Another motivation for reducing the randomness complexity of algorithms

Oded Goldreich
Department of Computer Science
Weizmann Institute of Science
Rehovot, ISRAEL.
oded.goldreich@weizmann.ac.il

November 21, 2006

Abstract

We observe that the randomness-complexity of an algorithm effects the time-complexity of implementing a version of it that utilizes a weak source of randomness (through a randomness-extractor). This provides an additional motivation for the study of the randomness complexity of randomized algorithms. We note that this motivation applies especially in the case that derandomization is prohibitingly costly.

Keywords: Randomness Complexity, Weak Sources of Randomness, Randomness Extractors, Pseudorandom Generators, Sampling, Property Testing.

Contents

Introduction: the standard motivations . . . 1
The main message: another motivation . . . 1
Two examples: the settings of Sampling and Property Testing . . . 3
Conclusions . . . 5
Bibliography . . . 5

Introduction: the standard motivations

The randomness-complexity of a randomized algorithm is a natural complexity measure associated with such algorithms. Furthermore, randomness is a "real" resource, and so trying to minimize the use of it falls within the standard paradigms of algorithmic design.

In addition to the aforementioned generic motivation (which was highlighted in [2]), there is a more concrete motivation (which was highlighted in [13]): If we manage to reduce the randomness-complexity to become sufficiently low then this opens the door to a relatively efficient derandomization. Specifically, a randomized algorithm having time-complexity t and randomness-complexity r yields a functionally equivalent deterministic algorithm of time-complexity $2^r \cdot t$.

The main message: another motivation

In this note we highlight another concrete motivation to the study of the randomness-complexity of randomized algorithms. We refer to the effect of the randomness-complexity on the overhead involved in implementing the algorithm when using only weak sources of randomness (rather than perfect ones). Specifically, we refer to the paradigm of implementing randomized algorithms by using (a single sample from) such a weak source, and trying all possible seeds to an adequate randomness extractor (see below). We will show that the overhead created by this method is determined by the randomness-complexity of the original algorithm.

Recall that a randomness extractor is a function $E : \{0,1\}^s \times \{0,1\}^n \to \{0,1\}^r$ that uses an s-bit long random seed in order to transform an n-bit long (outcome of a) weak source of randomness into an r-bit long string that is almost uniformly distributed in $\{0,1\}^r$. Specifically, we consider arbitrary weak sources that restricted (only) in the sense that, for a parameter k, no string appears as the source outcome with probability that exceeds 2^{-k}. Such sources are called (n,k)-sources (and k is called the min-entropy). Now, E is called a (k,ϵ)-extractor if for any (n,k)-source X it holds that $E(U_s, X)$ is ϵ-close to U_r, where U_m denotes the uniform distribution over m-bit strings (and the term 'close' refers to the statistical distance between the two distributions). For further details about (k,ϵ)-extractors, the interested reader is referred to Shaltiel's survey [10].

Next we recall the standard paradigm of implementing randomized algorithms while using sources of weak randomness. Suppose that the algorithm A has time-complexity t and randomness-complexity $r \leq t$. Recall that, typically, the analysis of algorithm A refers to what happens when A obtains its randomness from a perfect random source (i.e., for each possible input w, we consider the behavior of $A(w, U_r)$, where $A(w,\omega)$ denotes the output of A on input w when given randomness ω). Now, suppose that we have at our disposal only a weak source of randomness; specifically, a (n,k)-source for $n \gg k \gg r$ (e.g., $n = 10k$ and $k = 2r$). Then, using a (k,ϵ)-extractor $E : \{0,1\}^s \times \{0,1\}^n \to \{0,1\}^r$, we can transform the n-bit long outcome of this source into 2^s strings, each of length r, and use the resulting 2^s strings (which are "random on the average") in 2^s corresponding invocations of the algorithm A. That is, upon obtaining the outcome $x \in \{0,1\}^n$ from the source, we invoke the algorithm A for 2^s times such that in the i^{th} invocation we provide A with randomness $E(i,x)$. The results of these 2^s invocations are processed in the natural manner. For example, if A is a decision procedure, then we output the majority vote obtained in the 2^s invocations (i.e., when given the input w, we output the majority vote of the sequence $\langle A(w, E(i,x))\rangle_{i=1,\ldots,2^s}$).[1]

The analysis of the foregoing implementation is based on the fact that "on the average" the 2^s strings extracted from the source approximate a perfect r-bit long source (i.e., a random setting of the s-bit seed yields an almost uniformly distributed r-bit string). In the case of decision procedures this means that if A has error probability p and X is a (n,k)-source then the number of values in $\langle E(i,X)\rangle_{i=1,\ldots,2^s}$ that fail $A(w,\cdot)$ is at most $(p+\epsilon)\cdot 2^s$, where the expectation is taken over the distribution of X. It follows that the implementation (which rules by majority) errs with probability at most $2(p+\epsilon)$. This means that we should start with $p < 1/4$. (A similar analysis can be applied to the randomized search procedures discussed in Footnote 1.)

Let us consider the cost of the foregoing implementation. We assume, for simplicity, that the running-time of the randomness extractor is dominated by the running-time of A. Then, algorithm A can be implemented using a weak source, while incurring an overhead factor of 2^s. Recalling that $s > \log_2 n$ and that $n > k > r$ it follows that the aforementioned overhead is at least linear in r. On the other hand, if $s = (1+o(1))\log_2 n$ and $n = O(r)$ (resp., $s = O(\log n)$ and $n = \mathrm{poly}(r)$) then the aforementioned overhead is in fact linear in r (resp., polynomial in r). This establishes our

[1]For search problems in NP, we output any valid solution that is obtained in the relevant 2^s invocations. For general search problems (i.e., outside NP), some extra condition regarding the original randomized algorithm is required (e.g., either that it never outputs a wrong solution or that it outputs some specific correct solution with probability that exceeds 1/2 by a noticeable amount).

claim that the time-complexity of implementing randomized algorithms when using weak sources is related to the randomness-complexity of these algorithms. Let us take a closer look at this relationship.

We shall consider two types of (n,k)-sources, which are most appealing. Indeed, these type of sources have received a lot of attention in the literature. Recall that r denotes the number of bits that we need to extract for such a source (in order to feed our algorithm). Furthermore, it suffices to set the deviation parameter of the extractor (i.e., ϵ) to a small constant (e.g., $\epsilon = 1/10$ will do). The two cases we consider are:

1. *Linearly related* n, k *and* r: that is, for some constants $c > c' > 1$, it holds that $n = c \cdot r$ and $k = c' \cdot r$. In other words, we refer to sources having a constant rate of min-entropy.

   In this case, efficient randomness extractors that use $s = \log n + O(\log\log n) = \log_2 \tilde{O}(n)$ are known (cf. [12, 10]). Using these extractors, we obtain an implementation of A (using such weak sources) with overhead factor $\tilde{O}(r)$.

2. *Polynomially related* n, k *and* r: that is, for some $c > c' > 1$, it holds that $n = r^c$ and $k = r^{c'}$. In other words, we refer to a source having min-entropy that is polynomially related to its length.

   In this case, efficient randomness extractors that use $s = \log \tilde{O}(n) = c \log_2 \tilde{O}(r)$ are known (cf. [11, 10]). Using these extractors, we obtain an implementation of A (using such weak sources) with overhead factor $\tilde{O}(r^c)$.

In both cases, the overhead factor is approximately linear in the length of the source's outcome (which, in turn, is linearly or polynomially related to r).

We wish to stress that the implementation paradigm considered above is most relevant in the case that a full derandomization (incurring an overhead factor of 2^r) is prohibitingly costly. Two settings in which this is inherently the case are considered next.

Two examples: the settings of Sampling and Property Testing

Derandomization is not a viable option in the setting of sampling and property testing, and thus these settings provide a good demonstration of the importance of the new motivation. We start with the setting of sampling, although the more dramatic results are obtained in the context of property testing, which may be viewed as a generalization of the context of sampling.

Sampling. In many settings repeated sampling is used to estimate the average of a huge set of values. Namely, given a "value" function $\nu : \{0,1\}^n \to \mathsf{R}$, one wishes to approximate $\bar{\nu} \stackrel{\rm def}{=} \frac{1}{2^n} \sum_{x \in \{0,1\}^n} \nu(x)$ without having to inspect the value of ν at each point of the domain. The obvious thing to do is sampling the domain at random, and obtaining an approximation to $\bar{\nu}$ by taking the average of the values of ν on the sample points. It is essential to have the range of ν be bounded (or else no reasonable approximation is possible). For simplicity, we adopt the convention of having $[0,1]$ be the range of ν, and the problem for other (predetermined) ranges can be treated analogously. Our notion of approximation depends on two parameters: accuracy (denoted ϵ) and error probability (denoted δ). We wish to have an algorithm that, with probability at least $1-\delta$, gets within ϵ of the correct value. That is, a sampler is a randomized oracle machine that on input parameters n, ϵ, δ and oracle access to *any* function $\nu : \{0,1\}^n \to [0,1]$, outputs, with probability at least $1-\delta$, a value that is at most ϵ away from $\bar{\nu} \stackrel{\rm def}{=} \frac{1}{2^n} \sum_{x \in \{0,1\}^n} \nu(x)$.

We are interested in "the complexity of sampling" quantified as a function of the parameters n, ϵ and δ. Specifically, we will consider three complexity measures: The sample-complexity (i.e., the number of oracle queries made by the sampler); the randomness-complexity (i.e., the length of the random seed used by the sampler); and the computational-complexity (i.e., the running-time of the sampler). We say that a sampler is efficient if its running-time is polynomial in the total length of its queries (i.e., polynomial in both its sample-complexity and in n). It is easy to see that a deterministic sampler must have sample-complexity close to 2^n, and thus derandomization is not an option here.

While efficient samplers that have optimal (up-to a constant factor) sample-complexity are of natural interest, the motivation for the study of the randomness-complexity of such samplers is less evident. Indeed, one may offer the generic answer (i.e., that randomness as any other resource need to be minimized), but the previous section shows that (in a very natural setting) there is a very concrete reason to care about the randomness-complexity of the sampler: the randomness-complexity of the sampler effects the sample-complexity of an implementation that uses a weak random source.

Recall that the naive sampler uses sample-complexity $s \stackrel{\rm def}{=} O(\epsilon^{-2}\log(1/\delta))$ and randomness-complexity $r = s \cdot n$. Using sources of constant min-entropy rate, this yields an implementation of sample-complexity $s' \approx r \cdot s = s^2 n$. However using a better sampler that has sample-complexity $O(s)$ but randomness-complexity $r'' = 2n + O(\log(1/\delta))$ (cf. [1]), we obtain an implementation of sample-complexity $s'' \approx r'' \cdot s \approx \epsilon^2 s^2 n$ (assuming $\delta > 2^{-n}$). This is a significant saving, whenever good accuracy is required (i.e., ϵ is small).

Property testing. This notion refers to a relaxation of decision problems, where it suffices to distinguish inputs having a (fixed) property from objects that are far from satisfying the property (cf. [8, 5]). Typically, one seeks sublinear-time algorithms that query the object at few randomly selected locations. In the natural cases, derandomization is not a viable option, because a deterministic algorithm must inspect at least a constant fraction of the object (and thus cannot run in sublinear-time). Let us clarify this discussion by looking at the specific example of testing the bipartiteness property for graphs of bounded-degree.

Fixing a degree bound d, the task is to distinguish (N-vertex) bipartite graphs of maximum degree d from (N-vertex) graphs of maximum degree d that are ϵ-far from bipartite (for some parameter ϵ), where ϵ-far means that $\epsilon \cdot dN$ edges have to be omitted from the graph in order to yield a bipartite graph. It is easy to see that no deterministic algorithm of $o(N)$ time-complexity can solve this problem. Yet, there exists a probabilistic algorithm of time-complexity $\tilde{O}(\sqrt{N}\mathrm{poly}(1/\epsilon))$ that solves this problem correctly (with probability 2/3). This algorithm makes $q \stackrel{\rm def}{=} \tilde{O}(\sqrt{N}\mathrm{poly}(1/\epsilon))$ incidence-queries to the graph, and (as described in the original work [7]) has randomness-complexity $r > q > \sqrt{N}$ (yet $r < q \cdot \log_2 N$).[2]

Let us now turn to the question of implementing the foregoing tester in a setting where we have access only to a weak source of randomness. In this case, the implementation calls for invoking the original tester $\tilde{O}(r)$ times, which yields a total running time of $\tilde{O}(r) \cdot \tilde{O}(\sqrt{N}\mathrm{poly}(1/\epsilon)) > N$. But in such a case we better use the standard (deterministic) decision procedure for bipartiteness!

Fortunately, a randomness-efficient implementation of the original tester of [7] is possible. This implementation (presented in [9]) has randomness-complexity $r' = \mathrm{poly}(\epsilon^{-1}\log N)$ (rather than $r = \mathrm{poly}(\epsilon^{-1}\log N) \cdot \sqrt{N}$). Thus, the cost of the implementation that uses a weak source of randomness

[2] We comment that $\Omega(\sqrt{N})$ is a lower-bound on the query-complexity of any property tester of bipartiteness (in the bounded-degree model; see [6]).

is related to $r' \cdot s = \tilde{O}(\sqrt{N}\text{poly}(1/\epsilon))$, which matches the original bound (up to differences hidden in the $\tilde{O}()$ and poly() notation). Needless to say, this is merely an example, and randomness-efficient implementations of other property testers are also presented in [9].

Conclusions

This essay articulates an additional motivation for studying the randomness-complexity of algorithms. This motivation is relevant even if one believes that generating a random bit is not more expensive than performing a standard machine instruction. It refers to the fact that the randomness-complexity of a randomized algorithm effects the running-time of an implementation that may only utilize a weak source of randomness (and does so by using a randomness-extractor). Specifically, such an implementation incurs a multiplicative overhead factor that is (at least) linearly related to the randomness-complexity of the original algorithm. This fact motivates the attempt to present randomness-efficient versions of randomized algorithms and even justifies the use of pseudorandom generators for this purpose.[3]

Acknowledgments

We are grateful to Ronen Shaltiel for an update regarding the state-of-art of the problem of randomness extraction. We also wish to thank Adi Akavia and Shafi Goldwasser for a discussion that led to the current note.

References

[1] M. Bellare, O. Goldreich, and S. Goldwasser. Randomness in Interactive Proofs. *Computational Complexity*, Vol. 4, No. 4, pages 319–354, 1993.

[2] M. Blum and S. Micali. How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits. *SIAM Journal on Computing*, Vol. 13, pages 850–864, 1984. Preliminary version in *23rd FOCS*, 1982.

[3] O. Goldreich. *Foundation of Cryptography – Basic Tools.* Cambridge University Press, 2001.

[4] O. Goldreich, S. Goldwasser, and S. Micali. How to Construct Random Functions. *Journal of the ACM*, Vol. 33, No. 4, pages 792–807, 1986.

[5] O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. *Journal of the ACM*, pages 653–750, July 1998.

[6] O. Goldreich and D. Ron. Property testing in bounded degree graphs. *Algorithmica*, pages 302–343, 2002.

[7] O. Goldreich and D. Ron. A sublinear bipartite tester for bounded degree graphs. *Combinatorica*, Vol. 19 (3), pages 335–373, 1999.

[3] The running-time of the generator itself can be ignored, because one may use a pseudorandom generator that runs in time that is almost-linear in the length of its output. Such a generator can be constructed by using a pseudorandom function (cf. [4]). Alternatively, such a generator is implied by the notion of an on-line pseudorandom generator (cf. [3, Chap. 3, Exer. 21]).

[8] R. Rubinfeld and M. Sudan. Robust characterization of polynomials with applications to program testing. *SIAM Journal on Computing*, Vol. 25 (2), pages 252–271, 1996.

[9] O. Sheffet. M.Sc. Thesis, Weizmann Institute of Science, in preparation. Available from `http://www.wisdom.weizmann.ac.il/~oded/msc-os.html`

[10] R. Shaltiel. Recent Developments in Explicit Constructions of Extractors. *Bulletin of the EATCS 77*, pages 67–95, 2002.

[11] R. Shaltiel and C. Umans. Simple Extractors for All Min-Entropies and a New Pseudo-Random Generator. In *32nd IEEE Symposium on Foundations of Computer Science*, pages 648–657, 2001.

[12] A. Ta-Shma, D. Zuckerman, and S. Safra. Extractors from Reed-Muller Codes. In *32nd IEEE Symposium on Foundations of Computer Science*, pages 638–647, 2001.

[13] A.C. Yao. Theory and Application of Trapdoor Functions. In *23rd IEEE Symposium on Foundations of Computer Science*, pages 80–91, 1982.