

Plain stopping time and conditional complexities revisited

Mikhail Andreev


IPONWEB, Berlin
amishaa@mail.ru

Gleb Posobin¹

National Research University Higher School of Economics, Moscow
posobin@gmail.com

Alexander Shen²

LIRMM CNRS / University of Montpellier, France. On leave from IITP RAS, Moscow
alexander.shen@lirmm.fr

 <https://orcid.org/0000-0001-8605-7734>

Abstract

In this paper we analyze the notion of “stopping time complexity”, the amount of information needed to specify when to stop while reading an infinite sequence. This notion was introduced by Vovk and Pavlovic [9]. It turns out that plain stopping time complexity of a binary string x could be equivalently defined as (a) the minimal plain complexity of a Turing machine that stops after reading x on a one-directional input tape; (b) the minimal plain complexity of an algorithm that enumerates a prefix-free set containing x ; (c) the conditional complexity $C(x|x^*)$ where x in the condition is understood as a prefix of an infinite binary sequence while the first x is understood as a terminated binary string; (d) as a minimal upper semicomputable function K such that each binary sequence has at most 2^n prefixes z such that $K(z) < n$; (e) as $\max C^X(x)$ where $C^X(z)$ is plain Kolmogorov complexity of z relative to oracle X and the maximum is taken over all extensions X of x .

We also show that some of these equivalent definitions become non-equivalent in the more general setting where the condition y and the object x may differ, and answer an open question from Chernov, Hutter and Schmidhuber [3].

2012 ACM Subject Classification Mathematics of computing → Information theory

Keywords and phrases Kolmogorov complexity, stopping time complexity, structured conditional complexity, algorithmic information theory

Digital Object Identifier 10.4230/LIPIcs.MFCS.2018.2

Related Version <https://arxiv.org/abs/1708.08100>

Funding Supported by RaCAF ANR-15-CE40-0016-01 RaCAF grant

Acknowledgements The authors are grateful to Alexey Chernov, Volodya Vovk, members of the ESCAPE team (LIRMM, Montpellier), Kolmogorov seminar (Moscow) and Theoretical Computer Science Laboratory (National Research University Higher School of Economics, Computer Science department, Moscow), the participants of Dagstuhl meeting where some results of the paper were presented [2], and the reviewers of preliminary versions of this paper.

¹ Supported by Russian Academic Excellence Project 5–100

² Supported by RaCAF ANR-15-CE40-0016-01 RaCAF grant



© Mikhail Andreev, Gleb Posobin and Alexander Shen;
licensed under Creative Commons License CC-BY

43rd International Symposium on Mathematical Foundations of Computer Science (MFCS 2018).

Editors: Igor Potapov, Paul Spirakis, and James Worrell; Article No. 2; pp. 2:1–2:12

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction: stopping time complexity

Imagine that you explain to someone which exit on a long road she should take. You can just say “ N th exit”; for that you need $\log N$ bits. You may also say something like “the first exit after the first bridge”, and this message has bounded length even if the bridge is very far away.³

More formally, consider a machine with one-directional read-only input tape that contains bits $x_0, x_1, \dots, x_n, \dots$. We want to program the machine in such a way that it stops after reading bits x_0, \dots, x_{n-1} (and never sees x_n and the subsequent bits). Obviously, the complexity of this task does not depend on the values of x_n, x_{n+1}, \dots , because the machine never sees them, so this complexity should be a function of a bit string $x = x_0x_1 \dots x_{n-1}$. It can be called the “stopping time complexity” of x .

Such a notion was introduced recently by Vovk and Pavlovic [9]. In their paper an “interactive” version of stopping time complexity is considered where even (x_{2n}) and odd (x_{2n+1}) terms are considered differently, but this is just a special case, so we do not consider this setting. It turns out that the stopping time complexity is a special case of conditional Kolmogorov complexity with structured conditions. (In this paper we consider the plain version of stopping time complexity and postpone similar questions for prefix versions.)

The Kolmogorov complexity was introduced independently by Solomonoff, Kolmogorov, and Chaitin to measure the “amount of information” in a finite object (say, in a binary string). One can also consider the *conditional* version of complexity where some other object (a *condition*) is given “for free”. Later different versions of Kolmogorov complexity appeared (plain, prefix, a priori, monotone complexities). We assume that the reader is familiar with basic notions of algorithmic information theory, see, for example, [6] for a short introduction and [7] for a detailed exposition.

For the plain version of stopping time complexity we prove the equivalence between five different definitions (Section 2). First, we show that it can be equivalently defined as (1) the minimal plain complexity of a machine with one-way read-only input tape that stops after reading x , or (2) the minimal enumeration complexity of a prefix-free set that contains x . Then we show how the stopping time complexity can be expressed in terms of plain conditional complexity that is monotone with respect to conditions. Namely, we prove that (3) the stopping time complexity equals $C(x|x^*)$ where x is used both as an object and a condition. Of course, according to standard definitions, the complexity $C(x|x)$ is $O(1)$, but now we treat these two strings x differently (and use a star in the notation to stress this). One may say that the topologies in the space of objects and the space of conditions are different. The first x (object to be described) is considered as an isolated object (terminated string). The second x (in the condition) is considered as a prefix of an infinite sequence. In [7, Section 6.3] this approach is described in general (see also [5] for even more general setting); to make this paper self-contained, we give all necessary definitions for our special case. We call this version of complexity “monotone-conditional complexity” since this function is monotone with respect to the condition. Then we provide a characterization of stopping time complexity in quantitative terms proving that (4) stopping time complexity is the minimal upper semicomputable function satisfying some restrictions (no more than 2^n prefixes of any given sequence could have complexity at most n). Finally, we point out the connections with the relativized version $C^A(x)$ of plain complexity and prove that (5) the

³ We do not allow, however, the description “the last exit before the bridge”, since it uses information that is unavailable at the moment when we have to take the exit.

stopping time complexity of a binary string x is the maximal value of $C^A(x)$ for all oracles (infinite bit sequences) A that have prefix x .

Having such a robust definition for plain stopping time complexity, one may ask whether similar characterizations can be obtained for a more general notion of $C(y|x^*)$ where x and y are arbitrary strings. Unfortunately, here the situation is much worse, and we prove mostly negative results (Section 3). We show that while $C(y|x^*)$ can be defined as a minimal plain complexity of a prefix-stable program that maps x to y (Theorem 12), it cannot be defined as a minimal plain complexity of a prefix-free program that maps some prefix of x to y (Theorem 13; this result answers a question posed in [3]). Then we show that the attempt to define $C(y|x^*)$ by quantitative restrictions also fails: we get another function that may be up to two times less (Theorem 14).

2 Equivalent definitions

2.1 Machines and prefix-free sets

Consider a Turing machine M that has one-directional read-only input tape with binary alphabet, and a work tape with arbitrary alphabet (or many work tapes). Let x be a binary string. We say that M *stops at* x if M , being started with the input tape x (and an empty work tape, as usual), reads all the bits of x and stops without trying to read more bits. (We assume that initially the input head is on the left of x , so it needs to move right before seeing the first bit of x .) For a given x , we may consider the *minimal plain Kolmogorov complexity of a machine M that stops at x* . This quantity is independent (up to $O(1)$ -additive term) of the details of the definition (work tape alphabet, number of work tapes, etc.) since computable conversion algorithms exist, and a computable transformation may increase complexity only by $O(1)$. We arrive at the following definition:

► **Definition 1.** The *plain stopping time complexity* of x is the minimal plain Kolmogorov complexity of a Turing machine that stops at x .

Here is a machine-independent equivalent characterization of the plain stopping time complexity.

► **Theorem 2.** *Plain stopping time complexity of x equals (up to an $O(1)$ additive term) the minimal complexity of a program that enumerates some prefix-free set containing x .*

(A set of strings is called *prefix-free* if it does not contain a string and its proper prefix at the same time.)

Proof. One direction is simple: For a Turing machine M of the type described the set $\{x: M \text{ stops at } x\}$ is enumerable (we may simulate all runs) and prefix-free (if M stops at some x , then for every extension y of x the machine M will behave in the same way on tapes x and y , so M with input y stops after reading x and never reads the rest of y). This computable conversion (of a machine into an enumeration program) increases complexity at most by $O(1)$.

The other direction is a bit more complicated. Imagine that we have a program that enumerates some prefix-free set U of strings. How can we construct a machine that stops exactly at the strings in U ? Initially no bits of x are read. Enumerating U , we wait until some element u of U appears. (If this never happens, the machine never stops, and this is OK.) If u is empty, machine stops. In this case U cannot contain non-empty strings (being prefix-free), so the machine's behavior is correct. If u is not empty, we know that empty

string is not in U (since U is prefix-free), so we may read the first bit of x without any risk, and get some one-bit string v . Then we wait until v or some extension of v appears in U (it may have already happened if u is an extension of v). If v itself appears, the computation stops; if a proper extension of v appears, then v is not in U and we can safely read the next bit, etc. It is easy to check that indeed this machine stops at some x if and only if x belongs to U . ◀

2.2 Monotone-conditional complexity

In this section we show how the stopping time complexity can be obtained as a special case of some general scheme [5, 8, 7]. This scheme can be used to define different versions of Kolmogorov complexity. We consider *decompressors*, called also *description modes*. In our case decompressor is a subset D of the set

$$(\text{descriptions}) \times (\text{conditions}) \times (\text{objects}).$$

Here descriptions, conditions, and objects are binary strings. If $(p, x, y) \in D$, we say that p is a *description of y given x as condition*, and define the *conditional complexity of y given x* (with respect to the description mode D) as the length of the shortest description. The different versions of complexity correspond to different topologies on the spaces involved, and imply different restrictions on description modes. This is explained in [8] or [7, Chapter 6], and we do not go into technical details here. Let us mention only that descriptions and objects can be considered as isolated entities (terminated strings, natural numbers) or prefixes of an infinite sequence (extension of a string provides more information than the string itself). In this way we get four classical versions of complexity:

	isolated descriptions	descriptions as prefixes
isolated objects	plain complexity	prefix complexity
objects as prefixes	decision complexity	monotone complexity

As noted in [7], one can also consider different structures on the condition space, thus getting eight versions of complexity instead of four in the table. In this paper we use only one of them: objects and descriptions are isolated objects, and conditions are considered as prefixes. (Vovk and Pavlovic [9] consider also another version of stopping time complexity that corresponds to the other topology on the description space, but we do not consider this version now.)

To make this paper self-contained, let us give the definitions tailored to the special case we consider (the plain version of monotone-conditional complexity). In this case the set D is called a *description mode* if it satisfies the following requirements:

- D is (computably) enumerable;
- for every p and x there exists at most one y such that $(p, x, y) \in D$;
- if $(p, x, y) \in D$ and x is a prefix of some x' , then $(p, x', y) \in D$.

The last requirement reflects the idea that x is considered as a known prefix of a yet unknown infinite sequence; if x' extends x , then x' contains more information than x and can be used instead of x . To stress this kind of monotonicity, we use $*$ in the notation suggested by the following definition.

► **Definition 3.** For a given description mode D , we define the function

$$C_D(y|x*) = \min\{|p| : (p, x, y) \in D\}$$

and call it *monotone-conditional complexity of y with condition x with respect to description mode D* .

By definition, if x is a prefix of some x' , the same description can be used, so $C_D(y|x'*) \leq C_D(y|x*)$. Therefore, this function is indeed monotone with respect to the condition in a natural sense.

One could also use a name *plain monotone-conditional complexity* to distinguish this notion from prefix monotone-conditional complexity that can be defined in a similar way by adding the monotonicity restriction along the p -coordinate.

► **Proposition 4 (Solomonoff–Kolmogorov’s optimality theorem).** There exists a description mode D that makes C_D minimal up to $O(1)$ additive term in the class of all functions $C_{D'}$ for all description modes D' .

Proof. As usual, we first note that description modes can be effectively enumerated. This enumeration is obtained as follows. We generate all enumerable sets of triples and then modify them in such a way that the modified set becomes a description mode and is left unchanged if it already was a description mode. Namely, when a triple (p, x, y) appears in the enumeration, we add this triple and all triples (p, x', y) for all extensions x' of x , unless the second condition would be violated after that; in the latter case we ignore (p, x, y) .

Let U_n be the n th set in this enumeration. The optimal set U can be constructed as

$$U = \{(0^n 1 p, x, y) : (p, x, y) \in U_n\};$$

the standard argument shows that $C_U \leq C_{U_n} + n + 1$ as required. ◀

► **Definition 5.** Fix some optimal description mode D provided by Proposition 4. The function $C_D(y|x*)$ is denoted by $C(y|x*)$ and called the (plain) *monotone-conditional complexity of y given x* , or the (plain) *conditional complexity of y given x as a prefix*.

If we omit the third requirement for description modes, we get the standard conditional complexity $C(y|x)$ in the same way. The notation we use (placing $*$ after the condition) follows [3] though a different version of monotone-conditional complexity is considered there. In general, $C(y|x*)$ is greater than the standard conditional complexity $C(y|x)$ since we have more requirements for the description modes. One may say also that the condition now is weaker than in $C(y|x)$ since we do not know where x terminates. It is easy to show that the difference is bounded by $O(\log|x|)$, since we need at most $O(\log|x|)$ bits to specify how many bits should be read in the condition x . Difference of this order is possible: for example, $C(n|0^n) = O(1)$, but $C(n|0^n*) = C(n) + O(1)$ (the condition 0^n is a prefix of a computable sequence $000\dots$, so it does not help at all).

The following simple result shows that the plain stopping time complexity (Definition 1) is a special case of this definition when $x = y$ (so we do not need a separate notation for the stopping time complexity).

► **Theorem 6.** *The complexity $C(x|x*)$ is equal (up to $O(1)$ additive term) to the plain stopping time complexity of x .*

Proof. Let D be a description mode. Then for every p we may consider the set S_p of strings x such that $(p, x, x) \in D$. This set is prefix-free: if (p, x, x) and (p, x', x') belong to D and x is a prefix of x' , then $(p, x', x) \in D$ according to the third condition, and then $x = x'$ according to the second condition. The algorithm enumerating S_p can be constructed effectively if p is known, so its complexity is bounded by the length of p (plus $O(1)$, as usual). Choosing the shortest p such that $(p, x, x) \in D$, we conclude that the minimal complexity of an algorithm enumerating a prefix-free set containing x does not exceed $C(x|x*) + O(1)$.

Going in the other direction, consider an optimal decompressor $U(\cdot)$ that defines the (plain Kolmogorov) complexity of programs enumerating sets of strings. A standard trimming argument shows that we may modify U in such a way that all algorithms $U(p)$ enumerate only prefix-free sets of strings (not changing the sets there were already prefix-free). Then consider a set D of triples

$$(p, x, y) \in D \Leftrightarrow y \text{ is a prefix of } x \text{ and } y \text{ is enumerated by } U(p).$$

This set is obviously enumerable; the second requirement is satisfied since $D(p)$ enumerates a prefix-free set; the third requirement is true by construction, so D is a description mode. If p is the shortest description of a program that enumerates a set containing x , then $(p, x, x) \in D$, so $C_D(x|x^*) \leq |p|$. Switching to the optimal description mode, we get similar inequality with $O(1)$ additive term, as required. \blacktriangleleft

Another simple observation shows that indeed this complexity may be called the *stopping time* complexity.

► **Proposition 7.** If x has length n , then $C(x|x^*) = C(n|x^*) + O(1)$.

Proof. If D is the optimal description mode used to define $C(y|x^*)$, we may consider a new set $D' = \{(p, u, |x|) : (p, u, x) \in D\}$ that also is a description mode, and then note that $C_{D'}(n|x^*) \leq C_D(x|x^*)$. For the other direction, we consider $D' = \{(p, u, z) : \exists n [(p, u, n) \in D, |u| \geq n, \text{ and } z = (n\text{-bit prefix of } u)]\}$. \blacktriangleleft

► **Remark.** If $a_0a_1a_2\dots$ is a computable sequence, then

$$C(a_0\dots a_{n-1}|a_0\dots a_{n-1}^*) = C(n|a_0\dots a_{n-1}^*) = C(n)$$

with $O(1)$ -precision (the constant depends on the computable sequence, but not on n), so the stopping time complexity can be considered as a generalization of the plain complexity (of a natural number n).

2.3 Quantitative characterization

There is a well known characterization (see, e.g., [8, Section 1.1, Theorem 8]) for plain complexity in terms of upper semicomputable functions that satisfy some properties. Recall that a function is called *upper semicomputable* if it is a pointwise limit of a decreasing sequence of uniformly computable total functions. (Now we need this notion for integer-valued functions; in this case we may assume without loss of generality that the total computable functions used in the definition are also integer-valued; in the general case one needs to consider rational-valued functions.) An equivalent definition of a semicomputable natural-valued function $S(x)$ requires the set $\{(n, x) : S(x) < n\}$ to be enumerable.

Plain complexity function $C(x)$ is upper semicomputable; we know also that

$$\#\{x : C(x) < n\} < 2^n \tag{*}$$

since there are less than 2^n programs of length less than n . The characterization that we mentioned says that there exist a minimal (up to $O(1)$ additive term) upper semicomputable function that satisfies the requirement (*), and it coincides with the plain complexity function with $O(1)$ -precision.

It turns out that this characterization can be generalized to plain stopping time complexity (though the proof becomes more involved). Consider upper semicomputable functions $S(x)$

on strings that have the following property: for each infinite binary sequence α and for each n there exists less than 2^n prefixes x of α such that $S(x) < n$. The following statement is true (it appears as Theorem 18 in the extended version of Vovk–Pavlovic’s paper [9]).

► **Theorem 8.** *There exist a minimal (up to $O(1)$ additive term) function in this class; it coincides with the plain stopping time complexity $C(x|x^*)$ with $O(1)$ -precision.*

Proof. The easy part is to show that $C(x|x^*)$ belongs to the class. It is upper semicomputable, since in general the function $C(x|y^*)$ is upper semicomputable (enumerating the set D of triples, we get better and better upper bounds, finally reaching the limit value).

Let α be some infinite sequence. There are less than 2^n algorithms of complexity less than n enumerating prefix-free sets, and each of this prefix-free sets may contain at most one prefix of α . So the second condition is also true.

In the other direction we use some online (interactive) version of Dilworth’s theorem (saying that a partially ordered finite set where maximal chain is of length at most k can be partitioned into k antichains) where the set is growing and splitting into antichains should be performed at each stage (and cannot be changed later). The exact statement is as follows.

Consider a game with two players. Alice and Bob alternate. Alice may at each move (irreversibly) mark a vertex of the full binary tree. The restriction is that each infinite branch should contain at most k marked vertices. Bob replies by assigning a color from $1, \dots, k$ to the newly marked vertex. No vertices of the same color should be comparable (be on the same branch). The colors cannot be changed after they are assigned. Bob loses if he is unable to assign color at some stage (not violating the rules).

► **Lemma 9.** *Bob has a computable strategy that prevents him from losing.*

Proof of Lemma 9. This lemma can be proven in different ways. In the extended version of Vovk–Pavlovic’s paper [9] the following simple strategy is suggested: Bob assigns the first available color. In other terms, for a new vertex x Bob chooses the first color that is not used for any vertex comparable with x . One needs to check that k colors are always enough. It is not immediately obvious, since more than k vertices could be comparable with x (being its descendants, for example). However, we may note that during the process:

- Colors of comparable vertices are different. (By construction.)
- If a vertex x gets color i , then each smaller color is used either for a predecessor of x or for a descendant of x . (By construction.)
- If x is a vertex (colored or not), T_x is the set of colors used in the subtree rooted at x (including x itself), and P_x is the set of colors used on the path to x (not including x), then T_x and P_x are disjoint and T_x is the initial segment in the complement to P_x . (Indeed, the disjointness is mentioned above. If y appears in T_x , then all smaller colors appear either below y (therefore in P_x or in T_x), or above y (therefore in T_x).
- The sets T_{x_0} and T_{x_1} for two brother vertices x_0 and x_1 are comparable with respect to inclusion. (Indeed, they are two initial segments of the same ordered set, the complement to P_{x_0} or P_{x_1} ; note that $P_{x_0} = P_{x_1}$.)
- For each x the total number of colors used in T_x is minimal, i.e., this number equals the maximal number of marked vertices on some path in T_x . (Induction using the previous property.)

The last property implies that Bob never uses more than k colors, since by assumption the total number of marked vertices on a path is at most k .

A different description of the same strategy that explains why it is successful is provided in the extended version of this paper [1]. ◀

Now let us show how the lemma is used to finish the proof of Theorem 8. Let S be a function in the class; since S is upper semicomputable, Alice may, given n , enumerate strings x such that $S(x) < n$; we know that there is at most 2^n strings of this type along any branch of the tree, so Alice never violates the restriction for $k = 2^n$. The lemma then says that Bob can assign 2^n colors (represented as n -bit strings) to all the vertices in such a way that compatible vertices (a string and its prefix) never get the same color. We run these games for all n in parallel; if vertex x gets color c , we put x into an enumerable set indexed by c . The rules of the game guarantee that all these sets are prefix-free, and the algorithm enumerating c th set needs only $|c|$ bits of information. So, if $S(x) < n$, there exists an algorithm of complexity $n + O(1)$ that enumerates a prefix-free set containing x . This means that $C(x|x^*) \leq S(x) + O(1)$ as required. ◀

2.4 Oracles and the stopping time complexity

It is natural to compare the stopping time complexity $C(x|x^*)$ and the relativized complexity $C^X(x)$ where X is some oracle (infinite binary sequence) that has x as a prefix. (The relativized complexity $C^X(x)$ can be naturally defined as a function of two arguments, a binary string x and an infinite binary sequence X , up to $O(1)$ additive term.)

It is easy to see that

$$C^X(x) \leq C(x|x^*)$$

for every X that has prefix x : an oracle access to entire sequence X is more powerful than a bit-by-bit sequential access to x without the right to read too much (beyond x). More formally, let D be a set of triples (p, x, y) used to define $C(y|x^*)$ (Definition 5). Then we say that p is a description of x with oracle X (as the definition of $C^X(x)$ requires) if $(p, z, x) \in D$ for some z that is a prefix of X . For a given X every string p can be a description of only one x , since D is monotone. If $(p, x, x) \in D$ and X is an extension of x , then p is a description of x , and we get the required inequality.

The “last exit before the bridge” example shows that $C^X(x)$ can be much smaller than $C(x|x^*)$ for *some* extensions X of x : we have $C(0^n|0^{n*}) = C(n) + O(1)$, but $C^X(0^n) = O(1)$ for $X = 0^n10^\infty$.

So it is natural to take *maximum* over all oracles X that extend a given string x . Indeed this approach works:

► Theorem 10.

$$C(x|x^*) = \max\{C^X(x) : X \text{ is an infinite extension of } x\} + O(1).$$

Proof. As we have already mentioned, $C^X(x) \leq C(x|x^*) + O(1)$ for every infinite extension X of x . This shows that right hand side does not exceed the left hand side.

Before proving the reverse inequality, let us discuss informally its meaning (this discussion is not used in the argument below and can be omitted). The reverse inequality is a minimax-type result that shows that either (1) there exists a short program that produces x given *any* extension of x as an oracle (and never reads bits after x , but this is not so important for us now), or (2) there exists a “hard to use” extension X of x such that *any* program that computes x given X is long. In other words, it is not possible that for every extension X of x there exists a short program that computes x given X as an oracle, but these programs depend on X and only a much longer program works for all extensions.

For the proof of the reverse inequality we use the quantitative characterization of stopping time complexity (Theorem 8). Let $S(x)$ be the value of the right hand side. It is enough to

prove that S is upper semicomputable and that $S(x) < n$ cannot happen for 2^n different prefixes x of some infinite branch X .

The second claim follows directly from the definition. Let x_1, \dots, x_k be some prefixes of an infinite sequence X such that $S(x_i) < n$ for all $i = 1, \dots, k$. We need to show that $k < 2^n$. Since $S(x_i)$ is defined as maximum and X is an extension of x_i , we know that $C^X(x_i) < n$ for all i and the same X . It remains to note that the number of different programs of length less than n is smaller than 2^n (and the same programs with the same oracles give the same results).

To show that $S(x)$ is upper semicomputable, we use the standard compactness argument. As usual, it is enough to show that the binary relation $S(x) < n$ is (computably) enumerable. Indeed, for every x , the set $\{X : C^X(x) < n\}$ is the union, taken over all strings p of length less than n , of the sets

$$\{X : p \text{ is a description of } x \text{ with oracle } X\}.$$

Each of these sets is an open set in the Cantor space, since every terminating oracle computation uses only a finite part of the oracle, and the intervals in the Cantor space that form these sets, can be effectively enumerated for all p and x . The inequality $S(x) < n$ means that the union of these intervals for all p of length less than n covers the Cantor space. Now compactness guarantees that this happens already at some finite stage of the enumeration, so the property $S(x) < n$ is indeed enumerable. ◀

3 Non-equivalence results

3.1 Prefix-stable or prefix-free functions?

Looking at the characterization of $C(x|x^*)$ as the minimal enumeration complexity of a prefix-free set containing x (Theorem 6), one can ask whether a similar characterization works for the general case, i.e., whether $C(y|x^*)$ can be characterized as a minimal complexity of programs (machines) with some property. The answer is ‘yes’, but we should be careful choosing a property of programs used in this characterization. Here are the details.

- **Definition 11.** A partial function f defined on binary strings is called
- *prefix-free* if its domain is prefix-free (function is never defined on a string and its extension at the same time);
 - *prefix-stable* if for every x , if $f(x)$ is defined, then f is defined and has the same value on all (finite) extensions of x .

It is easy to see that the definition of $C(y|x^*)$ can be reformulated in terms of prefix-stable functions:

- **Theorem 12.** *The minimal plain complexity of a program that computes a prefix-stable function mapping x to y is equal to $C(y|x^*) + O(1)$.*

Proof. A description mode can be considered as a family of prefix-stable functions (indexed by the first argument p). This shows that there exist a program for a prefix stable function mapping x to y of complexity at most $C(y|x^*) + O(1)$. On the other hand, one can efficiently “trim” all programs to make them prefix-stable; if \hat{u} is the trimmed version of a program u and U is the decompressor used to define plain complexity of programs, then the set $D = \{p, x, \widehat{U(p)}(x) : p, x\}$ satisfies the conditions and may be considered as a decompressor in the definition of $C(y|x^*)$. Using this decompressor, we get the reverse inequality. ◀

More interesting question: is a similar statement true for *prefix-free* functions instead of prefix-stable ones? As we mentioned above, Theorem 6 implies that this is the case when $x = y$. (We spoke about programs that stop at x , but we may assume without loss of generality that the output is also x .) But in the general case it is not true anymore. Let us make this statement more precise. The first idea is to consider the minimal plain complexity of a program computing a prefix-free function mapping x to y . But this quantity does not look reasonable: the complexity of empty string Λ with condition x defined in this way may be arbitrarily large (and is actually the stopping time complexity of the condition x).

A more reasonable approach is to consider function $C'(y|x^*)$ defined as the minimal complexity of a prefix-free program that maps *some prefix of x* to y . This approach still does not work, as the following result shows.

► **Theorem 13.** *The inequality $C(y|x^*) \leq C'(y|x^*) + c$ holds for some c and for all x, y . The reverse inequality does not: there exist strings x_i, y_i (for $i = 0, 1, 2, \dots$) such that $C(y_i|x_i^*)$ is bounded while $C'(y_i|x_i^*)$ is unbounded.*

Proof. The first part is easy: if an algorithm computing a prefix-free function f is given, we can effectively transform it into an algorithm that computes its prefix-stable extension g such that $g(x) = y$ if $f(u) = y$ for some prefix u of x .

For the second part of the proof (using game arguments in the sense of [4]) see the extended version of this paper [1]. ◀

Theorem 13 implies that the conjecture from [3, p. 254] is false, and the function C_T defined there may exceed C_E more than by $O(1)$ additive term. We do not go into the details of the definition used in [3]; let us mention only that $C_E(y_i|x_i)$ is bounded while $C_T(y_i|x_i)$ is not: for every twice prefix machine (as defined in [3, p. 252]) we get a prefix-free function if we fix the first argument (denoted there by p).

3.2 Quantitative characterization of $C(y|x^*)$ works only up to factor 2

In Section 2.3 we provided a quantitative characterization of stopping time complexity, or $C(x|x^*)$, with $O(1)$ -precision (Theorem 8). The natural question is whether a similar characterization works in the general case, i.e., for $C(x|y^*)$. As we will see, the answer is negative.

For $C(x|y)$ (the standard version of conditional complexity, with no monotonicity requirement) such a characterization is well known: $C(x|y)$ is the minimal upper semicomputable function of two arguments $K(x, y)$ such that for every string y and every number n there is at most 2^n different strings x such that $K(x, y) < n$.

The natural approach is to keep this restriction and add the monotonicity requirements:

$$K(x, y0) \leq K(x, y) \text{ and } K(x, y1) \leq K(x, y), \text{ for every } x \text{ and } y.$$

We get some class of functions (that are upper semicomputable, satisfy the cardinality restriction and are monotone in the sense described). Can we characterize $C(x|y^*)$ as the minimal function in this class? No, as the following theorem shows.

► **Theorem 14.**

- (a) *Function $C(x|y^*)$ belongs to this class.*
- (b) *There exists a minimal (up to $O(1)$ additive term) function in this class;*
- (c) *Function $C(x|y^*)$ is not minimal in this class: there exist a function K in this class, and sequences of strings x_n and y_n such that $K(x_n, y_n) \leq n$, but $C(x_n|y_n^*) \geq 2n - c$ for some c and for every n .*

(d) The factor 2 that appears in the previous statement is optimal: if K is a function in the class (for example, the minimal one), then $C(x|y^*) \leq 2K(x, y) + c$ for some c and for all x and y .

Proof. The statements (a) and (b) are “good news”, while the statement (c) is “bad news” showing that our characterization does not work. (It is possible *a priori* that one can get a natural characterization of $C(x|y^*)$ by adding some other restrictions, but it is quite unclear what kind of restrictions could help here.) Finally, the statement (d) partly saves the situation and shows that the minimal function in the class and $C(x|y^*)$ differ at most by factor 2.

The statement (a) is obvious; note that $C(x|y^*)$ is bigger than $C(x|y)$, so the cardinality restriction remains true. Other requirements immediately follow from the definition.

The statement (b) can be proved in a standard way. We can enumerate all functions in the class and get a uniformly computable sequence of functions $K_m(x, y)$. For that we enumerate all monotone upper semicomputable functions and then “trim” them by deleting small values that make the cardinality restriction false. Then we construct the minimal function $K(x, y)$ by letting

$$K(x, y) = \min_m (K_m(x, y) + m + 1).$$

It is upper semicomputable and monotone; for every y , the set of x such that $K(x, y) < n$ is the union of sets $\{x: K_m(x, y) < n - m - 1\}$ that have cardinality at most 2^{n-m-1} , and $2^{n-1} + 2^{n-2} + \dots < 2^n$. The function K is minimal, since $K \leq K_m + m + 1$.

For the proofs of statements (c) and (d) see the extended version of this paper [1]. These proofs use game arguments in the sense of [4]. ◀

4 Questions

► **Question 1.** Imagine Turing machines with two read-only input tapes; for such a machine M consider a function f_M such that $f_M(x, y) = z$ if M stops at x and y on first and second tape respectively (reading all bits and not more) and produces z . Could we characterize the functions f_M (called *twice prefix free* in [3, page 242]) or at least their domains? Such a domain is an enumerable set of pairs that does not contain two pairs (x, y) and (x', y') where x is compatible with x' (one is a prefix of the other) and y is compatible with y' . Still this necessary condition is not sufficient, as the following argument shows. Let z_i be a computable sequence of pairwise incompatible strings (say, $z_i = 0^i 1$). Let P and Q be two enumerable sets that are inseparable (do not have a decidable separating set). Consider the set of pairs that contains

- $(z_i 0, z_i 0)$ for all i ;
- $(z_i, z_i 1)$ for $i \in P$;
- $(z_i 1, z_i)$ for $i \in Q$.

This set satisfies the necessary condition above (does not contain two compatible pairs). However, assume that some twice prefix free machine has this set as a domain. Then it should terminate after reading $z_i 0$ on the first tape and $z_i 0$ on the second tape. Consider the last zero bits on both tapes. One of these bits should be read first (if they are read simultaneously, we may choose any of two). If this is the first bit, then $i \in P$ is impossible (since the machine cannot read 1 on the second tape before reading 0 on the first tape). For the same reason, $i \in Q$ is impossible if the second bit is read first. Therefore, a decidable separator exists.

Can we add some conditions to get a characterization of domains of twice prefix free machines? What do we get if we define stopping time complexity for pairs using machines of this type? Does it have some equivalent description (for example, can it be defined using monotone-conditional complexity with pairs as conditions, Section 2.2)?

► **Question 2.** Do we have $C(x|x^*) = \max_z C(x|z) + O(1)$ where the maximum is taken over all *finite* extensions z of x ? (The problem is that the compactness argument does not work anymore.)

► **Question 3.** One may consider the function

$$K(x, y) = \max\{C^Y(x) : Y \text{ is an infinite extension of } y\}$$

We have shown that for $x = y$ it coincides with $C(x|x^*)$, showing that it does not exceed $C(x|x^*)$ and satisfies the quantitative restrictions of Theorem 8. Both arguments remain valid (with minimal changes) for the general case, and we conclude that $K(x, y)$ defined in this way does not exceed $C(x|y^*)$ and also satisfies the cardinality restrictions of Theorem 14, (b). However, now these upper bound and lower bound differ, and we do not know where the function $K(x, y)$ defined as shown above lies. Does it coincide with its upper bound $C(x|y^*)$ for arbitrary x and y , or with its lower bound, the minimal upper semicomputable function that satisfies the cardinality requirements (see Theorem 14), or neither?

References

- 1 Mikhail Andreev, Gleb Posobin, and Alexander Shen. Plain stopping time and conditional complexities revisited. *CoRR*, abs/1708.08100, October 2017.
- 2 Mikhail Andreev, Gleb Posobin, and Alexander Shen. Stopping time complexity, abstract. *Dagstuhl Reports, Computability Theory, Dagstuhl Seminar 17081, February 2017*, page 97, 2017.
- 3 Alexey V. Chernov, Marcus Hutter, and Jürgen Schmidhuber. Algorithmic complexity bounds on future prediction errors. *Information and Computation*, 205(2):242–261, 2007.
- 4 Andrei A. Muchnik, Ilya Mezhirov, Alexander Shen, and Nikolay Vereshchagin. Game interpretation of Kolmogorov complexity. *CoRR*, abs/1003.4712, March 2010.
- 5 Alexander Shen. Algorithmic variants of the notion of entropy. *Soviet Mathematics Doklady*, 29(3):569–573, 1984.
- 6 Alexander Shen. Around Kolmogorov complexity: Basic notions and results. In Vladimir Vovk, Harris Papadopoulos, and Alexander Gammerman, editors, *Measures of Complexity: Festschrift for Alexey Chervonenkis*, chapter 7, pages 75–116. Springer, Cham, 2015.
- 7 Alexander Shen, Vladimir A. Uspensky, and Nikolay Vereshchagin. *Kolmogorov complexity and algorithmic randomness*, volume 220 of *Mathematical Surveys and Monographs*. American Mathematical Society, <http://www.lirmm.fr/~ashen/kolmbook.pdf>, 2017.
- 8 Vladimir A. Uspensky and Alexander Shen. Relations between varieties of Kolmogorov complexities. *Mathematical Systems Theory*, 29(3):271–292, Jun 1996.
- 9 Vladimir Vovk and Dusko Pavlovic. Universal probability-free conformal prediction. In Alexander Gammerman, Zhiyuan Luo, Jesús Vega, and Vladimir Vovk, editors, *Conformal and Probabilistic Prediction with Applications*, pages 40–47, Cham, 2016. Springer, see also <https://arxiv.org/pdf/1603.04283.pdf> (March 2016; extended version, April 2017).