Towards Analysis of Information Structure of Computations[*]

Anatol Slissenko
Laboratory for Algorithmics, Complexity and Logic (LACL),
University Paris East Créteil (UPEC), 61 av. du Gnl de Gaulle
94010, Créteil France
anatol.slissenko@sfr.fr

Abstract: The paper presents considerations how one can try to analyze computations, and maybe computational problems from the point of view of information contents and information evolution. The considerations are rather preliminary. The 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 counter-productive from the point of view of applications. The present text discusses mainly what classical notions of information/entropy might give for analysis of computations. The classical notions of information/entropy seem to be insufficient. In order to better understand the problem, a philosophical discussion of the the essence and relation of knowledge/information/uncertainty in algorithmic processes (not necessarily executed by computer) may be useful.

Keywords: computation, problem, structure, partition, entropy, metric

Introduction

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 ($2^n/n$) and there is an algorithmic method to find such a realization for a given 'random' function. 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 may 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 decidability of the theory of real addition is exponential. 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

[*]Partially supported by French "Agence Nationale de la Recherche (ANR)" under the project EQINOCS (ANR-11-BS02-004)

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 $f(x) = 0$.

Take such an algorithm f whose worst-case complexity is exponential, and of order of φ. Within a reasonable algorithmic framework one may consider that the upper bound complexity of f is φ and its worst-case complexity is at least $\theta \cdot \varphi$ for some positive constant $\theta < 1$. 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 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 construct cannot appear from physical or human activity. The practical inputs are always described in a natural language whose constructs are numerous but incomparably less numerous than arbitrary constructs.

One may refer to the ideology of modern mathematics as theoretical computer science is a rigorous science. 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 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 look 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, the framework of study of reactive real-time systems is different from that of data base quires, 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 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 and similar.

But algorithms of this vaguely defined class may appear to be quite different from the point of view of their analysis. For example, take diagonal algorithms and compare such an algorithm with an algorithm from 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 inputs some time ago) are in the same class that the mentioned combinatorial algorithms because they do not execute the programs 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.

1. 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 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 weight are treated as binary numbers or whose size is $O(1)$ if they are treated abstractly.

We mention mainly 2 very simple examples, namely palindrome recognition and binary convolution where the inputs are strings, in particular 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 $\boldsymbol{dm}_n$ if F is clear from the context. The set of corresponding values of F are denoted $\boldsymbol{rn}_n(F)$ or $\boldsymbol{rn}_n$. We assume that n is also a part of inputs. We also assume that any output from $\boldsymbol{rn}_n(F)$ has

$O(n)$ components, each of size $O(logn)$ that can be written as a value of a function (variable) of the program of F. Below n is fixed and often omitted in the notations.

The algorithm computes the values of outputs using pre-interpreted constants like integers, rational numbers, Boolean values, and pre-interpreted functions like addition, order relations over numbers, Boolean operations. These functions are *static* (our terminology is inspired by the terminology used by Yu. Gurevich for his Abstract State Machines), i.e., they do not change during the executions of F. The inputs are given by the values of functions (that constitute the respective structure) that F can only read; they are *dynamic* and *external*. The functions that can be changed by F are its *internal functions*, they are subdivided into *output functions* and *proper internal* functions. Assume for simplicity that the output functions are changed only once. Dynamic functions may have arguments like, e.g., arrays.

As we consider the work of F only for inputs from $\boldsymbol{dm}_n$ (without loss of generality we assume that these inputs are of size exactly n), we treat functions with arguments as sets of functions without arguments (as *nullary* functions) however indexed by these arguments; for example, an array $f[1..n]$ becomes a set of functions $f(1), f(2), \ldots, f(n)$, where each $f(i)$ is nullary. This set of internal nullary functions is denoted $\mathcal{F}_n^*$.

An execution of F can be represented as a sequence of states (that is called *run*) or, equivalently in a more compact form, as a sequence of events (that is called *trace*). We mainly use traces. In any case the initial construct of a run or of a trace is an *initial state* that consists of an input and initial values of internal functions. We can assume that the initial value of all internal functions is a symbol different of all other used up to now, denote it by $\natural$. And for any $f \in \mathcal{F}_n^*$ we assume that $f^{-1}(\natural) = \emptyset$.

The initial state is usually not taken into account in our treatment of metrics in the set of traces.

After the initial state, a trace is constituted from 2 types of events: an *assignment* or *update* $g := \theta$, where g is an internal nullary function and θ is a term, and a *guard evaluation to true* (a guard verification) $\eta!$, where without loss of generality we consider η to be a nullary function with Boolean values.

An event has a symbolic representations and its interpretation. An event, taken alone as interpreted or symbolic, may have an occurrence in a trace and an occurrence in the program F. An event in a trace is usually seen as an occurrence in the trace. For such a symbolic event we can construct a term that describes its history by going back in time and by taking the consecutive symbolic expressions the involved functions.

Consider as an example binary convolution. Given 2 strings $x, y \in \mathbb{B}^n$ and denoting their bits as respectively $x(0), x(1), \ldots, x(n-1)$ and $y(0), y(1), \ldots, y(n-1)$ the algorithm computes $z(k) = \sum_{i=0}^{i=k} x(i)y(k-i)$, $0 \leq k \leq (2n-2)$ (we assume that $x(i) = y(i) = 0$ for $n-1 < i \leq (2n-2)$). We can compute these values using one nullary proper function, say, u: for each k we compute in a loop:

$u := x(0)y(k);\ u := u + x(1)y(k-1); \ldots ; u := u + x(k)y(0),$

and assign the last value as output value of the kth component of the convolution, i.e., to $z(k)$. The symbolic history of the last mentioned occurrence of u is the term for $z(k)$ defined above (sure, only 2-argument addition is used in this term).

In fact we use the notion of trace rather abstractly, without much references to the fact that they represent executions of an algorithm.

Denote by $\boldsymbol{tr}_n(X)$ the trace corresponding to an input $X \in \boldsymbol{dm}_n(F)$, and by $\boldsymbol{Tr}_n$ the set of all traces for inputs from $\boldsymbol{dm}_n$. The length $|\boldsymbol{tr}(X)|$ of a trace $\boldsymbol{tr}(X)$ is the number of occurrences of events in it, i.e., the time complexity for X (denoted by $\mathbf{t}_F^*(X)$), and the maximum of these lengths is the worst-case time complexity of F that is denoted $\mathbf{t}_F(n)$.

Denote by $\boldsymbol{ev}(X, t)$ the event in $\boldsymbol{tr}(X)$ at an instant t, and by $\boldsymbol{fn}(X, t)$ the function updated at this instant if the event is an assignment, and the function used in the guard if this event is a guard verification. 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.

Each $f \in \mathcal{F}^*$ has its range (the set of values) $\boldsymbol{rn}_n(X, f)$ in a trace $\boldsymbol{tr}(X)$, and its range $\boldsymbol{rn}_n(f)$ in the set of all traces: $\boldsymbol{rn}_n(f) = \bigcup_{X \in \mathbf{Tr}_n} \boldsymbol{rn}_n(X, f)$.

1.1. 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, and take as a measure of similarity the size of the rest. More precisely, if σ is the longest common subsequence then we take as measure the value $|\boldsymbol{tr}(X)| + |\boldsymbol{tr}(Y)| - 2|\sigma|$, 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. In other words (more technical presentation is in (?, ?)) for given inputs X and Y we take a bijection β of functions of $\mathcal{F}^*$ preserving the output functions, a bijection γ of $\boldsymbol{rn}(\gamma(f))$ and $\boldsymbol{rn}(f)$ for all $f \in \mathcal{F}^*$, and then we take two subsequences σ_X in $\boldsymbol{tr}(X)$ and σ_Y in $\boldsymbol{tr}(Y)$, and a bijection φ from σ_X to σ_Y preserving the causal order and compatible, in a standard way, with β and γ. The latter means that φ of an update $f := \theta$ in $\boldsymbol{tr}(X)$ is an update of $\beta(f) := \beta(\theta)$ in $\boldsymbol{tr}(Y)$ whose value is γ of the acquired value of f in $\boldsymbol{tr}(X)$. Similar for guard verifications. All the mentioned bijections constitute a *similarity (bijection)* from $\boldsymbol{tr}(X)$ to $\boldsymbol{tr}(Y)$.

After this choice of a similarity bijection we compute the symmetric difference as above and take minimum over all bijections and all subsequences. It 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 space $\boldsymbol{Tr}_n$ is clearly compact, this metric permits to define epsilon-entropy (?, ?) 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.

Unfortunately, it is not clear how this metric, based on 'syntactic similarity', is related to the advancement of the algorithm towards the result. What is worse, the epsilon entropy calculated on the basis of this metric is not well coherent with intuition (see (?, ?) for the evaluations that we use just below).

Take a straightforward algorithm for palindrome recognition, i.e., the algorithm that compares characters in an input string starting from the ends and going to the middle of the string until either it detects inequality (then it outputs 0 that signifies "non palindrome") or reaches the middle of the string (then it outputs 1 that signifies "palindrome"). We assume that the positions of characters in W are numbered from 1 to n: $W = W(1)W(2)\ldots W(n)$. For simplicity assume that n is even. Then the algorithm works in a loop whose counter i changes from 1 to at most $\frac{n}{2}$, and at 'step' i the algorithm checks $W(i) = W(n - i + 1)$. If this guard is true and $i = \frac{n}{2}$ then the output is 1. If the guard is false the output is 0. If $i < \frac{n}{2}$ and the guard is true then i is augmented $i := i + 1$, and the mentioned step is repeated.

For this algorithm for palindrome recognition for $\varepsilon > 2$ the ε-entropy is $\log n - c_0$ for a small positive constant c_0. This estimation looks reasonable. But if we take the straightforward algorithm for the convolution, namely the algorithm based on the formula for $z(k)$ above then even for ε of order smaller than n the value of ε-entropy is bigger than $c_1 \cdot n$ for a small $c_1 > 0$, that is the ε-entropy is exponentially higher than for palindrome recognition, though the algorithm for convolution is rather simple, without much diversity in computations. The comparison may be more clear if we normalize the distance by the worst-case complexity: in both cases we may take ε like $\frac{c}{n}$ and arrive at the mentioned difference in ε-entropy.

1.2. Remark on Kolmogorov Complexity Approach

A direct application of Kolmorogov algorithmic notion of entropy (?, ?) to measure similarity of traces does not gives results corresponding to our intuitive goal. Indeed, in the cited paper Kolmogorov defines entropy as relative Kolmogorov complexity $\boldsymbol{K}(\alpha|\beta)$. A straightforward way to measure similarity of structures α and β is to define it as $\mathfrak{K}(\alpha, \beta) = \boldsymbol{K}(\alpha|\beta) + \boldsymbol{K}(\beta|\alpha)$. If we suppose that the length of a composition (superposition) of two programs is not greater than the sum of their lengths then $\mathfrak{K}$ is a pseudo-metric.

If we try to evaluate Kolmogorov complexity $\boldsymbol{K}$ for traces directly we get

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

where $|F|$ and $|X|$ are the binary lengths of respectively the program for F and input X. It follows from this observation that the knowledge of the input X and of the program F are sufficient to calculate the trace $\boldsymbol{tr}(X)$. This means that whatever be an algorithm F and its computational complexity, the $\mathfrak{K}$-distance between traces for inputs from $\boldsymbol{dm}_n(F)$ is always not greater than $O(n)$ if we assume that the binary length of $X \in \boldsymbol{dm}_n(F)$ is of order n. On the other hand, given a minimal program G that computes $\boldsymbol{tr}(X)$ from $\boldsymbol{tr}(Y)$ one can compute X from Y as follows: from Y one computes $\boldsymbol{tr}(Y)$ with the help of F (whose size is a constant with respect to n), then with the help of G one computes $\boldsymbol{tr}(X)$ and finally one can extract X from $\boldsymbol{tr}(X)$ with the help of a simple program, say E, whose length can be considered as constant (without loss of generality we can assume that the input is not only in the initial state but is also reproduced at the beginning of each trace). All this gives

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

But if we assume that the cardinality of $\boldsymbol{dm}_n(F)$ exponential, then the chain rule for Kolmogorov complexity gives that for almost all X, Y we have

$\boldsymbol{K}(X|Y) = \boldsymbol{K}(X, Y) - \boldsymbol{K}(X) - O(\log \boldsymbol{K}(X, Y)) \geq n - c \log n$ 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 always of order of n that can hardly be seem as satisfactory for evaluation of similarity of traces from $\boldsymbol{Tr}_n$.

The remarks above does not exclude that more clever framework within ideas of this kind may give a reasonable approach. One can try a more general notion of entropy, for example the one from (?, ?) that is based on inference complexity.

1.3. Similarity via Entropy of Partitions

In this subsection we outline another approach to measure similarity of traces. We weaken the syntactic demand but strengthen the semantic one. This approach refers to the classical entropy of partitions, and 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 'knowledge' of the algorithm under study.

When an algorithm F starts its work it 'knows' nothing, in particular about its output. So all values from $\boldsymbol{rn}_n$ is equiprobable. It is not clear how to formalize this intuition for the entire set of values $\bigcup_n \boldsymbol{rn}_n(F)$ that is infinite. So we do it for a concrete n.

Let $M = |\boldsymbol{rn}_n(F)|$ be the number of different values of F for arguments from $\boldsymbol{dm}_n$. 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 $1/2$ each. 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.

For an input $X \in \boldsymbol{dm}_n(F)$ and a time instant t, $1 \leq t \leq \mathbf{t}^*(X)$, we use notation: $f[X,t]$ is the value of f in $\boldsymbol{tr}(X)$ at t (this does not mean that the value is acquired exactly at t, maybe this happened earlier).

Let $f = \boldsymbol{fn}(X,t)$ 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 try to compare only events that are 'similar' in some way, that is determined by our goal and our vision of the situation.

Formally speaking a similarity of events is a relation, that we denote $\sim$, over pairs of events: $\boldsymbol{ev}(X,t) \sim \boldsymbol{ev}(X',t')$, or even more formally, over pairs: $(X,t) \sim (X',t')$. We assume that $\boldsymbol{fn}(X,t)$ and $\boldsymbol{fn}(X',t')$ are updated by the same command of the program F for $(X,t) \sim (X',t')$.

A way to define such a relation, that works particularly well for non-branching programs (e.g., for the mentioned algorithm for convolution), is the following one (in a non-branching program at any instant in any trace there is executed the same command of the program). For an occurrence α of an action in the program F we define as similar the kth application of action α in all traces from $\boldsymbol{Tr}_n$. With this notion, in non-branching programs for any event in any trace there is a similar event in any other trace. However, in general case we cannot count on this property, so for a given event in a trace there may be no similar event in another given trace.

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 of partitions.

Take any input X and an instant t, $1 \leq t \leq \mathbf{t}^*(X)$. Let $f = \boldsymbol{fn}(X,t))$, and $v = f[X,t]$. Denote by $\boldsymbol{sm}(X,t)$ all the inputs X' such that $(X,t) \sim (X',t')$ and $f[X',t'] = v$ for some t' (recall that we assume that if $(X,t) \sim (X',t')$ then $\boldsymbol{fn}(X,t) = \boldsymbol{fn}(X',t')$). 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 $\boldsymbol{sm}(X,t)^c =_{df} \boldsymbol{dm}_n \setminus \boldsymbol{sm}(X,t)$.

The traces of the straightforward algorithm for palindrome recognition are of different lengths, so in order to compute the distance defined below we complete shorter traces by some special partition, denote it $\pitchfork$, to which the local distance is 'big'; we comment on this below.

Thus, each input X determines a *sequence* $(\boldsymbol{spt}(X,t))_t$ *of partitions* $\boldsymbol{pt}(X,t)$ of $\boldsymbol{dm}_n(F)$.

For measurable partitions of a probabilistic space $\mathcal{P} = (\Omega, \Sigma, P)$ one can define entropy (for general probabilistic spaces one should impose some technical constraints to treat the entropy, but such constraints are irrelevant in our case), see (?, ?, ?) or books like (?, ?).

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

$$H(\mathcal{A}) = -\sum_{A \in \mathcal{A}} P(A) \log P(A).$$

and conditional entropy as

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

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

The conditional entropy permits to introduce Rokhlin metric between partitions, see (?, ?, ?):

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

Using equality $H(\mathcal{A}/\mathcal{B}) = H(\mathcal{A}\vee\mathcal{B}) - H(\mathcal{B})$ one can rewrite it as

$\boldsymbol{\rho}(\mathcal{A},\mathcal{B}) = 2H(\mathcal{A}\vee\mathcal{B}) - H(\mathcal{A}) - H(\mathcal{B})$ that is often technically useful.

There are other ways to introduce distance between partitions, e.g., see (?, ?, 4.4), so one can take or invent maybe more productive metric.

Now we can define 'distance' between two traces $\boldsymbol{tr}(X)$ and $\boldsymbol{tr}(Y)$ as a 'distance' between partially ordered (by causal order) sets $\{\boldsymbol{spt}(X,t)\}_t$ and $\{\boldsymbol{spt}(Y,t)\}_t$.

For non-branching algorithms such a distance is easy to define as the events $\boldsymbol{ev}(X,t)$ and $\boldsymbol{ev}(Y,t)$ are similar for all t (we write the distance as a function of inputs):

$$\boldsymbol{d}(X,Y) =_{df} \frac{1}{\mathbf{t}(n)} \sum_{t=1}^{\mathbf{t}(n)} \boldsymbol{\rho}\big(\boldsymbol{spt}(X,t), \boldsymbol{spt}(Y,t)\big).$$

(Sure one can take another average.) Clearly, in this case we compare partitions of similar events: $(X,t)\sim(Y,t)$.

For palindrome recognition we again compare the sets of partitions as sequences one of which may be completed by suffixing it with $\pitchfork$. Again we get a distance.

In general case, given two inputs X and Y, we take a partial bijection, say β, between traces $\boldsymbol{tr}(X)$ and $\boldsymbol{tr}(Y)$ that preserves similarity and causal order. Then we extend β to a total bijection completing it with $\pitchfork$ that may be put anywhere in any trace. After that we take an average distance between obtained sequences, and minimize it over all bijections. A priori it is not clear whether we get a distance (triangle inequality should be verified).

This metric does not change much for palindrome recognition algorithm. This is not sup rising as the space of traces of this algorithm is, in fact very small and simple. On the other hand, the geometry of traces of the convolution algorithm becomes more interesting, the space is more compact, however its detail analysis is hindered by combinatorial difficulties that could be overcome after developing appropriate approximation methods that seems to be quite feasible.

One can look at the similarity from the viewpoint of metric spaces, and define an appropriate distance between metric spaces $\boldsymbol{spt}(X)$ and $\boldsymbol{spt}(Y)$ extending classical methods like in, e.g., (?, ?, ch. 7).

1.4. The Question of Information Convergence

Formalizing similarity of traces may become really useful if we could relate it to 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 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 updated function that also defines a partition denoted above $\boldsymbol{spt}(X,t)$. How this local knowledge that is in $\boldsymbol{spt}(X,t)$, is related to the resulting partition $(F^{-1}(F(X)), F^{-1}(F(X))^c)$ mentioned just above? One may answer: compare $\boldsymbol{spt}(X,t)$ (the local knowledge at an instant t in terms of partitions) with the partition $(F^{-1}(F(X)), F^{-1}(F(X))^c)$. If $F(X)$ consists of several components computed as values of different functions, we compare $\boldsymbol{spt}(X,t)$ with each component. Suppose for simplicity that $F(X)$ has one component that has M values. Let $y = F(X)$.

As we assumed that all values of $F(X)$ are equiprobable then the unconditional entropy of $F(X) = y$ is $-\frac{1}{M}\log\frac{1}{M} - (1-\frac{1}{M})\log(1-\frac{1}{M})$. If we take into account the knowledge given by partition $\boldsymbol{spt}(X,t) = \{B^+, B^-\}$ then, denoting $\{F^{-1}(y), F^{-1}(y)^c\}$ by $\{A^+, A^-\}$ we have conditional entropy

$$H((A^+, A^-)/\boldsymbol{spt}(X,t)) = -\sum_{\alpha,\beta\in\{+,-\}} \boldsymbol{P}(A^\alpha \cap B^\beta)\log\frac{\boldsymbol{P}(A^\alpha \cap B^\beta)}{\boldsymbol{P}(B^\beta)}$$

that says what is contributed to the knowledge of $F(X) = y$ by the computation $\boldsymbol{tr}(X)$ at step t.

This way of measuring ‘information convergence’ works for simple algorithms with a flavor of being ‘greedy’, i.e., for algorithms that at each step precipitate to get some information about the result. Our examples are of this type. But ‘clever’ fast algorithms are not necessarily like that. There may be repetitions of partitions that are invoked in the constructions above. Moreover, for algorithms with very high complexity repetitions are inevitable as the number of all possible partitions of $\boldsymbol{dm}_n(F)$ is bounded by some exponential function, and the time complexity of an algorithm may be higher.

Thus, more detailed analysis of the structure of inputs, and maybe that of computation is needed. The initial structure of inputs is given by the definition of problems, and we present very preliminary observations concerning this structure in the next section.

2. On the Structure of Problems

Here are presented examples of ‘practical’ 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. 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 (real or integer) linear programming and its generalizations like the 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 $x = x(0) \dots x(n-1)$ and $y = y(0) \dots y(n-1)$ calculate

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

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, i.e., conjunction of disjunctions), to find the longest satisfying assignment of variables.

(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 (A**??**)–(A**??**) 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 (A**??**) the structure is a simple linear graph $k, k+1, \dots, k+|U|-1$, and while walking along it, we calculate conjunction of $U(i) = W(i)$ for $k \le i < (k + |U|)$ until i reaches the last value or false appears.

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

The structures used in (A**??**) 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 (A**??**), walking through two given lists, namely a list of elements and a permutation, a third list (a list of permuted elements) is constructed.

Example (A**??**) 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 (B**??**)–(B**??**) give algorithmic problems of search among substructures coded in inputs. The number of these substructures, taken directly from the definition, is quadratic for (B**??**), and exponential for (B**??**)–(B**??**). The substructures under the search should satisfy conditions that characterize the corresponding direct problem. More complicated problems code substructures not so explicitly as in examples (B**??**)–(B**??**). 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 evolution. One can see that the cardinality constraint on the number of partitions that was mentioned in section **??** 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 example, where we should treat all the values, is multiplication; for concreteness, consider binary multiplication. Multiplication of 64-bit natural numbers is a common practice, and there are 2^{128} possible inputs. One can easily evaluate that most numbers will never be met in practice. But we do not know which ones, and I dare to say, we do not have a slightest idea how to separate those that will appear from those that will never appear.

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 is 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 rather 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 $2^{6 \cdot \log 26} > 2^{6 \cdot 4.7} > 2^{28}$ (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 sometimes 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 may be described in various terms, not only in the terms chosen in this paper; the latter are based on classical approaches to the description of information quantity.

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. We can find many algorithmic processes in the nature and in the society. 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 type of knowledge of of information do we acquire after each step of an algorithmic process?

Acknowledgments

I am thankful to Edward Hirsch and Eugene Asarin for their remarks and questions that stimulated this study.