Towards Analysis of Information Structure of Computations

Anatol Slissenko

Towards Analysis of Information Structure of Computations

Anatol Slissenko
Laboratory for Algorithmics, Complexity and Logic (LACL)
University Paris East Créteil (UPEC)
61 av. du Général de Gaulle 94010, Créteil France
and ITMO, Saint-Patersburg, Russia
slissenko@u-pec.fr

Abstract

The paper discusses how one can try to analyze computations, and maybe computational problems from the point of view of information evolution. The considerations presented here are very preliminary. The long-standing goal is twofold: on the one hand, to find other vision of computations that may help to design and analyze algorithms, and on the other hand, to understand what is realistic computation and what is real practical problem. The concepts of modern computer science, that came from classical mathematics of pre-computer era, are overgeneralized, and for this reason are often misleading and counterproductive from the point of view of applications. The present text discusses mainly what classical notions of entropy might give for analysis of computations. In order to better understand the problem, a philosophical discussion of the the essence and relation of knowledge/information/uncertainty in algorithmic processes might be useful.

Keywords: Computation, Problem, Partition, Entropy, Metric.

1 Introduction

The goal of this paper is to discuss along what lines one can look for ways to describe the quantity of information transformed by computations. This may permit to better understand the computations themselves and, possibly, what is practical

Talk at the conference "Philosophy, Mathematics, Linguistics: Aspects of Interaction 2012" (PhML-2012), Euler International Mathematical Institute, May 22-25, 2012.

computation and what is practical algorithmic problem. The considerations presented here are very preliminary, more of philosophical than of mathematical flavor. We consider rather straightforward geometrical and information ideas that come to mind. Usually they are not sufficient taken directly. Making explicit the obstacles may help to devise more productive approaches[1].

In Introduction we give some arguments that illustrate that the mathematical formulations of computational problems we usually consider, are overgeneralized, and sometimes this hinders the development of practical algorithms or the understanding why certain algorithms for theoretically hard problems work well in practice. In Section 2 we outline some approaches to measuring information in computations, and discuss their weak and strong points. Section 3 is about the structure of problems for which we can presumably develop measures of information along the lines described in the previous section. It contains also a short discussion of the role of linguistic considerations in describing practical problems.

Why traditional mathematical settings look too general for practical computer science? And when it is inevitable and when maybe not?

Most notions used in theoretical computer science either come from mathematics of pre-computer era or are developed along mathematical lines of that epoch. From mathematics of pre-computer era the computational theory borrows logics, logical style algorithms (lambda-calculus, recursive function, Turing machine), general deductive systems (grammars), Boolean functions, graphs. More specific notions like finite automata, Boolean circuits, random access machines etc., though motivated by modeling of computations, are of traditional mathematical flavor. All these concepts played and continue to play fundamental role in theoretical computer science, however other, more adequate concepts are clearly needed.

I can illustrate this thesis by Boolean functions and their realization by circuits. Almost all Boolean functions of n variables have exponential circuit complexity $(2^n/n)$ [9], and there is an algorithmic method to find such an optimal realization for a given 'random' function [6]. But it is clear that even for $n = 64$, that is not so big from practical viewpoint, one cannot construct a circuit with $2^n/n$ gates. So one can state that almost all Boolean functions will never appear in applications. The notion of Boolean function is of evident practical value, but not in its generality. All this does not say that the general notion and the mentioned result on the complexity of realization are useless in theory (moreover, they are known to be useful). But an optimal circuit construction for almost all Boolean functions is not of great value for practical Boolean functions.

Consider another example. We know that the worst-case complexity of the de-

[1]Some of them were developed later to become quite mathematical.

cidability of the theory of real addition is exponential [2]. This theory is a set of valid closed formulas that are constructed from linear inequalities with integer coefficients with the help of logical connectives, including quantifiers over real numbers (in fact, only rational numbers are representable by such formulas, as the only admissible constants are integers). In particular, one can express in this theory the existence of a solution of a system of linear inequalities, and various parametric versions of this problem, e.g., whether such a solution exists for any value of some variable in some interval. The complexity of recognition of validity of the formulas grows up with the number of quantifier alternations.

The mentioned exponential lower bound on the computational complexity of the theory of real addition is proven along the following lines. Denote $\mathbb{B} =_{df} \{0, 1\}$ and denote by $\mathbb{B}^*$ the set of all strings over $\mathbb{B}$. Under some technical constraints for any algorithm f from $\mathbb{B}^*$ to $\mathbb{B}$, whose complexity is bounded by some exponential function φ, and for any its input $x \in \mathbb{B}^*$ one can construct a formula $\Phi(f, x)$ of sufficiently small size (polynomial in the size of f and x) that is valid if and only if $f(x) = 0$.

Within a reasonable algorithmic framework (e.g., for some random access machines, like LRAM from [10]) one can construct a predicate $f : \mathbb{B}^* \to \mathbb{B}$ whose upper bound on computational complexity is φ, and any algorithm that computes this predicate has lower bound $\theta \cdot \varphi$, for some $0 < \theta < 1$. This f is a diagonal algorithm, I do not know other kind of algorithms for this context. Such a diagonal algorithms works like follows. Assume that the complexity of computing $\varphi(|x|)$, where $|x|$ is the length of $x \in \mathbb{B}^*$, is bounded by its value $\varphi(|x|)$. The algorithm f computes $\varphi(|x|)$ and makes roughly $\varphi(|x|)$ steps of simulation of algorithm with the code x applied to input x. If the process ends within less that $\varphi(|x|)$ steps then f outputs the value different from the value computed by the algorithm with the code x, otherwise it outputs say, 0 (in the latter case the value is not important).

Thus, the recognition of the validity of formulas $\Phi(f, x)$ has a high complexity. But they are not formulas that appear in practice. Moreover, practical formulas, that may have a good amount of quantifier alternations, are semantically much simpler, they never speak about diagonal algorithms, though may speak about practical algorithms, e.g., about execution and properties of hard real-time controllers.

The just presented argument is valid for all negative complexity results (undecidability, high lower bounds, relative hardness) with the existing proofs. And here one arrives at another 'incoherence' between theory and practice that can be illustrated by the TAUT problem, i.e., by the problem of recognition of the validity of propositional formulas. This problem is considered as relatively hard (more precisely, coNP-complete) in theory, but existing algorithms solve very efficiently practical instances of this problem, and the problem is considered as an easy one by

people working in applications. This is not the only example.

There are similar examples of another flavor, like the practical efficiency of linear programming algorithms. Here one finds mathematically interesting results of their average behavior. However, traditional evaluation of the average or Teng-Spielman smooth analysis [13] deal with sets of inputs almost none of which appears in practice. If one accepts Kolmorogov algorithmic vision of randomness, i.e., a string (or other combinatorial construct) is random if its Kolmogorov complexity is close to the maximal value, then one gets another argument that random constructs cannot appear from physical or human activity.

Many people believe that physical processes may produce truly random data. Many years ago, it was somewhere in the 70th, G. M. Adelson-Velsky[2] told me that M. M. Bongard[3] showed, using not very complicated learnability algorithm, that Geiger counter data, that were considered as truly random, can be predicted with a probability definitely higher that 1/2. Who else analyzed physical 'random data' in this way? Notice that standard statistical tests that are used to prove randomness can be easily fooled by simple deterministic sequences, e.g., Champernowne's sequence. Happily, in practice 'sufficiently random' sequences suffice.

The practical inputs are always described in a natural language whose constructs are numerous but incomparably less numerous than arbitrary constructs, so they are not so random.

One may refer to the ideology of modern mathematics. Modern mathematics does not study arbitrary functions, nor arbitrary continuous functions, nor even arbitrary smooth functions. It studies particular, often rather smooth, manifolds on which often, though not always, acts a group with some properties modeling properties inspired by applications in mind.

It is not so evident how to find a structure to study in algorithmic problems, but it is much simpler to see a structure in computations, namely, in sets of runs (executions). One can try to find geometry in these sets. An intuitive sentiment is that any algorithm transforms information, so we can try to find geometry in computations using this or that concept of information.

It is improbable that one approach will work for all types of algorithms that appear in practice. The frameworks we use to study different types of algorithms are different. For example, reactive real-time systems are studied not as data base queries, computer algebra algorithms are studied not in the same way as combinatorial algorithms etc. In this paper I try to look at off-line 'combinatorial' algorithms without defining this class rigorously. Roughly speaking such an algorithm processes

[2]Georgy Maximovich Adelson-Velsky (1922–2014) was a well-known Soviet and Israeli mathematician and computer scientist.

[3]Mikhail Moiseevich Bongard (1924–1971) was a well-known Soviet computer scientist.

a finite 'combinatorial' input accessible from the very beginning, where each bit is 'visible' except maybe some integers that are treated as abstract atoms or 'short' integers with addition and comparison. Examples are string matching, binary convolution, TAUT, shortest path in graphs with integer weights etc.

But algorithms of this vaguely defined class may be very different from the point of view of their analysis. For example, take diagonal algorithms and compare such an algorithm with an algorithm like just mentioned above. One can see that runs of diagonal algorithms are highly diverse, within the same length of inputs we may see a run that corresponds to an execution of a string-matching algorithm, another run that correspond to solving a linear system etc. In the algorithms mentioned above the runs are more or less 'similar'. My first idea was to say that this distinguish practical algorithms from non practical ones. However, E. Asarin immediately drew my attention to interpreters that are quite practical and whose sets of runs are of the same nature that the set of runs of diagonal algorithms. It is interesting that compilers (to which N. Dershowitz drew my attention in the context of a discussion on practical and impractical algorithms some time ago) are in the same class that the mentioned combinatorial algorithms because they do not execute the programs that they transform. But interpreters are not in the same class as the combinatorial algorithms that are under study here. We do not demand that an interpreter diminish the computational complexity of the interpreted algorithm. And the interpretation itself slows down the interpreted algorithm by a small multiplicative constant that we can try to diminish. In some way, the output of the interpreter is a trace of the interpreted algorithm, so their diversity is intrinsic, and the length of their outputs is compared with their time complexity. We consider algorithms whose outputs are 'much shorter' than their time complexity.

2 How to Evaluate Similarity of Computations?

Some syntactic precisions on the representation of runs of algorithms are needed. Suppose that F is an algorithm of bounded computational complexity that has as its inputs some structures (strings, graphs etc.) and whose outputs are also some structures.

By the size of an input we mean not necessarily the length of its bit code but some value that is more intuitive and 'not far' from its bit size. E.g., the number of vertices for a weighted graph, the length of vectors in binary convolution etc. In any case the bit size is polynomially bounded by our size. Thus, for a weighted graph we assume that weights are integers whose size is of the order of logarithm of the number of vertices if the weights are treated as binary numbers or whose size is $O(1)$

if they are treated abstractly.

We mention two very simple examples, namely palindrome recognition and sum of elements of a string over $\mathbb{B}$.

Assume that for the structures under consideration a reasonable notion of size is defined, and the set of all inputs of size n, that are in the domain of F, is denoted by $\boldsymbol{dm}_n(F)$ or $\boldsymbol{dm}$ if F and n are clear from the context. The set of corresponding values of F is denoted $\boldsymbol{rn}_n(F)$ or $\boldsymbol{rn}$. We assume that n is a part of inputs. Below n is fixed and often omitted in the notations.

We look at algorithms from the viewpoint of logic. Though in programming, as well as in logic, any program may be seen as an abstract state machine, there is no terminology that is commonly accepted in logic and programming. For example, what is called variable in programming is not variable in logic; from the point of view of logic it is a function without arguments but that may have different values during the execution of the program. In order to avoid such discrepancy we use logical terminology that was developed by Yu. Gurevich for his Abstract State Machines [3], and may be applicable to any kind of programs. Our framework is not that of Yu. Gurevich machines, we deal with executions of low-level programs seen as some kind of abstract state machines.

An algorithm computes the values of outputs using *pre-interpreted* constants like integers, rational numbers, Boolean values, characters of a fixed alphabet, and pre-interpreted functions like addition, order relations over numbers and other values, Boolean operations. These functions are *static*, i.e., they do not change during the executions of F. The other functions are *abstract* and *dynamic*. The inputs are given by the values of functions (that constitute the respective structure) that F can only read; they are *external* (as well as pre-interpreted functions). The functions that can be changed by F are its *internal functions*, they are subdivided into *output functions* and *proper internal* functions. We assume for simplicity that the output functions are updated only once. Dynamic functions may have arguments, like, e.g., arrays, and we limit ourselves to such functions that have one natural argument. When the argument i in such a function f is fixed, this $f(i)$ can be considered as nullary function, i.e. as a function without arguments. All these functions constitute a vocabulary of the algorithm.

We consider computations only for inputs from a finite set $\boldsymbol{dm}_n(F)$. These computations are represented as sets of traces that we describe below. We can treat such sets abstractly without precise notion of algorithm. However, for better intuitive vision, we describe a simple algorithmic language that gives a general notion of algorithm and that suffices for our examples.

Term is defined as usual, and without loss of generality, we consider non nested terms, i.e., terms whose arguments are only variables if any. *Guard* is a literal.

Update (assignment) is an expression $g := \theta$, where g is an internal function, and θ is a term. Constructors of a program (algorithm) are: update, sequential composition (denoted ;), branching **if** *guard* **then** P **else** P' , where P and P' are programs, **goto**, **halt**. As delimiters we use brackets.

A *state* is an interpretation of the vocabulary of the algorithm. A state is changed by updates in the evident way. The initial state is common for all inputs, we assume that the initial value of any internal function f is symbol $\natural$ that represents *undefined*, is never used in updates, and that $f^{-1}(\natural) = \emptyset$. A *run* is defined as usually as a sequence of states, but we use an equivalent representation of executions as traces.

Given an input $X \in \boldsymbol{dm}_n(F)$ a trace $\boldsymbol{tr}(X)$ is constructed as follows according to the executed operators: update is written as it is in the program; in the case of conditional branching **if** *guard* **then-else** we put in the trace either *guard* or its negation depending on what is true in this trace. For simplicity the initial state and **halt** are not explicitly mentioned in traces, neither **goto**. Thus, a trace is a sequence of updates and guards that are called *events*. The tth event in a trace $\boldsymbol{tr}(X)$ is denoted $\boldsymbol{tr}(X,t)$. These events are *symbolic*. An execution gives values to the internal functions, and thus, an interpretation of any event.

Denote by $\mathbf{t}^*_F(X)$ the time complexity of F for input X, and by $\mathbf{t}_F(n)$ the maximum of these values, i.e., the worst-case time complexity of F over $\boldsymbol{dm}_n(F)$.

For an input $X \in \boldsymbol{dm}_n(F)$ and a time instant t, $1 \le t \le \mathbf{t}^*(X)$, we denote by $f[X,t]$ the value of a internal function f in $\boldsymbol{tr}(X)$ at t, the value is defined recursively together with the recursive definition of trace given just above. If f is not undated at t then $f[X,t] = f[X,t-1]$. If $\boldsymbol{tr}(X,t)$ is of the form $f := g(\eta)$ then $f[X,t] = g(\eta[X,t-1])[X,t-1]$.

Consider two examples.

Palindrome recognition. Inputs are non empty strings of length n over an alphabet $\mathcal{A}$ with $\alpha \ge 2$ characters. For simplicity assume that n is even and set $\nu =_{df} \frac{n}{2}$. We denote the input by w, and the character in the ith position by $w(i)$. We take a straightforward algorithm φ that compares characters starting from the ends and going to the middle of the input. We use % to mark comments, and we omit **halt** that is evident.

Algorithm φ:

% i is a loop counter, r is the output (0 means non palindrome, 1 palindrome)

1: $i := 0$;

2: **if** $i < \nu$ **then** ($i := i + 1$;

if $w(i) = w(n-i+1)$ **then goto** 2 **else** r:=0)

else r:=1

Algorithm φ has two types of traces (one with output 0 and the other with output 1):

$i := 0,\ i < \nu,\ i := i+1,\ w(i) = w(n-i+1),\ \ldots,\ i < \nu,\ i := i+1,$
$w(i) = w(n-i+1),\ i < \nu,\ i := i+1,\ w(i) \neq w(n-i+1)$, r:=0

$i := 0,\ i < \nu,\ i := i+1,\ w(i) = w(n-i+1),\ \ldots,\ i < \nu,\ i := i+1,$
$w(i) = w(n-i+1),\ i \geq \nu$, r:=1

A trace of the first type may have different lengths starting from 5, but the length of the trace of the second type is always the same.

For a string $aabaaa$ with $a \neq b$, if in the respective trace we replace the internal functions, as well as n, by their values we can write:

$i := 0,\ 0 < 3,\ i := 0+1,\ w(1) = w(6),\ 1 < 3,\ i := 1+1,\ w(2) = w(5),$
$2 < 3,\ i := 2+1,\ w(3) \neq w(4)$, r:=0

Sum modulo 2 of bits of a string. Inputs are strings of the set $\mathbb{B}^n$.
Algorithm σ:

% x is input, r is output, i is a loop counter, s is an intermediate value
1: $i := 0;\ s := 0;$ %Initialization
2: **if** $i < n$ **then** $i := i+1;\ s := s + x(i)$; **goto** 2
3: **else** $r := s$ % case $i \geq n$

All traces of σ are 'symbolically' the same (the algorithm is oblivious), for clarity we put in an event the value of i acquired before this event:

$i := 0,\ s := 0,\ 0 < n,\ i := 0+1,\ s := s + x(1),\ 1 < n,\ i := 2,$
$s := s + x(2),\ \ldots,\ n-1 < n,\ i := n,\ s := s + x(n),\ n \geq n,\ r := s$

Remark. For Boolean circuits we can also produce traces that are even simpler, as a Boolean circuit is a non branching oblivious algorithm. Such a trace consists of updates, each one being an application of the Boolean function attributed to a vertex of the circuit, to the values attributed to its predecessors.

Denote by $\boldsymbol{Tr}_n$ the set of all traces for inputs from $\boldsymbol{dm}_n$. The length $|\boldsymbol{tr}(X)|$ of a trace $\boldsymbol{tr}(X)$, $X \in \boldsymbol{dm}_n$, is the number of occurrences of events in it, i.e., the time complexity $\mathbf{t}^*_F(X)$.

2.1 A Syntactic Similarity of Traces

A straightforward way to compare two traces is the following one. We look in $\boldsymbol{tr}(X)$ and $\boldsymbol{tr}(Y)$ for a longest common subsequence (we tacitly assume that some equivalence between events is defined), and take as a measure of similarity the size of the rest. More precisely, if S is the longest common subsequence then we take as measure the value $|\boldsymbol{tr}(X)| + |\boldsymbol{tr}(Y)| - 2|S|$, where $|S|$ is the size (the number of

elements) of a sequence S. This measure is something like the size of symmetric difference of two sequences.

We can go further, and to take into account only causal order in what concerns the order of events, and to permit a renaming of proper internal functions and their values. The causal order is defined as follows. If the function updated or used (in the case of guard verification) in an event e depends on a function updated earlier in an event e' then e' *causally precedes* e. Taking a transitive closure of this relation we get *causal order* between events in a given trace. This generalization is too technical (details can be found in [12]), and as I cannot give examples of realistic applications, it is just mentioned as a theoretical possibility.

The measure introduced above gives a pseudo-metric (it is like metric except that two different traces may have zero distance; in our case the zero distance relation is an equivalence) over traces. As the trace space $\boldsymbol{Tr}_n$ is clearly compact, this metric permits to define epsilon-entropy [5] on it. This entropy is defined as follows. For a given ε (in our case it is a natural number) take an ε-net of minimal size such that the ε-balls centered at the points of the net cover all the space. Then $\log s$, where s is the size of this net, is the ε-entropy. It gives the size complexity of the ε-approximation of the space, or to say it differently, how much information one needs to have, in order to describe an element of the space with accuracy ε.

Consider our examples.

Trace space of φ. We define similarity as follows (it is a rather general way to define it). First, in the right-hand side of each update $f := \theta$ replace all proper internal functions of θ by their values. In guards replace all internal functions by their values. We get as transformed events the expressions: $i := m$, where $m \in \mathbb{N}$ and $m = (\dots((0+1)+1)+\dots+1)$, $m < \nu$, $m \geq \nu$, $w(m) = w(n-m+1)$, $w(m) \neq w(n-m+1)$, $r = 0$, $r = 1$. As similarity (we refer to it as 'weak similarity') we take the syntactic equality of these transformed events.

With this similarity we have $(\nu+1)$ different (*classes of similar*) traces (ν classes with $r = 0$ at the end, and 1 class with $r = 1$): denote by P_k traces with $(k-1)$ equalities and one inequality in the k comparison, $1 \leq k \leq \nu$, and by P the only trace with $r = 1$. The distance between P_k and P_l is $3|k-l|$, and between P_k and P is $3(\nu-k)+2$. If we take $\varepsilon = 2$ then ε-net should include all the traces but, however, it is of size $(\log n + \mathcal{O}(1))$. If we take $\varepsilon = 3p$, $p \in \mathbb{N}$, then as an ε-net we can take each pth trace ordered according to their lengths; hence, $3p$-entropy is of size $\lceil \frac{n}{2p} \rceil = \lceil \frac{\nu}{p} \rceil$ (maybe plus 1).

The situation changes if we take stronger similarity. We say that $w(m) = w(n-m+1)$ and $w(m') = w(n-m'+1)$ are similar if $m = m'$ (as before) and the respective values of inputs are the same $w(m) = w(m')$ (for $\neq$ we demand also $w(n-m+1) = w(n-m'+1)$). In this case the trace space becomes of exponential

size. We illustrate this kind of similarity for algorithm σ.

Trace space of σ. We define similarity of event of the form $i := i + 1$ and of the form $i < n$ as in the previous case: values of $(i + 1)$ in similar events of the form $i := i + 1$ and the value of i in similar events of the form $i < n$ should be equal. Two events of the form $s := s + x(i)$ are similar if the values of i, as well as of the acquired values of s, are equal. Any string from $\mathbb{B}^n$ may be a string of consecutive values s starting from $s := 0 + x(1)$ that equals to $x(1)$. Thus the set of traces of σ with this similarity and our metric divided by 2 is isometric to the Boolean cube $\mathbb{B}^n$ with Hamming metric. This space is studied in the coding theory, and I cannot say more than can be found there.

Unfortunately, the metric spaces in the examples above do not say much about the advancement of the algorithm towards the result. If we take spaces of traces up to some time instant and their dynamics with growing time, it does not help much neither. Moreover, the size of the space $\boldsymbol{Tr}_n$ is bounded by $|\boldsymbol{dm}_n|$, and does not depend on the complexity of F, and this is also a shortcoming of this approach.

2.2 Remark on Kolmogorov Complexity Approach

Why not to measure distance between traces on the basis of Kolmogorov complexity? This question was put by some of my colleagues.

A direct application of Kolmorogov algorithmic entropy [4] to measure similarity of traces does not give results corresponding to our intuition. Indeed, in [4] Kolmogorov defines entropy as conditional complexity $\boldsymbol{K}(\alpha|\beta)$. Similarity of structures α and β may be measured as $\mathfrak{K}(\alpha, \beta) = \boldsymbol{K}(\alpha|\beta) + \boldsymbol{K}(\beta|\alpha)$. This is not a metric, strictly speaking, however, we call this function $\mathfrak{K}$-*distance* as it has a flavor of intuitive distance-like measure.

Denoting by $|F|$ and $|X|$ binary lengths of respectively F and X we get

$\boldsymbol{K}(\boldsymbol{tr}(X)/\boldsymbol{tr}(Y)) \leq |F| + \boldsymbol{K}(X/Y) + O(1) \leq |F| + |X| + O(1)$.

This formula follows from an observation that X and F are sufficient to calculate the trace $\boldsymbol{tr}(X)$. Thus, whatever be an algorithm F and whatever be its computational complexity, the $\mathfrak{K}$-distance between traces from $\boldsymbol{Tr}_n$ is not greater than $O(|X|)$ that we assume, for simplicity, to be $O(n)$. On the other hand, given a minimal length program G that computes $\boldsymbol{tr}(X)$ from $\boldsymbol{tr}(Y)$ (thus, $|G| = \boldsymbol{K}(\boldsymbol{tr}(X)/\boldsymbol{tr}(Y))$) one can get X from Y as follows: from Y one computes $\boldsymbol{tr}(Y)$ using F (whose size is a constant), then using G one computes $\boldsymbol{tr}(X)$ and finally extracts X from $\boldsymbol{tr}(X)$ with the help of a simple fixed program, say E, whose length is a constant (without loss of generality, we can assume that the input is reproduced at the beginning of each trace). All this gives (we put 'absolute' constants $|F|$, $|E|$ in the last $O(1)$)

$\boldsymbol{K}(X|Y) \leq |F| + |G| + |E| + O(1) \leq \boldsymbol{K}(\boldsymbol{tr}(X)/\boldsymbol{tr}(Y)) + O(1)$.

We assume that the cardinality of binary codes of $\boldsymbol{dm}_n(F)$ is at least 2^n (hence, almost all inputs have Kolmogorov complexity $n - o(n)$), then the chain rule for Kolmogorov complexity (e.g., see [4]) for almost all X, Y gives

$\boldsymbol{K}(X|Y) = \boldsymbol{K}(X,Y) - \boldsymbol{K}(Y) - O(\log \boldsymbol{K}(X,Y)) \geq n - c\log n$

for some constant $c > 0$.

Together with the previous formula this gives a lower bound for $\boldsymbol{K}(\boldsymbol{tr}(X)/\boldsymbol{tr}(Y))$ that shows that $\mathfrak{K}$-distance is almost always of order of n that can hardly be seem as satisfactory for evaluation of similarity of traces from $\boldsymbol{Tr}_n$.

What is said above, does not exclude that other types of Kolmogorov style complexity could work better (e.g., a more general notion of entropy [11] is based on inference complexity.). In particular, resource bounded complexity approaches may prove to be productive if we find a 'good' description of information extracted by algorithm as datum (structure); however, this remains an open question.

2.3 Similarity via Entropy of Partitions

In this subsection we outline another approach to measure similarity of traces. It refers to the classical entropy of partitions. We use partitions of the inputs. For this reason a probabilistic measure over the inputs is needed. Such a measure is a technical means, so there is no evident way to introduce it. We do it taking into account an intuition related to the evolution of the 'knowledge' of the algorithm. When an algorithm F starts its work it 'knows' nothing about its output. So all values from $\boldsymbol{rn}_n$ are equiprobable.

Let $M = |\boldsymbol{rn}_n(F)|$. As any of these M values is equiprobable (imagine that an input is given by an adversary who plays against F), we set $\boldsymbol{P}_n(F^{-1}(Y)) = \dfrac{1}{M}$ for all $Y \in \boldsymbol{rn}_n(F)$, and inside $F^{-1}(Y)$ the measure is uniform as the algorithm a priori has no preferences. In particular, if F is a 2-valued function, say $\boldsymbol{rn}(F) = \mathbb{B}$, then its domain is partitioned into two sets $F^{-1}(0)$ and $F^{-1}(1)$ with the same measure $1/2$ of each set. E.g., for palindromes we the measure of a palindrome is $\frac{1}{2\alpha^\nu}$ and that of a non palindrome is $\frac{1}{2(\alpha^n - \alpha^\nu)}$. There is nothing random in the situation we consider, we wish only to model the evolution of the knowledge of an algorithm during its work. So this way to introduce a measure may be not the best one.

Suppose that f is updated at t and $f[X,t] = v$. How to describe the knowledge acquired by F via this event at t that gives $v = f[X,t]$? This value v may be acquired by f in different traces, even several times in the same trace, and at different time instants. The traces are not 'synchronized' in time, however, we can compare events, as in subsection 2.1, due to this or that similarity relation, that is determined by our goal and our vision of the situation. Notice that formally speaking similarity is a

relation between pairs (X, t), where $X \in \boldsymbol{dm}(F)$ and $1 \leq t \leq \mathbf{t}^*_F(X)$. Similarity can be defined not only along the lines described in subsection 2.1. One may think about quite different ways. Just to give an idea, one can, for example, consider as similar events corresponding to the kth execution of the same command of the program with or without demanding equality of these or that values. Or one can permit renaming of internal function as it was mentioned at the beginning of subsection 2.1.

Suppose that some similarity relation $\sim$ is fixed.

To compare traces we attribute to each event of a trace a partition of inputs. Thus, to each trace there will be attributed a sequence of partitions. Taking into account that the set of inputs is a space with probabilistic measure we can define a distance between partitions and furthermore a distance between sequences or sets of partitions.

For any input X and an instant t, $1 \leq t \leq \mathbf{t}^*(X)$, denote by $\boldsymbol{sm}(X, t)$ all the inputs X' such that $(X, t) \sim (X', t')$ for some t'. Clearly, $X \in \boldsymbol{sm}(X, t)$. Denote by $\boldsymbol{pt}(X, t)$ the partition of $\boldsymbol{dm}_n$ into $\boldsymbol{sm}(X, t)$ and its complement that we denote $\boldsymbol{sm}(X, t)^c =_{df} \boldsymbol{dm}_n \setminus \boldsymbol{sm}(X, t)$.

Thus, each input X determines a *sequence* $(\boldsymbol{pt}(X, t))_t$ or a set $\{\boldsymbol{pt}(X, t)\}_t$ *of partitions* of $\boldsymbol{dm}_n(F)$. These constructions, namely sequence or set, provide different opportunities for further analysis, e.g., we can define distance between metric spaces, e.g., see [1, ch. 7].

For measurable partitions of a probabilistic space $\mathcal{P} = (\Omega, \Sigma, P)$ one can define entropy (no particular technical constraints are needed in our case of finite sets), see [8] or books like [7].

Let $\mathcal{A}$ and $\mathcal{B}$ be measurable partitions of $\mathcal{P}$ (in our situation all the sets are measurable).
Entropy $H(\mathcal{A})$ and conditional entropy $H(\mathcal{A}/\mathcal{B})$ are defined as

$$H(\mathcal{A}) = -\sum_{A \in \mathcal{A}} P(A) \log P(A),\ H(\mathcal{A}/\mathcal{B}) = -\sum_{B \in \mathcal{B}, A \in \mathcal{A}} P(A \cap B) \log \frac{P(A \cap B)}{P(B)} \quad (1)$$

The conditional entropy permits to introduce Rokhlin metric [8] between partitions:

$\boldsymbol{\rho}(\mathcal{A}, \mathcal{B}) = H(\mathcal{A}/\mathcal{B}) + H(\mathcal{B}/\mathcal{A}) = 2H(\mathcal{A} \vee \mathcal{B}) - H(\mathcal{A}) - H(\mathcal{B})$,

(here $\mathcal{A} \vee \mathcal{B}$ is common refinement of partitions $\mathcal{A}$ and $\mathcal{B}$, that is the partition formed by all pairwise intersection of sets of $\mathcal{A}$ and $\mathcal{B}$).

There are other ways to introduce distance between partitions, e.g., see [7, 4.4], so one can take or invent maybe more productive metrics or entropy-like measures.

Unfortunately, the combinatorial difficulties of estimating such distancies are discouraging, they do not justify what we get form them. We illustrate this for the palindrome recognition algorithm φ.

Denote $w^{=}(1..k)=_{df}\{w : \bigwedge_{1\le i\le k} w(i) = w(n-i+1)\}$ (the set of words whose prefix of length k permits to extend it to a palindrome), denote by $w^{\neq}(1..k)$ the complement of $w^{=}(1..k)$; in particular, $w^{=}(k..k) = \{w : w(k) = w(n-k+1)\}$ and $w^{\neq}(k..k) = \{w : w(k) \neq w(n-k+1)\}$. Probabilities are easy to calculate (we use them in the next subsection), here $1 \le k < m \le \nu$:

$$\boldsymbol{P}(w^{=}(1..k) \cap w^{\neq}(k+1..m)) = \frac{\alpha^{\nu-m}(\alpha^{m-k}-1)}{2(\alpha^{\nu}-1)}, \tag{2}$$

$$\boldsymbol{P}(w^{=}(1..k)) = \frac{1}{2} + \frac{\alpha^{\nu-k}-1}{2(\alpha^{\nu}-1)}, \quad \boldsymbol{P}(w^{\neq}(1..k)) = \frac{\alpha^{\nu-k}(\alpha^{k}-1)}{2(\alpha^{\nu}-1)}. \tag{3}$$

(We omit technical details, the role of the formulas is illustrative.)

However, when we try to calculate the distance between partitions, take for example $\boldsymbol{\rho}(\pi^{=}(k), \pi^{=}(m))$, where $\pi^{=}(s) = (w^{=}(1..s), w^{\neq}(1..s))$, we arrive at a formula that is a sum of several expressions like $\left(\frac{1}{2} + \frac{\alpha^{\nu-s}-1}{2(\alpha^{\nu}-1)}\right) \log \left(\frac{1}{2} + \frac{\alpha^{\nu-s}-1}{2(\alpha^{\nu}-1)}\right)$, that is hard to evaluate. And what is worse the result is not very instructive, e.g.,

$$\boldsymbol{\rho}(\pi^{=}(1), \pi^{=}(\nu)) \approx \begin{cases} 0.9 & \text{if } \alpha = 2 \\ 0.67 & \text{if } \alpha = 3 \\ 0.6 & \text{if } \alpha = 4 \end{cases}$$

Technical combinatorial difficulties do not discard the idea of geometry of spaces of events or traces, the point is to find a geometry and its interpretation that really deepens our understanding of algorithms and problems.

2.4 The Question of Information Convergence

Now we discuss how similarity of events may serve to evaluate the rate of convergence of a given algorithm towards the result.

Among the first ideas that come to mind is the following one. The result $F(X)$ for an input X is represented in terms of a partition of $\boldsymbol{dm}_n(F)$ into $F^{-1}(F(X))$ and its complement $F^{-1}(F(X))^c$. The current knowledge of F at an instant t is in its current event that also defines a partition denoted above $\boldsymbol{pt}(X,t)$.

How this local knowledge represented by $\boldsymbol{pt}(X,t)$, is related to the partition $(F^{-1}(F(X)), F^{-1}(F(X))^c)$ mentioned just above? A possible answer is: compare $\boldsymbol{pt}(X,t)$ (the local knowledge at an instant t in terms of partitions) with the partition $(F^{-1}(F(X)), F^{-1}(F(X))^c)$. This idea can be a priori implemented differently, for example, in terms of conditional probabilities or in terms of conditional entropies.

If we try to apply this idea to any of our examples, we find that the commands that control the loops give trivial partition $(\boldsymbol{dm}, \emptyset)$ because they are in all traces, and these events give nothing useful. So we take only events that process inputs.

Consider φ (the example of palindromes). Using (2), (3) we get, omitting technicalities and taking sufficient approximations:

$$\boldsymbol{P}(r = 1|w^{=}(1..k)) \approx \frac{1}{1 + A(k)}, \quad \boldsymbol{P}(r = 0|w^{=}(1..k)) \approx \frac{A(k)}{1 + A(k)}, \tag{4}$$

where $A(k) = \alpha^{-k} - \alpha^{-\nu}$. The probabilities (4) do not reflect our information intuition that φ converges to the result when $k \to \nu$ as one goes to 1, and the other to 0. But if we take the respective entropy

$$-\frac{1}{1 + A(k)} \log \frac{1}{1 + A(k)} - \frac{A(k)}{1 + A(k)} \log \frac{A(k)}{1 + A(k)}, \tag{5}$$

we see that it goes to 0, thus, to total certainty.

Consider σ (sum modulo 2). The similarity that we used for the trace space of σ in subsection 2.1 we call here *weak similarity*. Denote $\sigma^{-1}(a)$ by $\sigma = a$. Clearly, $|\sigma = 0| = |\sigma = 1| = 2^{n-1}$, $\boldsymbol{P}(\sigma = a) = \frac{1}{2}$, and $\boldsymbol{P}$ is a uniform distribution over $\mathbb{B}^n$.

Denote by $S_k(a)$, where $a \in \mathbb{B}$, the set $\{x : s + x(k) = a\}$; it is a set of type $\boldsymbol{sm}(X, t)$. For $k < n$ and all $a, b \in \mathbb{B}$ we have

$$\boldsymbol{P}(S_k(a)) = \frac{|S_k(a)|}{2^n} = \frac{2^{n-1}}{2^n} = \frac{1}{2}, \quad \boldsymbol{P}(S_n(a) \cap S_k(b)) = \frac{1}{4} \tag{6}$$

$$\boldsymbol{P}(\sigma = a|S_k(b)) = \frac{\boldsymbol{P}(S_n(a) \cap S_k(b))}{\boldsymbol{P}(S_k(b))} = \frac{1}{2} \tag{7}$$

We see that nothing changes with advancing of time, i.e., with $k \to n$. If we apply formula (1) for conditional entropy, it gives a constant. Hence, with this similarity, we do not see any convergence of σ to the result.

Let us try a stronger similarity: we say that a event $s := s + x(k)$ is (strongly) similar to $s := s + x'(k)$ if $x(i) = x'(i)$ for all $1 \le i \le k$. Denote by $Z(\chi)$, where $\chi \in \mathbb{B}^k$, $1 \le k < n$, the set of inputs x such that for event $s := s + x(k)$ there holds $x(i) = \chi(i)$ for $1 \le i \le k$; this set describes the set of inputs of strongly similar events. We have $|Z(\chi)| = 2^{n-k}$, and $|(\sigma = a) \cap Z(\chi)| = 2^{n-k-1}$, thus, $\boldsymbol{P}(Z(\chi)) = 2^{-k}$ and $\boldsymbol{P}((\sigma = a) \cap Z(\chi)) = 2^{-k-1}$. So the measure of the space of continuations of the known part of the input diminishes. The respective term in conditional entropy (1) gives $-2^{-k-1} \log \frac{1}{2} = 2^{-k-1}$ that is encouraging but the term related to $Z(\chi)^c$ (notice that $\boldsymbol{P}(Z(\chi)^c) = 1 - 2^{-k}$ and $\boldsymbol{P}((\sigma = a) \cap Z(\chi)^c) = \frac{1}{2} - 2^{-k-1}$) bring us back to values that practically do not diminish. All this means only that the classical entropy does not work, and we are to seek for entropy-like measures that truly reflect our intuition.

The partition based measures of convergence look promising. However, one can say that the number of partitions is limited by an exponential function of $|\boldsymbol{dm}|$. So if the complexity is very high, e.g., hyper-exponential, then there is 'not enough' of partitions to represent the variety computations. In fact, we think about certain class of problems that are outlined in the next section for which there seems to be 'enough' of partitions. As for high complexity problems, another interpretation of input data is needed. Some hints are given in the next section 3.

3 On the Structure of Problems

Here are presented examples of problems together with a reference to their inner structure that may be useful for further study of information structure of computations and that of problems themselves along the lines discussed in the paper. The examples below concern only simple 'combinatorial problems'. The instances of these problems are finite graphs (in particular, strings, lists, trees etc.) whose edges and vertices may be supplied with additional objects that are either abstract atoms with some properties or strings. As examples of problems that are not in this class one can take problems with exponential complexity like theory of real addition or Presburger arithmetics. The problems in the examples below are divided into 'direct' and the respective 'inverse' ones.

Direct Problems

(A1) *Substring verification.* Given two strings U, W over an alphabet with at least two characters and a position k in W, to recognize whether $U = W(k, k+1, \ldots, k+|U|-1)$, i.e., whether U is a substring of W from position k.

(A2) *Path weight calculation.* Given a weighted (undirected) graph and a path, calculate the weight of the path.

(A3) *Evaluation of a Boolean formula for a given value of variables.* Given a Boolean formula Φ and a list X of values of its variables, calculate the value $\Phi(X)$ for these values of variables.

(A4) *Permutation.* Given a list of elements and a permutation, apply the permutation to the list.

(A5) *Binary convolution (or binary multiplication).* For simplicity we consider binary convolution that represents also the essential difficulties of multiplication. Given 2 binary vectors or strings $x = x(0) \ldots x(n-1)$ and $y = y(0) \ldots y(n-1)$ calculate

$$z(k) = \sum_{i=0}^{i=k} x(i)y(k-i), \qquad 0 \le k \le (2n-2),$$

assuming that $x(i) = y(i) = 0$ for $n-1 < i \le (2n-2)$.

Inverse Problems

(B1) *String matching.* Given two strings W and U over an alphabet with at least two characters, to recognize whether U is a substring of W.

(B2) *Shortest path.* Given a weighted (undirected) graph G and its vertices u and v, find a shortest path (a path of minimal weight) from u to v.

(B3) *Propositional tautology TAUT* Given a propositional formula Φ, to recognize whether it is valid, i.e., is true for all assignment of values to its variables. A variant that is more interesting in out context is MAX-SAT: given a CNF (conjunctive normal form), to find the longest satisfying assignment of variables, i.e. an assignment that satisfies the maximal number of clauses.

(B4) *Sorting.* Given a list of elements of a linearly ordered set, to find a permutation that transforms it into an ordered list.

(B5) *Factorization.* Given z, to find x and y whose convolution or product (in the case of multiplication) is z.

Examples (A1)–(A4) give algorithmic problems whose solution, based directly on their definitions, is practically and theoretically the most efficient. Each solution consists in a one-directional walk through a simple data structure making, again rather simple, calculations – something that is similar to scalar product calculation.

In (A1) the structure is a list $(k, k+1, \ldots, k+|U|-1)$, and while walking along it, we calculate conjunction of $U(i) = W(i)$ for $k \leq i < (k+|U|)$ until i reaches the last value or $false$ appears.

Example (A2) is similar, where the list of vertices constituting the linear structure is explicitly given, and the role of conjunction of (A1) is played by addition.

The structures used in (A3) depend on the representation of Φ and of the distribution of values of its variables. In any case one simple linear structure does not suffice here. Suppose Φ is represented in DNF (Disjunctive Normal Form), i.e., as a disjunction of conjunctions. This can be seen as a list of lists of literals, and a given distribution of values is represented as an array corresponding to a fixed order of variables. So given a variable, its value is immediately available. Thus, the representation of values is a linear structure, and DNF is a linear structure of linear structures. It is more interesting to suppose that Φ is a tree. Then we deal with the representation of values and with a walk, again without return, through a tree with calculating the respective Boolean functions at the vertices of the tree. So we see another simple basic structure, namely a tree.

In example (A4), while walking through two given lists, namely a list of elements and a permutation, a third list (a list of permuted elements) is constructed.

Example (A5) is more complicated, and the definition of problem does not give an algorithm that may be considered as the best; it is known that the direct algorithm for convolution is not the fastest one. Here there is no search, and for this reason

this problem is put in the class of direct ones, but there is a non-trivial intermixing of data. One may see the description of the problem as a code of data structures to extract, and then to calculate the resulting values by simple walks through these data structures. The number of the data structures to extract is quadratic. In order to find a faster algorithm, one should ensure the same intermixing but using different data structures and operations.

Examples (B1)–(B5) give algorithmic problems of search among substructures coded in inputs. The number of these substructures, taken directly from the definition, is quadratic for (B1), and exponential for (B2)–(B5). The substructures under search should satisfy conditions that characterize the corresponding direct problem. More complicated problems code substructures not so explicitly as in examples (B1)–(B5). To illustrate this, take e.g., quantifier elimination algorithm for the formulas of the theory of real addition, not necessarily closed formulas. Here it is not evident how to define the substructures to consider. The quantifier elimination by any known algorithm produces a good amount of linear inequalities that are not in the formula. So the formula codes its expansion that is more than exponentially bigger as compared with the initial formula itself.

Whatever be the mentioned difficulties, intuitively the substructures and constraints generated by a problem may be viewed as an extension of the set of inputs. And in this extended set one can introduce not only measure but also metrics that give new opportunities to analyze the information contents and the information evolution. One can see that the cardinality constraints on the number of partitions that was mentioned in subsections 2.3 and 2.4 is relaxed. This track has not been yet studied, though one observation can give some hint to how to proceed. When comparing substructures it seems productive to take into account its context, i.e., how it occurs in the entire structure. For example, we can try to understand the context of an assignment A of values to variables of a propositional formula Φ in the following manner. Pick up a variable x_1 and its value v_1 from A and calculate the result $\Phi(x_1, v_1)$ of the standard simplification of Φ where x_1 is replaced by Boolean value v_1. This resulting residue formula gives some context of (x_1, v_1). We can take several variables and look at the respective residue as at a description of context. This or that set of residues may be considered as a context of A. It is just an illustration of what I mean here by 'context'.

A metric over substructures may distinguish 'smooth' inputs from 'non-smooth' ones, and along this line we may try to distinguish practical inputs from non practical ones. Though it is not so evident.

For some 'simple' problems such a distinction is often impossible. It looks hard to do for numerical values. The set of such values often constitutes a variety with specific properties that may represent realistic features but almost all elements of

such varieties will never appear in practical computations. An evident example is binary multiplication. Among 2^{128} possible inputs of multiplication of 64-bit numbers most of them will never be met in practice.

A remark on the usage of linguistical frameworks

One more way to narrow the sets of inputs to take into account, is a language based one. Inputs describing human constructions, physical phenomena, and their properties, when they are not intended to be hidden, have descriptions in a natural language. Encrypted data are not of this nature. So for input data with non hidden information, we have a grammar that generates these inputs. Such a grammar dramatically reduces the number of possible inputs and, what is more important, defines a specific structure of inputs. The diminishing of the number of generated inputs is evident. For example, the number of 'lexical atoms' of the English is not more than 250 thousands, i.e., not more than 2^{18}. On the other hand, the number of strings with at most, say, 6 letters is at least $26^2 = 2^{6 \cdot \log 26} > 2^{6 \cdot 4.7} > 2^{28}$ (here 26 is the number of letters in English alphabet). The set of cardinality 2^{18} is tiny with respect to the set of cardinality 2^{28}. If one tries to evaluate the number of phrases, the difference becomes much higher.

But this low density of 'realistic' inputs does not help much without deeper analysis. The particular structure of inputs may help to devise algorithms more efficient over these inputs than the known algorithms over all inputs; there are examples, however not numerous and mainly of more theoretical value. So if one wishes to describe practical inputs in a way that may help to devise efficient algorithms, one should find grammars well aimed at the representation of particular structures of inputs. This point of view does not go along traditional mathematical lines when we look for simple and general descriptions, that are usually too general to be adequate to the computational reality.

The grammar based view of practical inputs may influence theoretical vision of a problem. For example, consider the question of quality of encryption. The main property of any encryption is to be resistant to cryptanalysis. Notice that linguistic arguments play an essential role in practical cryptanalysis. In reality the encryption is not applied to all strings, it mostly deals only with strings produced by this or that natural language, often rather primitive. Thus, there are relations defined over plain texts. E.g., some substrings are related as subject-predicate-direct compliment, etc. A good encryption should not leave traces of these relations in the encrypted text. What does it mean? Different precisions come to mind. A simple example: let P be a predicate of arity 2 defined over plain texts, and its arguments be of small bounded size. Take concrete values A and B of arguments of P. Assume that we

introduced a probabilistic measure on all inputs (plain texts), and hence we have a measure of the set S^+ of inputs where $P(A, B)$ holds and of its complement S^-. Now suppose that we have chosen a predicate Q over 'substructures' of encrypted texts (I speak about 'substructures' to underline that the arguments of Q are not necessarily substrings, as for P), again simple to understand. Denote by E^+ the set of encrypted texts for which Q is true for at least one argument and by E^- its complement. The encryption well hides $P(A, B)$ if the measures of all 4 sets $(S^\alpha \cap E^\beta)$, where $\alpha, \beta \in \{+, -\}$, are very 'close'. This example gives only an idea but not a productive definition.

However, in order to find grammars that help to solve efficiently practical problems 'semantical' nature of sets of practical inputs should be studied.

Conclusion

The considerations presented above are very preliminary. The crucial question is to define information convergence of algorithms, not necessarily of general algorithms, but at least of practical ones.

One can imagine also other ways of measuring similarity of traces. We can hardly avoid syntactical considerations when keeping in mind the computational complexity. However semantical issues are crucial, and may be described not only in the terms chosen in this paper.

The analysis of philosophical question of relation of determinism versus uncertainty in algorithmic processes could clarify the methodology to choose. Here algorithmic process is understood at large, not necessarily as executed by a computer. Though the process is often deterministic, and if we adhere to determinism then it is always deterministic, at a given time instant, when it is not yet accomplished, we do not know with certainty the result, though some knowledge has been acquired. The question is: what is or how to formalize the knowledge (information) that the algorithm acquires after each step of its execution?

Acknowledgments

I am thankful to Edward Hirsch and Eugene Asarin for their remarks and questions that stimulated this study. I am also grateful to anonymous referees for their remarks and questions that considerably influenced the final text.

The research was partially supported by French *Agence Nationale de la Recherche* (ANR) under the project EQINOCS (ANR-11-BS02-004) and by by Government of the Russian Federation, Grant 074-U01.

References

[1] D. Burago, Yu. Burago, and S. Ivanov. *A Course in Metric Geometry*, volume 33, Graduate Studies in Mathematics. Americal Mathematical Society, Providence, Rhode Island, 2001.

[2] M. Fischer and M. Rabin. Super-exponential complexity of presburger arithmetic. In *Complexity of Computation, SIAM-ASM Proceedings, vol. 7*, pages 27–41, 1974.

[3] Y. Gurevich. Evolving algebra 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–93. Oxford University Press, 1995.

[4] A. N. Kolmogorov. On the Logical Foundations of Information Theory and Probability Theory. *Probl. Peredachi Inf.*, 5(3):3–7, 1969. In Russian. English translation in: Problems of Information Transmission, 1969, 5:3, 1–4, or in: Selected Works of A.N. Kolmogorov: Volume III: Information Theory and the Theory of Algorithms (Mathematics and its Applications), Kluwer Academic Publishers, 1992.

[5] A. N. Kolmogorov and V. M. Tikhomirov. ε-entropy and ε-capacity of sets in function spaces. *Uspekhi Mat. Nauk*, 14(2(86)):3–86, 1959. In Russian. English translation in: Selected Works of A.N. Kolmogorov: Volume III: Information Theory and the Theory of Algorithms (Mathematics and its Applications), Kluwer Academic Publishers, 1992.

[6] O. B. Lupanov. The synthesis of contact circuits. *Dokl. Akad.Nauk SSSR*, 119:23–26, 1958.

[7] N. F. G. Martin and J. W. England. *Mathematical Theory of Entropy.* Addison-Wesley, Reading, Massachusetts, 1981.

[8] V. A. Rokhlin. Lectures on the entropy theory of measure-preserving transformations. *Russian Math. Surveys*, 22(5):1–52, 1967.

[9] C. Shannon. The synthesis of two-terminal switching circuits. *Bell System Technical Journal*, 28(1):59–98, 1949.

[10] A. Slissenko. Complexity problems of theory of computation. *Russian Mathematical Surveys*, 36(6):23–125, 1981. Russian original in: *Uspekhi Matem. Nauk*, 36(2):21–103, 1981.

[11] A. Slissenko. On Measures of Information Quality of Knowledge Processing Systems. *Information Sciences: An International Journal*, 57–58:389–402, 1991.

[12] A. Slissenko. On Entropy in Computations. In *Proc. of the 8th Intern. Conf. on Computer Science and Information Technology (CSIT'2011), September 26–30, 2011, Yerevan, Armenia. Organized by National Academy of Science of Armenia*, pages 25–30. National Academy of Science of Armenia, 2011. ISBN 978-5-8080-0797-0.

[13] D. A. Spielman and S.-H. Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *J. ACM*, 51(3):385–463, 2004.

Received \jreceived