UCT

Program Size Complexity

Klaus Sutner

Carnegie Mellon University

Spring 2022

Wurzelbrunft's Problem

Prof. Dr. Alois Wurzelbrunft has discovered a special type of algebraic structure, so-called Wurzelbrunft algebras. W-algebras come in two types, tame or wild. In an heroic intellectual effort, Wurzelbrunft has shown that there are exactly 42 W-algebras. Unfortunately, it seems quite difficult to figure out which of them are tame.

> Wurzelbrunft wonders whether he should try to prove that tameness of W-algebras is undecidable.

If only he had taken a course in complexity theory ...

The Misery of Finite Problems

Any decision problem with finitely many instances is automatically decidable, albeit for entirely the wrong reasons.

To wit, we can hardwire the answers:

x_1	x_2	x_3	$\dots$	x_{n-1}	x_n
b_1	b_2	b_3	$\dots$	b_{n-1}	b_n

Here b_i is a bit that encodes the answer.

The problem is that the correct bit-vector $b_1, b_2, \dots, b_n$ exists, basta.

Alas, we may not know what it is. We know a decision algorithm exists, but we cannot produce it.

Same Old

We have talked about this issue before, in the context of showing that

> A set is decidable
> iff
> its principal function is computable.

Right-to-left runs into a problem: we cannot tell from a program for the principal function whether its support is finite.

The decision algorithm for the set critically depends on this bit of information that we cannot determine effectively.

Still, the algorithm always exists ...

Algorithm for Riemann Hypothesis

We can push this to absurd levels: the RH Problem has only one instance (a banana), and it's a yes-instance iff the RH is true. This problem is decidable.

Algorithm I: return Yes

Algorithm II: return No

One of those two algorithms works. Done.

Of course, this is absolute rubbish, our framework of computability simply does not work in the context of problems with finitely many instances.

To Be Or Not To Be

Pure existence in the standard, non-constructive sense is a bit thin, we would like to have some method to determine the bits, to actually construct the answers.

For example, consider only integer polynomials with at most

100 variables, degree 10^{100}, coefficients below 10^{100}

We would very much like to have an algorithm that determines which of these finitely many polynomials have integral roots.

There is no hope for this, none whatsoever. Matiyasevic casts a huge shadow.

How Complicated Is A Bit-Vector?

One way to tackle this issue is to try to come up with some measure to the complexity (in the intuitive sense) of finite bit-vectors.

A priori, computation and complexity theory seem to be utterly useless here, everything is trivial in this framework.

For example, for any string $x \in \mathbf{2}^n$, we can build a finite state machine $\mathcal{A}$ that accepts only this string.

Of course, the machine is essentially just the string itself . . .

64 Bits

$0101010101010101010101010101010101010101010101010101010101010101$

$0101101110111101111101111110111111101111111101111111110111111111$

$1011010100000100111100110011001111111001110111100110010010000100$

$0011100101100001011001010100001110011010111111001010000110010011$

Which is the least/most complicated?

More Bits?

A good way to think about this is to try to predict "future bits" in the sequence, assuming there is somehow a natural way to extend it (maybe to an infinite string). Yes, that's not even ill-defined. Still . . .

- $(01)^\omega$
- 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. Kiss ZFC goodbye.

A π Program

How about writing a program that generates the finite string in question?

```
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;
}
```

This program compiles (with a few warnings) and running it produces the first 10000 decimal digits of π.

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

Program-Size Complexity

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 in $\mathbf{2}^\star$.

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

Short Programs

In the actual theory, one uses universal Turing machines to formalize the notion of a program and its execution. But, as we have seen many times, Turing machines are bit unwieldy, so for intuition it is better to think of

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

Why C? Because it is a no-BS language, close to actual hardware.

So, informally we are interested in the shortest C program that will produce same particular target output. As the π example shows, these programs might be rather weird (in fact, really short programs often are bizarre).

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

Background

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).

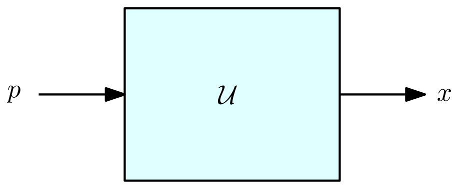
The machine has a single tape for input/work/output.

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

$$\mathcal{U}(p) \simeq x$$

The Picture



Kolmogorov-Chaitin Complexity

Definition

For any word $x \in \mathbf{2}^\star$, 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:

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

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

Think of $\widehat{x}$ as the ultimate compressed form of x, the shortest possible description available (at least in the particular environment $\mathcal{U}$).

The Basics

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

print “$x_1 x_2 \ldots x_n$”

No problem. Moreover, we have a simple bound, there is a constant c such that for any string x whatsoever

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

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. Of course, non-halting programs are useless as far as Kolmogorov-Chaitin complexity is concerned.

Hold It . . .

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

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

The program string $p = \widehat{x} \in \mathbf{2}^\star$ here 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? After all, everything is just a bunch of 0s and 1s . . .

Self-Delimiting Programs

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. Alas, this seems to inflate the complexity of the program part by a factor of 2. We'll have more to say about coding issues later.

Note that there are other simple possibilities like $p = 0^{|u|} 1 \, u \, x$.

Cheating

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

Let's say

- $\mathcal{U}'$ on input 0 outputs χ.
- $\mathcal{U}'$ 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 χ, which gets the fraudulently low complexity of 1. Similarly we could cheat on a finite collection of strings $\chi_1, \ldots, \chi_n$.

Invariance

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

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

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.

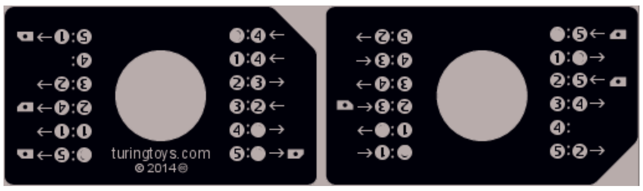
Avoiding Cheaters

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.

Small Universal



A small universal machine like the one above (4 states, 6 tape symbols) would seem to be free of any kind of treachery.

Concrete $\mathcal{U}$

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 $\widehat{x}$.

Numbers

Proposition

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

This is just plain binary expansion: we can write $x > 0$ in

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

bits using standard binary notation.

But note that for some x the complexity $K(x)$ 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.

Copy

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

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

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

Again, in the Turing machine model this takes a bit of work: we have a program p that generates x. To build a program for xx we could run p, then copy the output. Alternatively, we could try to run p twice and put the output right next to the first run. Neither method is trivial, since our tape alphabet is fixed. Just try it.

String Operations

A very similar argument shows that

$$K(x^{\text{op}}) \leq K(x) + O(1).$$

Concatenation is slightly more complicated: we have to be able to determine the parts of the program for xy that corresponds to $\widehat{x}$ and $\widehat{y}$.

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

Say, $n = K(x) \leq K(y)$. The we write down $n\,\widehat{x}\,\widehat{y}$ where n is in binary and self-delimiting.

Computable String Operations

Here is a slightly counterintuitive 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 $K(f(x)) \leq K(x) + O(1)$.

Proof.

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

□

Say What?

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

Take your favorite exceedingly-fast-growing recursive function, say, Friedman's mindnumbing α function. Recall that the algorithm for α is actually quite simple.

But $\alpha(3)$ is a mind-boggling atrocity; a number much, much larger than anything we can begin to make sense of. The kind of monster that only exists in recursion theory, not in any other branch of mathematics.

And yet

$$K(\alpha(3)) \leq \log 3 + \text{ a little } = \text{ a little}$$

Some Exercises

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 $L = \{\, x\,x^{\text{op}} \mid x \in \mathbf{2}^\star \,\}$ of even length palindromes cannot be accepted by an finite state machine.

Conditional Complexity

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

In some sense, x is trivial, but $K(x)$ may still be high, simply because $K(n)$ is high: printing 0s is trivial, but we need to know how many.

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: $K(x \mid y)$.

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

And $K(x \mid \widehat{x}) = O(1)$.

The Chain Rule

Lemma

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

Proof.

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

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

□

Note that we can also do $K(y) + K(x \mid y) + \ldots$.

Compression

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

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/space to compute $\widehat{x}$ and to get back from there to x is also critically important. In our setting, time and space complexity are being ignored completely.

Alas, there is also the slight problem that neither $K(x)$ nor $\widehat{x}$ is computable.

Incompressibility

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 $K(x) \geq |x| - c$ where $c \geq 0$.
x is incompressible if it is 0-incompressible.

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.

Kolmogorov Random

A string $x \in \mathbf{2}^\star$ is Kolmogorov-random if $K(x) \geq |x|$.

So Kolmogorov-random means 0-incompressible.

Claim

There are Kolmogorov-random strings of all lengths.

This is a straightforward application of pigeon hole.

Existence

Having incompressible/random 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.

In general, what can we say about c-incompressible strings? Here is a striking result whose proof is based on simple counting.

Lemma

Let $S \subseteq \mathbf{2}^\star$ be a 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

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

Examples

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 Kolmogorov-random string of length k.

Example

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

But is it True?

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.

The Proof

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 $\log n - c$.

$\square$

Observation

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

$$D = w_1 \ldots w_n$$

that is accessible to p as on oracle.

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

Unbounded Complexity

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

Lemma

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

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

Here $x \leq_{\ell\ell} z$ 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 $K(0^n \mid n)$ is still small, it's the n that causes all the problems.

Halting

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.

Computation versus Kolmogorov-Chaitin

Let's try to understand intuitively why Kolmogorov-Chaitin complexity must be non-computable.

- Given a string x of length n, we would look at all programs $p_1, \ldots, p_N$ of length at most $n + c$ where c is the right constant to deal with the "print" statement.
- We run all these programs on $\mathcal{U}$, in parallel.
- At least one of them, say, p_i, must halt on output x.
- Hence $K(x) \leq |p_i|$.

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.

Non-Computability Proof

Theorem

The function $x \mapsto K(x)$ 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.

```
read $n$
foreach $x \in \mathbf{2}^\star$ do
    let $m = K(x)$
    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 for some constants c and c'

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

contradiction. □

The Connection

Consider the following variant of the Halting set H (for empty tape), and define the Kolmogorov set H', the graph of $K(.)$:

$$H = \{\, e \mid \{e\}() \downarrow \,\}$$
$$H' = \{\, x\#n \mid K(x) = n \,\}$$

Theorem

H and H' are Turing equivalent.

Proof.

We have just seen that H' is H-decidable.

Opposite Direction

This is harder.

We have H' as oracle, and we need to decide whether Turing machine $\mathcal{M}_e$ halts on empty tape.

Let $n = |e|$ assuming a binary index e.

Use oracle H' to filter out the set of compressible strings

$$S = \{\, z \in \mathbf{2}^{2n} \mid K(z) < 2n \,\}$$

Let τ be the time when all the corresponding programs $\widehat{z}$ halt. Since we know $K(z) < 2n$, we can simply run all programs of length less than $2n$ until the right one terminates, to determine τ.

Solving Halting

Here is the key fact: τ is large enough to resolve the Halting question for $\mathcal{M}_e()$.

Claim: $\mathcal{M}_e() \downarrow$ iff $\mathcal{M}_{e,\tau}() \downarrow$

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

Use $\mathcal{M}_e$ as a clock to determine $t > \tau$ such that $\mathcal{M}_{e,t}() \downarrow$.

But then we can run all programs of size less than $2n$ for t steps and obtain S, and thus an incompressible string $z' \in \mathbf{2}^{2n} - S$.

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

□

Limits

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

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

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 once $D(x, \sigma) < \sigma$ (we have found the first program that produces x), the function is decreasing in the second argument.

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