len}_1(x) &= |x| \\ \mathrm{len}_{i+1}(x) &= |\mathrm{len}_i(x)| \\ \mathrm{len}^*(x) &= \min\big(\, i \mid \mathrm{len}_i(x) = 2 \,\big) \end{aligned}$$

Example

For a bit-string x of length 20000 we obtain the following $\mathrm{len}_i(x)$:

$$100111000100000, 1111, 100, 11, 10$$

So the length of $E_\infty(x)$ is $20000 + 32$.

How much do we have to pay for a prefix version of x? Essentially a sum of iterated logs.

Lemma

$$|E_\infty(x)| = n + \log n + \log\log n + \log\log\log n \ldots + \log^*(n) + O(1)$$

So this is an upper bound on $K(x)$.

Of course, some other coding scheme might produce even better results.

A good rough approximation to $K(x)$ is $n + \log n$, in perfect keeping with our intuition about

print next n **bits** $x_1 x_2 \ldots x_n$

It's clear that prefix complexity is a bit harder to deal with than ordinary Kolmogorov-Chaitin complexity. What are the payoffs?

For one thing, it is much easier to combine programs. This is useful e.g. for concatenation.

Suppose we have prefix programs p and q that produce x and y, respectively.

But then pq is uniquely parsable, and we can easily find a header program h such that

$$h\,p\,q$$

is an admissible program for $\mathcal{U}_{\text{pre}}$ that executes p and q to obtain xy.

Thus

$$K(xy) \leq K(x) + K(y) + O(1)$$

Define $K(x, y)$ to be the length of the shortest program that writes $x\,b\,y$ on the tape (recall that our tape alphabet is $\{0, 1, b\}$).

Note that $K(xy) \leq K(x, y) + O(1)$, but the opposite direction is tricky (think about $x, y \in 0^\star$).

At any rate, the last argument shows that $K()$ is subadditive:

$$K(x, y) \leq K(x) + K(y) + O(1)$$

This simply fails for plain KC complexity.

From a more axiomatic point of view, plain KC complexity is slightly deficient in several ways:

- Not subadditive: $C(x, y) \leq C(x) + C(x) + c$.
- Not prefix monotonic: $C(x) \leq C(xy) + c$.
- Plain KC complexity does not help much when applied to the problem of infinite random sequences.

Many arguments still work out fine, but there is a sense that the theory could be improved.

Here is the killer app for prefix complexity.

Definition

The total halting probability of any prefix program is defined to be

$$\Omega = \sum_{\mathcal{U}_{\mathrm{pre}}(p)\downarrow} 2^{-|p|}$$

Ignoring the motivation behind this for a moment, note that this definition works because of the following bound.

Lemma (Kraft Inequality)

Let $S \subseteq \mathbf{2}^\star$ be a prefix set. Then $\sum_{x \in S} 2^{-|x|} \leq 1$.

We can define the halting probability for a single target string x to be

$$P(x) = \sum_{\mathcal{U}_{\mathrm{pre}}(p) \simeq x} 2^{-|p|}.$$

and extend this to sets of strings by adding: $P(S) = \sum_{x \in S} P(x)$.

Then $\Omega = P(\mathbf{2}^\star)$. Ω depends quite heavily on $\mathcal{U}_{\mathrm{pre}}$, so one could write $\Omega(\mathcal{U}_{\mathrm{pre}})$ or some such for emphasis.

Proposition

Ω is a real number and $0 < \Omega < 1$.

In fact, for one particular $\mathcal{U}_{\mathrm{pre}}$, one can show with quite some pain that

$$0.00106502 < \Omega(\mathcal{U}_{\mathrm{pre}}) < 0.217643$$

Proposition

Ω is incompressible in the sense that $K(\Omega[n]) \geq n - c$, for all n.

As a consequence, (the binary expansion of) Ω is Martin-Löf random.

This may seem a bit odd since we have a perfectly good definition of Ω in terms of a converging infinite series. But note the Halting Problem lurking in the summation – from a strictly constructivist point of view Ω is in fact quite poorly defined.

Lemma

Consider $q \in \mathbf{2}^n$. Given $\Omega[n]$, it is decidable whether $\mathcal{U}_{\mathrm{pre}}$ halts on input q.

Proof.

Start with approximation $\Omega' = 0$.

Dovetail computations of $\mathcal{U}_{\mathrm{pre}}$ on all inputs.

Whenever convergence occurs on input p, update the approximation: $\Omega' = \Omega' + 2^{-|p|}$.

Stop as soon as $\Omega' \geq \Omega[n]$. Then

$$\Omega[n] \leq \Omega' < \Omega < \Omega[n] + 2^{-n}.$$

But then no program of length n can converge at any later stage. □

For $n \approx 10000$, knowledge of $\Omega[n]$ would settle, at least in principle, several major open problems in Mathematics such as the Goldbach Conjecture or the Riemann Hypothesis:

These conjectures can be refuted by an unbounded search, and the corresponding Turing machine can be coded in 10000 bits.

For example, here is the Goldbach conjecture:

Conjecture: Every even number larger than 2 can be written as the sum of two primes.

We can easily construct a small Turing machine that will search for a counterexample to this conjecture, and will halt if, and only if, the Goldbach conjecture is false.

Of course, we don't have the first 10000 bits of Ω, nor will we ever.

In fact, things are much, much worse than that.

Suppose some demon gave you these bits. It would take a long time to exploit this information: the running time of the oracle algorithm above is not bounded by any recursive function of n.

The answers would be staring at us, but we could not pull them out.

- Straight Line Programs
- Program-Size Complexity
- Prefix Complexity

4. Incompleteness

David Hilbert wanted to crown 2000+ years of development in math by constructing an axiomatic system that is

- consistent
- complete
- decidable

Alas . . .

Theorem (Gödel 1931)

Every consistent reasonable theory of mathematics is incomplete.

Theorem (Turing 1936)

Every consistent reasonable theory of mathematics is undecidable.

Good news for anyone interested in foundations, who would want to live in a boring world?

Gödel's argument is a very careful elaboration and formalization of the old liar's paradox:

> This here sentence is false.

Turing uses classical Cantor-style diagonalization applied to computable reals.

Both arguments are perfectly correct, but they seem a bit ephemeral; they don't quite have the devastating bite one might expect.

Ω can help to make the limitations of the formalist/axiomatic approach much more concrete. First a warm-up.

Émile Borel defined a normal number in base B to be a real r with the property that all digits in the base B expansion of r appear with limiting frequency $1/B$.

Theorem (Borel)

With probability 1, a randomly chosen real is normal in any base.

Alright, but how about concrete examples? It seems that $\sqrt{2}$, π and e are normal (billions of digits have been computed), but no one currently has a proof.

$$C = 0.12345678910111213141516171819202122\ldots$$

Champernowne showed that this number is normal in base 10 (and powers thereof), the proof is not difficult.

Proposition

Ω is normal in any base.

Of course, there is a trade-off: we don't know much about the individual digits of Ω.

Time to get serious. Fix some n. Suppose we want to prove all correct theorems of the form

$$K(x) \geq n \qquad K(x) = m$$

where $m < n$ and $x \in \mathbf{2}^\star$.

How much information would we need to do this?

All we need is the maximal halting time τ of all programs of length at most $n - 1$. It is not hard to see that

$$K(\tau) = n + O(1)$$

Essentially nothing less will do.

In the following we assume that $\mathcal{T}$ is some axiomatic theory of mathematics that includes arithmetic.

Think of $\mathcal{T}$ as Peano Arithmetic, though stronger systems such as Zermelo-Fraenkel with Choice is perfectly fine, too (some technical details get a bit more complicated; we have to interpret arithmetic within the stronger theory).

Assertions like

$$K(x) \geq n \qquad K(x) = m$$

can certainly be formalized in $\mathcal{T}$ and we can try to determine how easily these might be provable in $\mathcal{T}$.

We need $\mathcal{T}$ to be consistent: it must not prove wrong assertions. This is strictly analogous to the situation in Gödel's theorem: inconsistent theories have no trouble proving anything.

Technically, all we need is Σ_1 consistency: any theorem of the following form, provable in $\mathcal{T}$, must be true:

$$\exists\, x\, \varphi(x)$$

where φ is "primitive recursive" (defines a primitive recursive property in $\mathcal{T}$) and the existential quantifier is arithmetic.

We assume that certain rules of inference are fixed, once and for all. So the theory $\mathcal{T}$ comes down to its set of axioms.

If there only finitely many, we can think of them as a single string and define $K(\mathcal{T})$ accordingly.

If there are infinitely many axioms (as in PA), the set of all axioms is still decidable and we can define $K(\mathcal{T})$ as the complexity of the corresponding decision algorithm.

Note that this approach totally clobbers anything resembling semantics: it does not matter how clever the axioms are, just how large a data structure is needed to specify them.

Theorem (Chaitin 1974/75)

If $\mathcal{T}$ proves the assertion $K(x) \geq n$, then $n \leq K(\mathcal{T}) + O(1)$.

Proof.

Enumerate all theorems of $\mathcal{T}$, looking for statements $K(x) \geq n$.

For any $m \geq 0$, let x_m be the first string so discovered where $n > K(\mathcal{T}) + m$.

By consistency, we have

$$K(\mathcal{T}) + m < K(x_m)$$

By construction,

$$\begin{aligned} K(x_m) &\leq K(\mathcal{T}, K(\mathcal{T}), m) + O(1) \\ &\leq K(\mathcal{T}) + K(m) + O(1) \end{aligned}$$

This is one place where subadditivity is critical.

But then it follows that

$$m < K(m) + O(1)$$

and thus $m \leq m_0$ for some fixed m_0.

□

Similarly one can prove that no consistent theory can determine more than

$$K(\mathcal{T}) + O(1)$$

bits of Ω.

We have a perfectly well-defined real, but we can only figure out a few of its digits.

One can sharpen Chaitin's theorem to a point where it almost seems absurd:

Theorem

Let $\mathcal{T}$ be as before. Then there is a universal prefix machine $\mathcal{U}_{\mathrm{pre}}$ such that

- *Peano Arithmetic proves that $\mathcal{U}_{\mathrm{pre}}$ is indeed universal.*
- *$\mathcal{T}$ cannot determine a single digit of Ω.*

Of course, the Ω in question here is $\Omega(\mathcal{U}_{\mathrm{pre}})$.

The proof depends on a very clever construction of a particular universal prefix machine and uses Kleene's recursion theorem.