On the Implementation of Huge Random Objects

Oded Goldreich*† Shafi Goldwasser*†‡ Asaf Nussboim*

Abstract

We initiate a general study of pseudo-random implementations of huge random objects, and apply it to a few areas in which random objects occur naturally. For example, a random object being considered may be a random connected graph, a random bounded-degree graph, or a random error-correcting code with good distance. A pseudo-random implementation of such type T objects must generate objects of type T that can not be distinguished from random ones, rather than objects that can not be distinguished from type T objects (although they are not type T at all).

We will model a type T object as a function, and access objects by queries into these functions. We investigate supporting both standard queries that only evaluates the primary function at locations of the user's choice (e.g., edge queries in a graph), and complex queries that may ask for the result of a computation on the primary function, where this computation is infeasible to perform with a polynomial number of standard queries (e.g., providing the next vertex along a Hamiltonian path in the graph).

1 Introduction

Suppose that you want to run some experiments on random codes (i.e., subsets of $\{0,1\}^n$ that contain $K = 2^{\Omega(n)}$ strings). You actually take it for granted that the random code will have *large* (i.e., linear) *distance*, because you know some Coding Theory and are willing to discard the negligible probability that a random code will not have a large distance. Suppose that you want to be able to keep succinct representations of these huge codes and/or that you want to generate them using few random bits. Being aware of the relevant works on pseudorandomness (e.g., [16, 5, 24, 13]), you plan to use pseudorandom functions [13] in order to efficiently generate and store representations of these codes; that is, using the pseudorandom function $f : [K] \to \{0,1\}^n$, you can define the code $C_f = \{f(i) : i \in [K]\}$, and efficiently produce codewords of C_f. *But wait a minute, do the codes that you generate this way have a large distance?*

The point is that having a large distance is a global property of the code, which in turn is a huge (i.e., exp(n)-sized) object. This global property cannot be decided by looking at polynomially many (i.e., poly(n)-many) codewords, and so its violation cannot be translated to a contradiction of the pseudorandomness of the function. Indeed, the substitution of a random function (or a random code) by a pseudorandom one is not guaranteed to preserve the global property. Specifically, all pseudorandom codes generated as suggested above may have small distance.[1]

So, *can we efficiently generate random-looking codes of large distance?* Specifically, can we provide a probabilistic polynomial-time procedure that allows to sample codewords from a code of large distance such that the sampled codewords look as if they were taken from a random code (which, in particular, means that we do not generate linear codes). The answer is essentially positive: see Theorem 3.12. However, this is merely an example of the type of questions that we deal with. (Another illustrative example is presented in Appendix A.)

We initiate a general study of the feasibility of implementing (huge) random objects. For a given Type T of objects, we aim at generating pseudorandom objects of Type T. That is, we want the generated object to always be of Type T, but we are willing to settle for Type T objects that look as if they are truly random Type T objects (although they are not). We stress that our focus is on *Type T objects that look like random Type T objects*, rather than *objects that look like random Type T objects although they are not of Type T at all.* For example, we disapprove of a random function as being an implementation of a random permutation, although the two look alike to anybody restricted to resources that are polynomially related to the length of the inputs to the function. Beyond the intuitive conceptual reason for the

*Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot, ISRAEL. Email: {oded, shafi, asafn}@wisdom.weizmann.ac.il.

†Supported by the MINERVA Foundation, Germany.

‡Laboratory for Computer Science, MIT.

[1] Indeed, for each function f_s taken from some pseudorandom ensemble $\{f_s : [2^{|s|/10}] \to \{0,1\}^{|s|}\}_s$, it *may* hold that the Hamming distance between $f_s(i_s)$ and $f_s(i_s+1)$ is one, for some i_s that depends arbitrarily on f_s. For example, given a pseudorandom ensemble $\{f_s\}$, consider the ensemble $\{f_{s,i}\}$ such that $f_{s,i}(i) = 0^n$, $f_{s,i}(i+1) = 0^{n-1}1$ and $f_{s,i}(x) = f_s(x)$ for all other x's.

above disapproval, there are practical considerations. For example, if somebody supplies an element in the range then we may want to be guaranteed that this element has a unique preimage (as would be the case with any permutation but not with a random function).

In general, when one deals (or experiments) with an object that is supposed to be of Type T, one may assume that this object has all the properties enjoyed by all Type T objects. If this assumption does not hold (even if one cannot detect this fact during initial experimentation) then an application that depends on this assumption may fail. One reason for the failure of the application may be that it uses significantly more resources than those used in the initial experiments that failed to detect the problem. Another issue is that the probability that the application fails may indeed be negligible (as is the probability of detecting the failure in the initial experiments), but due to the importance of the application we are unwilling to tolerate even a negligible probability of failure.

We explore several areas in which the study of random objects occurs naturally. These areas include graph theory, coding theory and cryptography. We provide implementations of various natural random objects, which were considered before in these areas (e.g., the study of random graphs [6]).

Objects, specifications, implementations and their quality

Our focus is on huge objects; that is, objects that are of size that is exponential in the running time of the applications. Thus, these (possibly randomized) applications may inspect only small portions of the object (in each randomized execution). The object may be viewed as a function (or an oracle), and inspecting a small portion of it is viewed as receiving answers to a small number of adequate queries. For example, when we talk of huge dense graphs, we consider adjacency queries that are vertex-pairs with answers indicating whether or not the queried pair is connected by an edge. When we talk of huge bounded-degree graphs, we consider incidence queries that correspond to vertices with answers listing the neighbors of the queried vertex.

We are interested in classes of objects (or object types), which can be viewed as classes of functions. (Indeed, we are not interested in the trivial case of generic objects, which is captured by the class of all functions.) For example, when we talk of simple undirected graphs in the adjacency predicate representation, we only allow symmetric and non-reflexive Boolean functions. Similarly, when we talk of such bounded-degree graphs in the incident-lists representation, we restrict the class of functions in a less trivial manner (i.e., u should appear in the neighbor-list of v iff v appears in the neighbor-list of u). More interestingly, we may talk of the class of connected (or Hamiltonian) graphs, in which case the class of functions is even more complex. This formalism allows to talk about objects of certain types (or of objects satisfying certain properties). In addition, it captures complex objects that support "compound queries" to more basic objects. For example, we may consider an object that answers queries regarding a global property of a Boolean function (e.g., the parity of all the function's values). The queries may also refer to a large number of values of the function (e.g., the parity of all values assigned to arguments in an interval that is specified by the query).

We study probability distributions over classes of objects. Such a distribution is called a specification. Formally, a specification is presented by a *computationally-unbounded* probabilistic Turing machine, where each setting of the machine's random-tape yields a huge object. The latter object is defined as the corresponding input-output relation, and so queries to the object are associated with inputs to the machine. We consider the distribution on functions obtained by selecting the specification's random-tape uniformly. For example, a random N-vertex Hamiltonian graph is specified by a computationally-unbounded probabilistic machine that uses its random-tape to determine such a (random Hamiltonian) graph, and answers adjacency queries accordingly. Another specification may require to answer, in addition to adjacency queries regarding a uniformly selected N-vertex graph, also more complex queries such as providing a clique of size $\log_2 N$ that contains the queried vertex. We stress that the specification is not required to be even remotely efficient (but for sake of simplicity we assume that it is recursive).

Our ultimate goal will be to provide a *probabilistic polynomial-time* machine that implements the desired specification. That is, we consider the *probability distribution* on functions induced by fixing of the random-tape of the latter machine in all possible ways. Again, each possible fixing of the random-tape yields a function corresponding to the input-output relation (of the machine per this contents of its random-tape).

Indeed, a key question is how good is the implementation provided by some machine. We consider two aspects of this question. The first (and more standard) aspect is whether one can distinguish the implementation from the specification when given oracle access to one of them. Variants include perfect indistinguishability, statistical-indistinguishability and computational-indistinguishability. We stress a second aspect regarding the quality of implementation: the truthfulness of the implementation with respect to the specification, where being truthful means that any possible function that appears with non-zero probability in the implementation must also appear with non-zero probability in the specification. For example, if the specification is of a random Hamiltonian graph then a truthful implementation must always yield a Hamiltonian graph. (A reasonable relaxation of the notion of truthfulness is to require that all but a negligible part of the probability mass of the implementation is assigned to functions that appear with non-zero probability in the specification; an implementation satisfying this relaxation is called almost-truthful.)

Organization

In Section 2, we present formal definitions of the notions discussed above as well as basic observations regarding these notions. These are followed by a few known examples of non-trivial implementations of various random objects (which are retrospectively cast nicely in our formulation). In Section 3, we state a fair number of new implementations of various random objects, while deferring the constructions (and proofs) to our technical report [14]. These implementations demonstrate the applicability of our notions to various domains such as functions, graphs and codes. Conclusions and open problems are presented in Section 4.

2 Formal Setting and General Observations

Throughout this work we let n denote the feasibility parameter. Specifically, feasible-sized objects have an explicit description of length $\text{poly}(n)$, whereas huge objects have (explicit description) size exponential in n. The latter are described by functions from $\text{poly}(n)$-bit strings to $\text{poly}(n)$-bit strings. Whenever we talk of efficient procedures we mean algorithms running in $\text{poly}(n)$-time. The proofs of the (novel) results stated in this section appear in our technical report [14].

2.1 Specification

A huge random object is specified by a *computationally-unbounded* probabilistic Turing machine. For a fixed contents of the random-tape, such a machine defines a (possibly partial) function on the set of all binary strings. Such a function is called an instance of the specification. We consider the input-output relation of this machine when the random-tape is uniformly distributed. Loosely speaking, this is the random object specified by the machine.

For sake of simplicity, we confine our attention to machines that halt with probability 1 on every input. Furthermore, we will consider the input-output relation of such machines only on inputs of some specified length ℓ, where ℓ is always polynomially related to the feasibility parameter n. Thus, for such a probabilistic machine M and length parameter $\ell = \ell(n)$, with probability 1 over the choice of the random-tape for M, machine M halts on every $\ell(n)$-bit long input.

Definition 2.1 (specification): *For a fixed function $\ell : \mathsf{N} \to \mathsf{N}$, the* instance specified by a probabilistic machine M, random-tape ω and parameter n *is the function $M_{n,\omega}$ defined by letting $M_{n,\omega}(x)$ be the output of M on input $x \in \{0,1\}^{\ell(n)}$ when using the random-tape $\omega \in \{0,1\}^{\infty}$. The* random object specified by M and n *is defined as $M_{n,\omega}$ for a uniformly selected $\omega \in \{0,1\}^{\infty}$.*

Note that, with probability 1 over the choice of the random-tape, the random object (specified by M and n) depends only on a finite prefix of the random-tape. Let us clarify our formalism by casting in it several simple examples, which were considered before (cf. [13, 21]).

Example 2.2 (a random function): *A random function from n-bit strings to n-bit strings is specified by the machine M that, on input $x \in \{0,1\}^n$ (parameter n and random-tape ω), returns the $\text{idx}_n(x)$-th n-bit block of ω, where $\text{idx}_n(x)$ is the index of x within the set of n-bit long strings.*

Example 2.3 (a random permutation): *Let $N = 2^n$. A random permutation over $\{0,1\}^n \equiv [N]$ can be specified by uniformly selecting an integer $i \in [N!]$; that is, the machine uses its random-tape to determine $i \in [N!]$, and uses the i-th permutation according to some standard order. An alternative specification, which is easier to state (alas even more inefficient), is obtained by a machine that repeatedly inspect the N next n-bit strings on its random-tape, until encountering a run of N different values, using these as the permutation. Either way, once a permutation π over $\{0,1\}^n$ is determined, the machine answers the input $x \in \{0,1\}^n$ with the output $\pi(x)$.*

Example 2.4 (a random permutation coupled with its inverse): *In continuation to Example 2.3, we may consider a machine that selects π as before, and responds to input (σ, x) with $\pi(x)$ if $\sigma = 1$ and with $\pi^{-1}(x)$ otherwise. That is, the object specified here provides access to a random permutation as well as to its inverse.*

2.2 Implementations

Definition 2.1 places no restrictions on the complexity of the specification. Our aim, however, is to implement such specifications *efficiently*. We consider several types of implementations, where in all cases we aim at *efficient* implementations (i.e., machines that respond to each possible input within polynomial-time). Specifically, we consider two parameters:

1. The type of model used in the implementation. We will use either a polynomial-time *oracle machine having access to a random oracle* or a standard probabilistic polynomial-time machine (viewed as a deterministic *machine having access to a finite random-tape*).

2. The similarity of the implementation to the specification; that is, is the implementation may be *perfect*, *statistically indistinguishable* or only *computationally indistinguishable* from the specification (by probabilistic polynomial-time oracle machines that try to distinguish the implementation from the specification by querying it at inputs of their choice).

Our real goal is to derive implementations by *ordinary machines* (having as good a quality as possible). We thus view implementations by *oracle machines having access to a random oracle* as merely a clean abstraction, which is useful in many cases (as indicated by Theorem 2.9 below).

Definition 2.5 (implementation by oracle machines): *For a fixed function* $\ell : \mathbb{N} \to \mathbb{N}$*, a* (deterministic) *polynomial-time* oracle *machine* M *and oracle* f*, the* instance implemented by M^f and parameter n *is the function* M^f *defined by letting* $M^f(x)$ *be the output of* M *on input* $x \in \{0,1\}^{\ell(n)}$ *when using the oracle* f*. The* random object implemented by M with parameter n *is defined as* M^f *for a uniformly distributed* $f : \{0,1\}^* \to \{0,1\}$.

In fact, $M^f(x)$ depends only on the value of f on inputs of length bounded by a polynomial in $|x|$. Similarly, an ordinary probabilistic polynomial-time (as in the following definition) only uses a poly($|x|$)-bit long random-tape when invoked on input x. Thus, for feasibility parameter n, the machine handles $\ell(n)$-bit long inputs using a random-tape of length $\rho(n) = \text{poly}(\ell(n)) = \text{poly}(n)$, where (w.l.o.g.) ρ is 1-1.

Definition 2.6 (implementation by ordinary machines): *For fixed functions* $\ell, \rho : \mathbb{N} \to \mathbb{N}$*, an ordinary polynomial-time machine* M *and a string* r*, the* instance implemented by M and random-tape r *is the function* M_r *defined by letting* $M_r(x)$ *be the output of* M *on input* $x \in \{0,1\}^{\ell(\rho^{-1}(|r|))}$ *when using the random-tape* r*. The* random object implemented by M with parameter n *is defined as* M_r *for a uniformly distributed* $r \in \{0,1\}^{\rho(n)}$.

We stress that an instance of the implementation is fully determined by the machine M and the random-tape r (i.e., we disallow "implementations" that construct the object on-the-fly while depending and keeping track of all previous queries and answers).

For a machine M (either a specification or an implementation) we identify the pair (M, n) with the random object specified (or implemented) by machine M and feasibility parameter n.

Definition 2.7 (indistinguishability of the implementation from the specification): *Let* S *be a specification and* I *be an implementation, both with respect to the length function* $\ell : \mathbb{N} \to \mathbb{N}$*. We say that* I perfectly implements S *if, for every* n*, the random object* (I, n) *is distributed identically to the random object* (S, n)*. We say that* I closely-implements S *if, for every oracle machine* M *that on input* 1^n *makes at most* polynomially-many queries *all of length* $\ell(n)$*, the following difference is negligible*[2] *as a function of* n

$$|\mathbf{Pr}[M^{(I,n)}(1^n) = 1] - \mathbf{Pr}[M^{(S,n)}(1^n) = 1]| \quad (1)$$

We say that I pseudo-implements S *if* Eq. (1) *holds for every probabilistic polynomial-time oracle machine* M *that makes only queries of length equal to* $\ell(n)$.

We stress that the notion of a close-implementation does not say that the objects (i.e., (I, n) and (S, n)) are statistically close; it merely says that they cannot be distinguished by a (computationally unbounded) machine that asks polynomially many queries. Indeed, the notion of pseudo-implementation refers to the notion of computational indistinguishability (cf. [16, 24]) as applied to functions (see [13]). Clearly, *any perfect implementation is a close-implementation*, and *any close-implementation is a pseudo-implementation*. Intuitively, the oracle machine M, which is sometimes called a (potential) distinguisher, represents a user that employs (or experiments with) the implementation. It is required that such a user cannot distinguish the implementation from the specification, provided that the user is limited in its access to the implementation or even in its computational resources (i.e., time).

Indeed, it is trivial to perfectly implement a random function (i.e., the specification given in Example 2.2) by using an *oracle machine* (with access to a random oracle). In contrast, the main result of Goldreich, Goldwasser and Micali [13] can be cast by saying that there exist a pseudo-implementation of a random function by an *ordinary machine*, provided that pseudorandom generators (or, equivalently, one-way function [17]) do exist. In fact, under the same assumption, it is easy to show that *every specification having a pseudo-implementation by an oracle machine also has a pseudo-implementation by an ordinary machine*. A stronger statement will be proven below (see Theorem 2.9).

Truthful implementations. An important notion regarding (non-perfect) implementations refers to the question of whether or not they satisfy properties that are enjoyed by the corresponding specification. Put in other words, the question is whether *each instance of the implementation is also an instance of the specification*. Whenever this condition holds, we call the implementation truthful. Indeed, every perfect implementation is truthful, but this is not necessarily the case for close-implementations. For example, a random function is a close-implementation of a random permutation (because it is unlikely to find a collision among polynomially-many

[2]A function $\mu : \mathbb{N} \to [0,1]$ is called negligible if for every positive polynomial p and all sufficiently large n's it holds that $\mu(n) < 1/p(n)$.

preimages); however, a random function is *not* a *truthful* implementation of a random permutation.

Definition 2.8 (truthful implementations): *Let S be a specification and I be an implementation. We say that I* is truthful to *S if for every n the support of the random object (I, n) is a subset of the support of the random object (S, n).*

Much of this work is focused on truthful implementations. The following simple result is useful in the study of the latter. We comment that this result is typically applied to (truthful) *close*-implementations by *oracle* machines, yielding (truthful) *pseudo*-implementations by *ordinary* machines.

Theorem 2.9 *Suppose that one-way functions exist. Then any specification that has a pseudo-implementation by an* oracle machine (*having access to a random oracle*) *also has a pseudo-implementation by an* ordinary machine. *Furthermore, if the former implementation is truthful then so is the latter.*

The sufficient condition is also necessary, because the existence of pseudorandom functions (i.e., a pseudo-implementation of a random function) implies the existence of one-way functions. In view of Theorem 2.9, whenever we seek truthful implementations (or, alternatively, whenever we do not care about truthfulness at all), we may focus on implementations by oracle machines.

Almost-Truthful implementations. Truthful implementations guarantee that each instance of the implementation is also an instance of the specification (and is thus "consistent with the specification"). A meaningful relaxation of this guarantee refers to the case that almost all the probability mass of the implementation is assigned to instances that are consistent with the specification (i.e., are in the support of the latter). Specifically, we refer to the following definition.

Definition 2.10 (almost-truthful implementations): *Let S be a specification and I be an implementation. We say that I* is almost-truthful to *S if the probability that (I, n) is not in the support of the random object (S, n) is bounded by a negligible function in n.*

Interestingly, almost-truthfulness is not preserved by the construction used in the proof of Theorem 2.9. In fact, there exists specifications that have almost-truthful close-implementations by oracle machines but not by ordinary machines (see Theorem 2.11 below). Thus, when studying almost-truthful implementations, one needs to deal directly with ordinary implementations (rather than focus on implementations by oracle-machines). Indeed, we will present a few examples of almost-truthful implementations that are not truthful.

Theorem 2.11 *There exists a specification that has an almost-truthful close-implementation by an oracle machine but has no almost-truthful implementation by an ordinary machine.*

We stress that the theorem holds regardless of whether or not the latter (almost-truthful) implementation is indistinguishable from the specification.

2.3 Known non-trivial implementations

In view of Theorem 2.9, *when studying truthful implementations*, we focus on implementations by oracle machines. In these cases, we shorthand the phrase *implementation by an oracle machine* by the term *implementation*. Using the notion of truthfulness, we can cast the non-trivial implementation of a random permutation provided by Luby and Rackoff [21] as follows.

Theorem 2.12 [21]: *There exists a* truthful *close-implementation of the specification provided in Example 2.3. That is, there exists a truthful close-implementation of the specification that uniformly selects a permutation π over $\{0,1\}^n$ and responses to the query $x \in \{0,1\}^n$ with the value $\pi(x)$.*

Contrast Theorem 2.12 with the trivial non-truthful implementation (by a random function) mentioned above. Note that, even when ignoring the issue of truthfulness, it is non-trivial to provide a close-implementation of Example 2.4 (i.e., a random permutation along with its inverse).[3] However, Luby and Rackoff [21] have also provided a truthful close-implementation of Example 2.4.

Theorem 2.13 [21]: *There exists a truthful close-implementation of the specification that uniformly selects a permutation π over $\{0,1\}^n$ and responses to the query $(\sigma, x) \in \{-1,+1\} \times \{0,1\}^n$ with the value $\pi^\sigma(x)$.*

Another known result that has the flavor of the questions that we explore was obtained by Naor and Reingold [22]. Loosely speaking, they provided a truthful close-implementation of a permutation selected uniformly among all permutations having a certain cycle-structure.

Theorem 2.14 [22]: *For any $N = 2^n$, $t = \text{poly}(n)$, and $C = \{(c_i, m_i) : i = 1, ..., t\}$ such that $\sum_{i=1}^{t} m_i c_i = N$, there exists a truthful close-implementation of a uniformly distributed permutation that has m_i cycles of size c_i, for $i = 1, ..., t$.*[4] *Furthermore, the implementation instance*

[3] A random function will fail here, because the distinguisher may distinguish it from a random permutation by asking for the inverse of a random image.

[4] Special cases include involutions (i.e., permutations in which all cycles have length 2), and permutations consisting of a single cycle (of length N). These cases are cast by $C = \{(2, N/2)\}$ and $C = \{(N, 1)\}$, respectively.

that uses the permutation π can also support queries of the form (x, i) to be answered by $\pi^i(x)$, for any $x \in \{0,1\}^n$ and any integer i (which is presented in binary).

We stress that the latter queries are served in time poly(n) also in case $i \gg \text{poly}(n)$.

2.4 A few general observations

Theorem 2.11 asserts the existence of specifications that cannot be implemented in an *almost-truthful* manner by an *ordinary* machine, regardless of the level of indistinguishability (of the implementation from the specification). We can get negative results that refer also to implementations by *oracle* machines, regardless of truthfulness, by requiring the implementation to be sufficiently indistinguishable (from the specification). Specifically:

Proposition 2.15 *The following refers to implementations by oracle machines and disregard the issue of truthfulness.*

1. *There exist specifications that cannot be closely-implemented.*
2. *Assuming the existence of one-way functions, there exist specifications that cannot be pseudo-implemented.*

The randomness complexity of implementations: Looking at the proof of Theorem 2.9, it is evident that as far as pseudo-implementations by ordinary machines are concerned (and assuming the existence of one-way functions), *randomness can be reduced to any power of the feasibility parameter* (i.e., to n^ϵ for every $\epsilon > 0$). The same holds with respect to truthful pseudo-implementations. On the other hand, the proof of Theorem 2.11 suggests that this collapse in the randomness complexity cannot occur with respect to almost-truthful implementations by ordinary machines (regardless of the level of indistinguishability of the implementation from the specification).

Theorem 2.16 (a randomness hierarchy): *For every polynomial ρ, there exists a specification that has an almost-truthful close-implementation by an ordinary machine that uses a random-tape of length $\rho(n)$, but has no almost-truthful implementation by an ordinary machine that uses a random-tape of length $\rho(n) - \omega(\log n)$.*

Composing implementations: A simple observation that is used in our work is that one can "compose implementations". That is, if we implement a random object R1 by an oracle machine that uses oracle calls to a random object R2, which in turn has an implementation by a machine of type T, then we actually obtain an implementation of R1 by a machine of type T. To state this result, we need to extend Definition 2.5 such that it applies to oracle machines that use arbitrary specifications (rather than a random oracle). Let us denote by $(M^{(S,n)}, n)$ an implementation by the oracle machine M (and feasibility parameter n) with oracle access to the specification (S, n).

Theorem 2.17 *Let $Q \in \{\text{perfect}, \text{close}, \text{pseudo}\}$. Suppose that the specification (S_1, n) can be Q-implemented by $(M^{(S_2,n)}, n)$ and that (S_2, n) has a Q-implementation by an ordinary machine* (resp., by an oracle machine with a random oracle). *Then, (S_1, n) has a Q-implementation by an ordinary machine* (resp., by an oracle machine with a random oracle). *Furthermore, if both the implementations in the hypothesis are truthful* (resp., almost-truthful) *then so is the implementation in the conclusion.*

2.5 Objects of feasible size

In contrast to the rest of this work, we shortly discuss the complexity of generating random objects of feasible size (rather than huge random objects). In other words, we are talking about implementing a distribution on poly(n)-bit long strings, and doing so in poly(n)-time. This problem can be cast in our general formulation by considering specifications that ignore their input (i.e., have output that only depend on their random-tape). In other words, we may view objects of feasible size as constant functions, and cosider a specification of such random objects as a distribution on constant functions. Thus, without loss of generality, the implementation may also ignore its input, and consequently in this case there is no difference between an implementation by ordinary machine and an implementation by oracle machine with a random oracle.

We note that perfect implementations of such distributions were considered before (e.g., in [1, 4, 11]), and distributions for which such implementations exist are called sampleable. In the current context, where the observer sees the entire object, the distinction between perfect implementation and close-implementation seems quite technical. What seems fundamentally different is the study of pseudo-implementations.

Theorem 2.18 *There exist specifications of feasible-sized objects that have no close-implementation, but do have* (both truthful and non-truthful) *pseudo-implementations.*

The proof of Theorem 2.18 also establishes the existence of specifications (of feasible-sized objects) that have have no truthful (and even no almost-truthful) implementation, regardless of the level of indistinguishability from the specification. Turning the table around, ignoring the truthfulness condition, we ask whether there exist specifications of feasible-sized objects that have no pseudo-implementations. A partial answer is provided by the following result, which relies on a non-standard assumption (see Footnote 5).

Proposition 2.19 *Assuming the existence of a collision-free hash* function[5], *there exists a specification of a random feasible-sized object that has no pseudo-implementation.*

Open Problem 2.20 (A stronger version of Proposition 2.19:) *Provide a specification of a random feasible-sized object that has no pseudo-implementation, while relying on a standard intractability assumption.*

Let us digress and consider close-implementations. For example, Bach's elegant algorithm for generating random composite numbers along with their factorization [3] can be cast as a (non-trivial) close-implementation of the said distribution.[6] A more elementary set of examples refers to the generation of integers (out of a huge domain) according to various "nice" distributions (e.g., the binomial distribution of N trials).[7] In fact, Knuth [19, Sec. 3.4.1] considers the generation of various such distributions, and his treatment (of integer-valued distributions) can be easily adapted to fit our formalism. This direction is further pursued in our technical report [14]. In general, recall that in the current context (where the observer sees the entire object), a close-implementation must be statistically close to the specification. Thus, almost-truthfulness follows "for free":

Proposition 2.21 *Any close-implementation of a specification of a feasible-sized object is almost-truthful to it.*

Multiple samples. Our general formulation can be used to specify an object that whenever invoked returns an independently drawn sample from the same distribution. Specifically, the specification may be by a machine that answers each "sample-query" by using a distinct portion of its random-tape (as coins used to sample from the basic distribution). Using a pseudorandom function, we may pseudo-implement multiple samples from any distribution for which one can pseudo-implement a single sample. That is:

Proposition 2.22 *Suppose that one-way functions exist, and let $D = \{D_n\}$ be a probability ensemble such that each D_n ranges over* $\mathrm{poly}(n)$*-bit long strings. If D can be pseudo-implemented then so can the specification that answers each query by an independently selected sample of D. Furthermore, the latter implementation is by an ordinary machine and is truthful provided that the former implementation is truthful.*

[5] We stress that the assumption used here (i.e., the existence of a *single* collision-free hash function) seems stronger than the standard assumption that refers to the existence of an *ensemble* of collision-free functions (cf. [8]).

[6] We mention that Bach's concrete motivation was to generate prime numbers P along with the factorization of $P-1$, in order to allow efficient testing of whether a given number is a primitive element modulo P. Thus, one may say that Bach's paper provides a close-implementation (by an ordinary probabilistic polynomial-time machine) of the specification that selects at random an n-bit long prime P and answers the query g by 1 if and only if g is a primitive element modulo P.

[7] That is, for a huge $N = 2^n$, we want to generate i with probability $p_i \stackrel{\text{def}}{=} \binom{N}{i}/2^N$. Note $i \in \{0, 1, \ldots N\}$ has feasible size, and yet the problem is not trivial (because we cannot afford to compute all p_i's).

3 Our Main Results

We obtain several new implementations of random objects. For sake of clarity, we present the results in two categories referring to whether they yield truthful or only almost-truthful implementations. Here we only state the results, whereas their proofs appear in our technical report [14].

3.1 Truthful Implementations

All implementations stated in this section are by (polynomial-time) *oracle machines* (which use a random oracle). Corresponding pseudo-implementations by ordinary (probabilistic polynomial-time) machines can be derived using Theorem 2.9. Namely, assuming the existence of one-way functions, *each of the specifications considered below can be pseudo-implemented in a truthful manner by an ordinary probabilistic polynomial-time machine*.

The basic technique underlying the following implementations is the embedding of additional structure that enables to efficiently answer the desired queries in a consistent way or to force a desired property. That is, this additional structure ensures truthfulness (with respect to the specification). The additional structure may cause the implementation to have a distribution that differs from that of the specification, but this difference is infeasible to detect (via the polynomially-many queries). In fact, the additional structure is typically randomized in order to make it undetectable, but each possible choice of coins for this randomization yields a "valid" structure (which in turn ensures truthfulness rather than only almost-truthfulness).

3.1.1 Supporting complex queries regarding boolean functions

As mentioned above, a random boolean function is trivially implemented (and in a perfect way) by an oracle machine. By this we mean that the specification and the implementation merely serve the standard evaluation queries that refer to a random function (i.e., query x is answered by the value of the function at x). Here we consider specifications that supports more powerful queries.

Example 3.1 (answering some parity queries regarding a random function): *Consider a specification by a machine* (*and length parameter* $\ell = 2n$) *that, on input* (i, j) *where* $1 \le i \le j \le 2^n$, *replies with the parity of the bits in locations* i *through* j *of its random-tape. Intuitively, this machine specifies an object that, based on a random function* $f : [2^n] \to \{0, 1\}$, *provides the parity of the values of* f *on any desired interval of* $[2^n]$.

Clearly, the implementation cannot afford to compute the parity of the corresponding values in its random oracle. We present a perfect implementation of Example 3.1, as well as truthful close-implementations of more general types of random objects (i.e., answering any symmetric "interval" query). (See details in Appendix B.) Specifically, we prove:

Theorem 3.2 *For every polynomial-time computable function* g*, there exists a truthful close-implementation of the following specification of a random object. The specification machine uses its random-tape to define a random function* $f : \{0, 1\}^n \to \{0, 1\}$, *and answers the query* $(\alpha, \beta) \in \{0, 1\}^{n+n}$ *by* $g(\sum_{\alpha \le s \le \beta} f(s))$.

It would be interesting to further extend the above result; specific suggestions are made in our technical report [14].

3.1.2 Supporting complex queries regarding length-preserving functions

We consider specifications that, in addition to the standard evaluation queries, answer additional queries regarding a random length-preserving function. Such objects have potential applications in computational number theory, cryptography, and the analysis of algorithms (cf. [10]). Specifically, we prove:

Theorem 3.3 *There exists a truthful close-implementation of the following specification. The specifying machine, uniformly selects a function* $f : \{0, 1\}^n \to \{0, 1\}^n$, *and, in addition to the standard evaluation queries, answers the inverse-query* $y \in \{0, 1\}^n$ *with the set* $f^{-1}(y)$.

Alternatively, the implementation may answer with a uniformly distributed preimage of y under f (and with a special symbol in case no such preimage exists).

Theorem 3.4 *There exists a truthful close-implementation of the following specification. The specifying machine, uniformly selects a function* $f : \{0, 1\}^n \to \{0, 1\}^n$, *and answers the query* (x, m), *where* $x \in \{0, 1\}^n$ *and* $m \in [2^{\text{poly}(n)}]$, *with the value* $f^m(x)$ (*i.e.,* f *iterated* m *times on* x).

This result is related to questions studied in [22, 23]; for more details, see our technical report [14].

3.1.3 Random graphs of various types

Random graphs have been extensively studied (cf. [6]), and in particular are known to have various properties. But does it mean that we can provide *truthful* close-implementations of uniformly distributed (huge) graphs having any of these properties?

Let us first consider a specification for a random N-vertex graph, where $N = 2^n$. Indeed, such a random graph can be specified by the machine, which viewing its random-tape ω as an N-by-N matrix, answers input $(i, j) \in [N] \times [N]$ with the value 0 if $i = j$, the value $\omega_{i,j}$ if $i < j$, and $\omega_{j,i}$ otherwise. But how about implementing a uniformly distributed graph that has various properties?

Example 3.5 (uniformly distributed connected graphs): *Suppose that we want to implement a uniformly distributed* connected *graph* (*i.e., a graph uniformly selected among all connected* N*-vertex graph*). *An adequate specification may scan its random-tape, considering each* N^2*-bit long portion of it as a description of a graph, and answer adjacency-queries according to the first portion that yields a connected graph. Note that the specification works in time* $\Omega(N^2)$, *whereas an implementation needs to work in* poly$(\log N)$*-time. On the other hand, recall that a random graph is connected with overwhelmingly high probability. This suggests to implement a random connected graph by a random graph. Indeed, this yields a close-implementation,* but not a truthful one (*because occasionally, yet quite rarely, the implementation will yield an unconnected graph*).[8]

We present truthful close-implementations of Example 3.5 as well as of related specifications (i.e., of uniformly distributed graphs having various additional properties). These are all special cases of the following result:

Theorem 3.6 *Let* Π *be a* monotone *graph property that is satisfied by a family of strongly-constructible* sparse *graphs. That is, for some negligible function* μ (*and every* N), *there exists a perfect implementation of a* (single) N*-vertex graph with* $\mu(\log N) \cdot N^2$ *edges that satisfies property* Π. *Then, there exists a truthful close-implementation of a uniformly distributed graph that satisfies property* Π.

The proof relies on the following lemma, which may be of independent interest. Loosely speaking, the lemma asserts that if a monotone graph property Π is satisfied by some sparse graphs then a uniformly distributed graph having property Π is indistinguishable from a truly random graph.

[8] Indeed, the trivial implementation (by a random graph) is almost-truthful, but here we seek a truthful implementation (because otherwise we cannot derive from it (via Theorem 2.9) even an almost-truthful pseudo-implementation by an ordinary machine).

Lemma 3.7 *Let Π be a* monotone *graph property that is satisfied by some N-vertex graph having $\epsilon \cdot \binom{N}{2}$ edges. Then, any machine that makes at most q adjacency queries to a graph, cannot distinguish a random N-vertex graph from a uniformly distributed N-vertex graph that satisfies Π, except than with probability $O(q\sqrt{\epsilon}) + qN^{-(1-o(1))}$.*

3.1.4 Supporting complex queries regarding random graphs

Suppose that we want to implement a random N-vertex graph along with supporting, in addition to the standard adjacency queries, also some complex queries that are hard to answer by only making adjacency queries. For example suppose that on query a vertex v, we need to provide a clique of size $\log_2 N$ containing v. We present a truthful close-implementations of this specification:

Theorem 3.8 *There exists a truthful close-implementation of the following specification. The specifying machine selects uniformly an N-vertex graph and, in addition to the standard adjacency queries, answers* (Log-Clique) queries *of the form v by providing a random $\lceil \log_2 N \rceil$-vertex clique that contains v (and a special symbol if no such clique exists).*

Another result of similar flavour follows:

Theorem 3.9 *There exists a truthful close-implementation of the following specification. The specifying machine selects uniformly an N-vertex graph G, and in case G is Hamiltonian it uniformly selects a (directed) Hamiltonian Cycle in G, which in turn defines a cyclic permutation $\sigma : [N] \to [N]$. In addition to the standard adjacency queries, the specification answers* travel queries *of the form (trav, v, t) by providing $\sigma^t(v)$, and* distance queries *of the form (dist, v, w) by providing the smallest $t \geq 0$ such that $w = \sigma^t(v)$.*

3.1.5 Random bounded-degree graphs of various types

Random bounded-degree graphs have also received considerable attention. We present truthful close-implementations of random bounded-degree graphs $G = ([N], E)$, where the machine specifying the graph answers the query $v \in [N]$ with the list of neighbors of vertex v. We stress that even implementing this specification is non-trivial if one insists on truthfully implementing *simple* random bounded-degree graphs (rather than graphs with self-loops and/or parallel edges). Furthermore, we present truthful close-implementations of random bounded-degree graphs having additional properties such as connectivity, Hamiltonicity, having logarithmic girth, etc. All these are special cases of the following result:

Theorem 3.10 *Let d be fixed and Π be a graph property that satisfies the following two conditions:*

1. *The probability that Property Π is not satisfied by a uniformly chosen d-regular N-vertex graph is negligible in $\log N$.*
2. *Property Π is satisfied by a family of strongly-constructible d-regular N-vertex graphs having girth $\omega(\log \log N)$.*

Then, there exists a truthful close-implementation of a uniformly distributed d-regular N-vertex graph that satisfies property Π.

The proof relies on the following lemma, which may be of independent interest. Loosely speaking, the lemma asserts that a random isomorphic copy of a fixed d-regular graph of large girth is indistinguishable from a truly random d-regular graph.

Lemma 3.11 *let $G = ([N], E)$ be any d-regular N-vertex graph having girth g. Let G' be obtained by randomly permuting the vertices of G (and presenting the incidence lists in some canonical order). Then, any machine M that queries the graph for the neighborhoods of q vertices of its choice, cannot distinguish G' from a random d-regular N-vertex (simple) graph, except than with probability $O(q^2/(d-1)^{(g-1)/2})$. In the case $d = 2$ and $q < g-1$, the probability bound can be improved to $O(q^2/N)$.*

3.2 Almost-Truthful Implementations

All implementations stated in this section are by *ordinary* (probabilistic polynomial-time) machines. All these results *assume the existence of one-way functions*.

Again, the basic technique is to embed a desirable structure, but (in contrast to Section 3.1) here the embeded structure forces the desired property only with very high probability. Consequently, the resulting implementation is only almost-truthful, which is the reason that we have to directly present implementations by ordinary machines.

A specific technique that we use is obtaining a function as a value-by-value combination of a pseudorandom function and a function of a desired combinatorial structure. The combination is done such that the combined function inherits both the pseudorandomness of the first function and the combinatorial structure of the second function (in analogy to a construction in [18]). In some cases, the combination is by a value-by-value XOR, but in others it is by a value-by-value OR with a second function that is very sparse.

3.2.1 Random codes of large distance

In continuation to the discussion in the introduction, we prove:

Theorem 3.12 *For $\delta = 1/6$ and $\rho = 1/9$, assuming the existence of one-way functions, there exists an almost-truthful pseudo-implementation of the following specification: The specification machine uses its random-tape to uniformly select a code $C \subset \{0,1\}^n$ having cardinality $K \stackrel{\text{def}}{=} 2^{\rho n}$ and distance at least δn, and answers the query $i \in [K]$ with the i-th element in C.*

We comment that the above actualy specifies (and implements) an encoding algorithm for the corresponding code. It would be very interesting if one can also implement a corresponding decoding algorithm; see further discussion in our technical report [14].

3.2.2 Random graphs of various types

Having failed to provide truthful pseudo-implementations to the following specifications, we provide almost-truthful ones.

Theorem 3.13 *Let $c(N) = (2 \pm o(1)) \log_2 N$ be the largest integer i such that the expected number of cliques of size i in a random N-vertex graph is larger than one. Assuming the existence of one-way functions, there exist almost-truthful pseudo-implementations of the following specifications:*

1. A random graph of Max-Clique $c(N) \pm 1$: *The specification uniformly selects an N-vertex graph having maximum clique size $c(N) \pm 1$, and answers edge-queries accordingly.*
2. A random graph of Chromatic Number $(1 \pm o(1)) \cdot N/c(N)$: *The specification uniformly selects an N-vertex graph having Chromatic Number $(1 \pm \log_2^{-1/3} N) \cdot N/c(N)$, and answers edge-queries accordingly.*

Another interesting question is to provide an almost-truthful pseudo-implementation of a uniformly distributed graph having a high (global) connectivity property. Unfortunately, we do not know how to do this. Instead, we provide an almost-truthful pseudo-implementation of a random graph for which almost all pairs of vertices enjoy a high connectivity property.

Theorem 3.14 *For every positive polynomial p, assuming the existence of one-way functions, there exists an almost-truthful pseudo-implementation of the following specification. The specifying machine selects a graph that is uniformly distributed among all N-vertex graphs for which all but at most an $\epsilon(N) \stackrel{\text{def}}{=} 1/p(\log_2 N)$ fraction of the vertex pairs are connected by at least $(1 - \epsilon(N)) \cdot N/2$ vertex-disjoint paths. Edge-queries are answered accordingly.*

Interestingly, the same implementation works for all polynomials p; that is, the implementation is independent of p, which is only used in the definition of the specification.

4 Conclusions and Open Problems

The questions that underlie our work refer to the existence of good implementations of various specifications. At the very least, we require the implementations to be computationally-indistinguishable from the corresponding specifications.[9] That is, we are interested in pseudo-implementations. Our ultimate goal is to obtain such implementations via ordinary (probabilistic polynomial-time) machines, and so we ask:

Q1: Which specifications have truthful pseudo-implement ations (by ordinary machines)?

Q2: Which specifications have almost-truthful pseudo-implementations (by ordinary machines)?

Q3: Which specifications have pseudo-implementations at all?

In view of Theorem 2.9, as far as Questions Q1 and Q3 are concerned, we may as well consider implementations by oracle machines (having access to a random oracle). Indeed, the key observation that started us going was that the following questions are the "right" ones to ask:

Q1r (Q1 revised)**:** Which specifications have truthful close-implementations by oracle machines (having access to a random oracle)?

Q3r (Q3 revised)**:** Which specifications have such close-implementations at all?

We remark that even in case of Question Q2, it may make sense to study first the existence of implementations by oracle machines, bearing in mind that the latter cannot provide a conclusive positive answer (as shown in Theorem 2.11).

In this work, we have initiated a comprehensive study of the above questions. In particular, we provided a fair number of non-trivial implementations of various specifications relating to the domains of random functions, random graphs and random codes. The challenge of characterizing the class of specifications that have good implementations (e.g., Questions Q1r and Q3r) remains wide open. A good start may be to answer such questions when restricted to interesting classes of specifications (e.g., the class of specifications of random graphs having certain type of properties).

Limited-independence implemenations. Our definition of pseudo-implementation is based on the notion of computational indistinguishability (cf. [16, 24, 13]) as a definition of similarity among objects. A different notion of similarity

[9] Without such a qualification, the questions stated below are either meaningless (i.e., every specification has a "bad" implementation) or miss the point of generating random objects.

underlies the construction of sample spaces having limited-independence properties (see, e.g., [2, 7]). For example, we say that an implementation is k-wise close to a give specification if the distribution of the answers to any k fixed queries to the implementation is staistically close to the distribution of these answers in the specification. The study of Question Q1r is also relevant to the construction of truthful k-wise close implementations, for any $k = \text{poly}(n)$. In particular, one can show that *any specification that has a truthful close-implementation by an oracle machine, has a truthful k-wise close implementation by an ordinary probabilistic polynomial-time machine*. A concrete example which is useful for streaming applications (i.e., a "range-summable" sequence [9] of k-wise close random variables) appears in our technical report [14] (following the proof of Theorem 3.2).

Acknowledgments

The first two authors wish to thank Silvio Micali for discussions that took place two decades ago. The main part of Theorem 2.9 was essentially observed in these discussions. These discussions reached a dead-end because the notion of a specification was missing (and so it was not understood that the interesting question is which specifications can be implemented at all (i.e., even by an oracle machine having access to a random function)).

We are grateful to Noga Alon for very helpful discussions regarding random graphs and explicit constructions of bounded-degree graphs of logarithmic girth. We also thank Avi Wigderson for a helpful discussion regarding the proof of Lemma 3.7.

References

[1] M. Abadi, E. Allender, A. Broder, J. Feigenbaum, and L. Hemachandra. On Generating Hard, Solved Instances of Computational Problem. In *Crypto88*, pages 297–310.

[2] N. Alon, L. Babai and A. Itai. A fast and Simple Randomized Algorithm for the Maximal Independent Set Problem. *J. of Algorithms*, Vol. 7, pages 567–583, 1986.

[3] E. Bach. *Analytic Methods in the Analysis and Design of Number-Theoretic Algorithms*. ACM Distinguished Dissertation (1984), MIT Press, Cambridge MA, 1985.

[4] S. Ben-David, B. Chor, O. Goldreich, and M. Luby. On the Theory of Average Case Complexity. *JCSS*, Vol. 44, No. 2, 1992, pages 193–219. Preliminary version in *21st STOC*, 1989.

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

[6] B. Bollobas. *Random Graphs*. Academic Press, 1985.

[7] B. Chor and O. Goldreich. On the Power of Two–Point Based Sampling. *Jour. of Complexity*, Vol 5, 1989, pages 96–106. Preliminary version dates 1985.

[8] I. Damgård. Collision Free Hash Functions and Public Key Signature Schemes. In *EuroCrypt'87*, Springer-Verlag, LNCS 304, pages 203–216.

[9] J. Feigenbaum, S. Kannan, M. Strauss, M. Viswanathan. An Approximate L1-Difference Algorithm for Massive Data Streams. Proceedings of *40th FOCS*, pages 501–511, 1999.

[10] P. Flajolet and A.M. Odlyzko. Random mapping statistics. In *EuroCrypt'89*, Springer-Verlag, LNCS 434, pages 329–354.

[11] O. Goldreich. A Note on Computational Indistinguishability. *IPL*, Vol. 34, pages 277–281, May 1990.

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

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

[14] O. Goldreich, S. Goldwasser, and A. Nussboim. On the Implementation of Huge Random Objects. *ECCC*, TR03-045, 2003.

[15] O. Goldreich, and H. Krawczyk, On Sparse Pseudorandom Ensembles. *Random Structures and Algorithms*, Vol. 3, No. 2, (1992), pages 163–174.

[16] S. Goldwasser and S. Micali. Probabilistic Encryption. *JCSS*, Vol. 28, No. 2, pages 270–299, 1984. Preliminary version in *14th STOC*, 1982.

[17] J. Håstad, R. Impagliazzo, L.A. Levin and M. Luby. A Pseudorandom Generator from any One-way Function. *SICOMP*, Volume 28, Number 4, pages 1364–1396, 1999. Preliminary versions by Impagliazzo et. al. in *21st STOC* (1989) and Håstad in *22nd STOC* (1990).

[18] R. Impagliazzo and A. Wigderson. P=BPP if E requires exponential circuits: Derandomizing the XOR Lemma. In *29th STOC*, pages 220–229, 1997.

[19] D.E. Knuth. *The Art of Computer Programming*, Vol. 2 (*Seminumerical Algorithms*). Addison-Wesley Publishing Company, Inc., 1969 (first edition) and 1981 (second edition).

[20] L.A. Levin. Average Case Complete Problems. *SICOMP*, Vol. 15, pages 285–286, 1986.

[21] M. Luby and C. Rackoff. How to Construct Pseudorandom Permutations from Pseudorandom Functions. *SICOMP*, Vol. 17, 1988, pages 373–386.

[22] M. Naor and O. Reingold. Constructing Pseudo-Random Permutations with a Prescribed Structure, *Jour. of Crypto.*, Vol. 15 (2), 2002, pages 97–102.

[23] B. Tsaban. Permutation graphs, fast forward permutations, and sampling the cycle structure of a permutation. *Journal of Algorithms*, Vol. 47 (2), pages 104–121, 2003.

[24] A.C. Yao. Theory and Application of Trapdoor Functions. In *23rd FOCS*, pages 80–91, 1982.

Appendix A: Another illustrative Example

Suppose that you want to run some simulations on huge random graphs. You actually take it for granted that the random graph is going to be Hamiltonian, because you have read Bollobas's book [6] and you are willing to discard the negligible probability that a random graph is not Hamiltonian. Suppose that you want to be able to keep succinct representations of these graphs and/or that you want to generate them using few random bits. Having also read some works on pseudorandomness (e.g., [16, 5, 24, 13]), you plan to use pseudorandom functions [13] in order to efficiently generate and store representations of these graphs. *But wait a minute, are the graphs that you generate this way really Hamiltonian?*

The point is that being Hamiltonian is a global property of the graph, which in turn is a huge (i.e., exp(n)-sized) object. This global property cannot be checking the adjacency of polynomially many (i.e., poly(n)-many) vertex-pairs, and so its violation cannot be translated to a contradiction of the pseudorandomness of the function. Indeed, the substitution of a random function (or a random graph) by a pseudorandom one is not guaranteed to preserve the global property. Specifically, it may be the case that all pseudorandom graphs are even disconnected.[10] So, *can we efficiently generate huge Hamiltonian graphs?* As we show in this work, the answer to this question is positive.

Appendix B: A Perfect Implementation of Example 3.1

In this appendix we show that the specification of Example 3.1 can be perfectly implemented (by an oracle machine). Recall that we seek to implement access to a random function $f : \{0,1\}^n \rightarrow \{0,1\}$ augmented with answers regarding the parity (or XOR) of the values of f on given intervals, where the intervals are with respect to the standard lex-order of n-bit string. That is, the query $q = (\alpha, \beta) \in \{0,1\}^{n+n}$, where $0^n \leq \alpha \leq \beta \leq 1^n$, is to be answered by $\oplus_{\alpha \leq s \leq \beta} f(s)$. The specification can answer this query in the straightforward manner, but an implementation cannot afford to do so (because a straightforward computation may take $2^n = 2^{|q|/2}$ steps). Thus, the implementation will do something completely different.

We present an oracle machine that uses a random function $f' : \cup_{i=0}^{n} \{0,1\}^i \rightarrow \{0,1\}$. Using f', we define $f : \{0,1\}^n \rightarrow \{0,1\}$ as follows. We consider a binary tree of depth n and associate its ith level vertices with strings of length i such that the vertex associated with the string s has a left (resp., right) child associated with the string $s0$ (resp., $s1$). As a mental experiment, going from the root to the leaves, we label the tree's vertices as follows:

1. We label the root (i.e., the level-zero vertex, which is associated with λ) by the value $f'(\lambda)$.

2. For $i = 0, ..., n-1$, and each internal vertex v at level i, we label its *left* child by the value $f'(v0)$, and label its *right* child by the XOR of the label of v and the value $f'(v0)$. (Thus, the label of v equals the XOR of the values of its children.)

3. The value of f at $\alpha \in \{0,1\}^n$ is defined as the label of the leaf associated with α.

By using induction on $i = 0, ..., n$, it can be shown that the level i vertices are assigned uniformly and independently distributed labels (which do depend, of course, on the level $i-1$ labels). Thus, f is a random function. Furthermore, the label of each internal node v equals the XOR of the values of f on all leaves in the subtree rooted at v.

Note that the random function f' is used to directly assign (random) labels to all the left-siblings. The other labels (i.e., of right-siblings) are determined by XORing the labels of the parent and the left-sibling. Furthermore, the label of each node in the tree is determined by XORing at most $n+1$ values of f' (residing in appropriate left-siblings). Specifically, the label of the vertex associated with $\sigma_1 \cdots \sigma_i$ is determined by the f'-values of the strings $\lambda, 0, \sigma_1 0, ..., \sigma_1 \cdots \sigma_{i-1} 0$. Thus, we obtain the value of f at any n-bit long string by making at most $n+1$ queries to f'. More generally, we can obtain the label assigned to each vertex by making at most $n+1$ queries to f'. It follows that we can obtain the value of $\oplus_{\alpha \leq s \leq \beta} f(s)$ by making $O(n^2)$ queries to f'. Specifically, the desired value is the XOR of the leaves residing in at most $2n-1$ full binary sub-trees, and so we merely need to XOR the labels assigned to the roots of these sub-trees. Actually, $O(n)$ queries can be shown to suffice, by taking advantage on the fact that we need not retrieve the labels assigned to $O(n)$ arbitrary vertices (but rather to vertices that correspond to roots of sub-trees with consecutive leaves). We get get *a perfect implementation (by an oracle machine) of the specification of Example 3.1.*

The above procedure can be generalize to handle queries regarding any (efficiently computable) symmetric function of the values assigned by f to any given interval. In fact, it suffices to answer queries regarding the sum of these values. This yields a proof of Theorem 3.2.

[10] Indeed, for each function f_s taken from some pseudorandom ensemble $\{f_s : [2^n] \times [2^n] \rightarrow \{0,1\}\}_s$, it *may* hold that $f_s(v_s, u) = f_s(u, v_s) = 0$ for all $u \in [2^n]$, where v_s depends arbitrarily on f_s. For example, given a pseudorandom ensemble $\{f_s\}$ consider the ensemble $\{f_{s,v}\}$ such that $f_{s,v}(v,u) = f_{s,v}(u,v) = 0^n$ for all u's, and $f_{s,v}(x,y) = f_s(x,y)$ for all other (x,y)'s.