2016 IEEE 57th Annual Symposium on Foundations of Computer Science

Amplification and Derandomization Without Slowdown

Ofer Grossman
Department of EECS
MIT
Cambridge, MA
ofer.grossman@gmail.com

Dana Moshkovitz
Department of Computer Science
UT Austin
Austin, TX
danama@cs.utexas.edu

***Abstract*—We present techniques for decreasing the error probability of randomized algorithms and for converting randomized algorithms to deterministic (non-uniform) algorithms. Unlike most existing techniques that involve repetition of the randomized algorithm and hence a slowdown, our techniques produce algorithms with a similar run-time to the original randomized algorithms. The amplification technique is related to a certain stochastic multi-armed bandit problem. The derandomization technique – which is the main contribution of this work – points to an intriguing connection between derandomization and sketching/sparsification. We demonstrate the techniques by showing algorithms for approximating free games (constraint satisfaction problems on dense bipartite graphs).**

***Keywords*-Amplification; derandomization; Free game; multi-armed bandit; sketching;**

I. Introduction

Randomized algorithms are by now ubiquitous in algorithm design. With certain probability – called their *error probability* – they do not output the correct answer within the allotted time. In return, they may have gains in efficiency, or *speedup*, over the best known deterministic algorithms, at least for some problems. At the same time, for many other problems, the design of an efficient randomized algorithm is eventually followed by the design of an equally efficient deterministic algorithm. “Derandomization” is achieved either by one of a few general derandomization methods, or by solutions tailored to the problem at hand. Unfortunately, most general derandomization methods incur a *slowdown*, i.e., a loss in efficiency in the deterministic algorithm compared to the randomized algorithm. For instance, the main approach to derandomization is by designing a *pseudorandom generator* and invoking the algorithm on all its seeds, which slows down the deterministic algorithm by a factor equal to the number of seeds. In general, the number of seeds is likely at least linear in the number of pseudorandom bits needed [18], yielding a substantial slowdown.

This material is based upon work supported by the National Science Foundation under grants number 1218547 and 1648712.

Closely related is the slowdown incurred by decreasing the error probability of a randomized algorithm to a non-zero quantity. Given a randomized algorithm that has error probability $1/3$, we can construct a randomized algorithm with error probability $2^{-\Omega(k)}$ by repeating the algorithm k times. However, the resulting algorithm is slower by a factor of k than the original algorithm, which is significant when k is large (for instance, consider k that equals the input size n, or equals n^{ϵ} for some constant $\epsilon > 0$). One could save in randomness, implementing k-repetition roughly with the number of random bits required for a single repetition [20], but the number of invocations – and hence the run-time – provably remains large (see, e.g., [14]).

In this work we develop general methods for derandomization and error reduction that do not incur a substantial slowdown. Specifically, we give positive answers in certain cases to the following questions:

- Amplification: Can we decrease the error probability of a randomized algorithm without substantially increasing its run-time?
- Derandomization: Can we convert a randomized algorithm to a deterministic (non-uniform) algorithm without substantially increasing its run-time?

The increase in the run-time, or *slowdown*, in our applications is poly-logarithmic in the input size, beating most existing derandomization methods (we provide a detailed comparison in Section I-C). In some cases, our methods may yield only a constant slowdown. Our derandomization method yields non-uniform algorithms. We explain the reason when we describe the method in Section I-A and we discuss the importance of non-uniform algorithms in Section I-B.

The methods themselves are quite different from commonly used methods. Ironically, they employ ideas rooted in the study of randomized algorithms, like sketching and stochastic multi-armed bandit problems. In the full version of the paper [16], we demonstrate the utility of the methods by deriving improved algorithms for problems like finding a dense set in a graph that contains a large clique, finding an approximate max-cut in a dense graph with a large cut, approximating constraint satisfaction problems on dense graphs ("free games") and going from Reed-Muller list decoding to unique decoding. We hope that the methods will find more applications in the future.

A. Derandomization From Sketching

Our derandomization method is based on the derandomization method of Adleman [2]. Adleman's method works in a black-box fashion for all algorithms, and generates a non-uniform deterministic algorithm that is slower by a linear factor than the randomized algorithm. Our method works for many, but not all, algorithms, and generates a non-uniform deterministic algorithm with a significantly smaller slowdown. In this section we recall Adleman's method and discuss our method.

Adleman's idea (somewhat modified from its original version) is as follows. Suppose there exists some algorithm A which solves our problem with error probability $\frac{1}{3}$. We create a new algorithm B, which runs algorithm A in series $\Theta(n)$ times (where n is the input size). We then output the majority of the outputs of the executions of A. By the Chernoff bound, the error probability of algorithm B can be made less than 2^{-n}. By a union bound over all 2^n inputs, we see that there must exist some choice of randomness r such that algorithm B succeeds for all inputs of length n given the randomness r. The randomness string r can be hard-wired to a non-uniform algorithm as an advice string. We note that algorithm B is slower than algorithm A by a factor of $\Theta(n)$.

Our methods will allow us to reduce this $\Theta(n)$ slowdown of Adleman's technique. The principle behind the method is simple. Fix a randomized algorithm A that we wish to derandomize. In Adleman's technique, we amplified the error probability below 2^{-n}, and then used a union bound on all inputs. Instead of applying a union bound on the 2^n possible inputs, we partition the inputs into $2^{n'}$ sets where $n' \ll n$, such that inputs in the same part have mostly the same successful randomness strings (a randomness string is *successful* for an input if the algorithm is correct for the input when using the randomness string). One can think of inputs in the same part as inputs on which the algorithm A behaves similarly with respect to the randomness[1]. Then, one can perform the union bound from Adleman's proof over the $2^{n'}$ different parts, instead of over the 2^n inputs. It suffices that the error probability is lower than $2^{-n'}$ (i.e., for every part, the fraction of common successful randomness strings is larger than $1-2^{-n'}$) to deduce the existence of a randomness string on which the algorithm is correct for all inputs. Therefore, the only slowdown that is incurred is the one needed to get the error probability below $2^{-n'}$, and not the one needed to get the error probability below 2^{-n}.

It's surprising that this principle can be useful for algorithms that may access any bit of their input. Our contribution is in setting up a framework for arguing about partitions as above, and then using the framework to derive desired partitions for various algorithms. The framework is based on *sketching* and *oblivious verification*.

We associate an n'-bit string with every part, and think of it as a *sketch* (a lossy, compressed version) of the inputs in the part. In our applications the input is often a graph, and the sketch is a small "representative" sub-graph, whose existence we argue by analyzing a probabilistic construction. Note that there are no computational limitations on the computation of the sketch, only the information-theoretic bound of n' bits.

To argue about inputs with the same sketch having common successful randomness strings we design an *oblivious verifier* for the algorithm, as we define next. The task of an oblivious verifier is to test whether the algorithm works for some input and randomness string *given only the sketch of the input*. This means that the verifier cannot in general simulate the algorithm. However, unlike the algorithm, we impose no computational limitations on the verifier. The verifier's mere existence proves that the randomized algorithm behaves similarly on inputs with the same sketch as we wanted. The verifier should satisfy:

1) If the verifier accepts a randomness string and a sketch of an input, then the algorithm necessarily succeeds on the input using the randomness string.
2) For every sketch, the verifier accepts almost all[2] randomness strings.

It is not difficult to check that the existence of an oblivious verifier is equivalent to the existence of a partition of the inputs as above and hence to a saving in Adleman's union bound. The difficulty lies in the design of sketches and oblivious verifiers. In our opinion, it is

[1]We'd like to emphasize that the algorithm usually distinguishes inputs in the same part: its execution and output are different for different inputs. The similar behavior is only in terms of which randomness strings are successful.

[2]I.e., more than $1-2^{-n'}$ fraction. To achieve this, it suffices that the error probability is not much higher than $2^{-n'}$, since in this case we can use repetition to bring the error probability below 2^{-n} without incurring much slowdown.

surprising that the algorithms we consider – ordinary algorithms that require full access to their input – can be verified obliviously, and indeed we work quite hard to design oblivious verifiers. Our oblivious verifiers often take the form of repeatedly verifying that certain key steps of the algorithm work as expected, at least approximately, using the sketch. In addition – since the verifier cannot access the input and therefore cannot simulate the algorithm – the verifier exhaustively checks all possible branchings of the algorithm. Interestingly, the algorithm, the verifier and the sketch are typically all randomized, and yet the argument above shows that they yield a deterministic (non-uniform) algorithm.

B. The Significance of Non-Uniform Algorithms

Like Adleman's method, our derandomization method produces non-uniform algorithms, i.e., sequences of algorithms, one for each input size. Since the input size is known, the algorithm can rely on an "advice" string depending on the input size, though this advice string may be hard to compute. This is different than the usual, uniform, model of computation, where the same algorithm works for all input sizes.

In this section we discuss non-uniformity and several connections between non-uniform and uniform algorithms. *Below we only refer to non-uniform algorithms that (i) if given a correct advice, are correct on all their inputs. (ii) If given an incorrect advice, may either be correct on their input or output $\perp$ (but never an incorrect output).* This is true of all the non-uniform algorithms we consider in this work. For some of the items below this requirement can be relaxed.

An intriguing open problem in derandomization is about non-uniform algorithms: The problem of finding an optimal deterministic minimum spanning tree algorithm is known to be equivalent to the problem of finding an optimal deterministic *non-uniform* algorithm [26]. Indeed, this was the original motivation for our work. For any problem for which inputs of size n can be reduced to many inputs of size a where a is sufficiently smaller than $\log n$ ("downward self-reduction"), a deterministic non-uniform algorithm implies a deterministic uniform algorithm with essentially the same run-time. The reason is that one can find the advice string for input size a using brute force (checking all possible advice strings and all possible inputs) in sub-linear time in n. Next one can solve the many inputs of size a using the advice. Many problems have downward self-reductions including minimum spanning tree, matrix multiplication and 3SUM [7].

Amortization gets rid of non-uniformity: If one needs to solve a problem on a long sequence of inputs, much longer than the number of possible advice strings[3], one can amortize the cost searching for the correct advice over all possible inputs. We start the first input with the first advice string. If the algorithm is incorrect with the current advice string, it moves on to the next one.

Preprocessing gets rid of non-uniformity: A non-uniform algorithm can be simulated by a uniform algorithm and a preprocessing step to find a correct advice.

Non-determinism gets rid of non-uniformity: In complexity theory one often wishes to argue about uniform *non-deterministic* algorithms. Such algorithms can guess the advice string, invoke the algorithm on the advice, and then verify the output.

Non-uniformity as a milestone: An efficient non-uniform deterministic algorithm gives evidence that such efficiency is possible deterministically, and may eventually lead to an efficient uniform deterministic algorithm.

Non-uniformity is natural: In real life often we do have a bound on the input size. If one can design better algorithms using this bound, that's something we'd like to know

C. Comparison With Other Derandomization Methods

There are two main existing methods for derandomization: the method of conditional probabilities and pseudorandom generators. In the method of conditional probabilities one derandomizes by fixing the random bits one after the other. This is possible when there is a way to quickly assess the quality of large subsets of randomness strings. This is the case, for instance, when searching for an assignment that satisfies $7/8$ fraction of the clauses in a 3SAT formula. Interestingly, the method typically incurs no slowdown (see, e.g., [23]). However, it is useful only in very specific cases, and – in a sense – when it's useful, it shows that the randomization was only a conceptual device rather than an actual resource. In contrast, the current work is about incurring little slowdown for broader classes of randomized algorithms.

Perhaps the main method to derandomize algorithms is via pseudorandom generators (PRGs). These are constructions that expand a small seed to sufficiently many pseudorandom bits. The pseudorandom bits "look random" to the algorithm. This is possible when the algorithm uses the randomness in a "sufficiently weak" way. For instance, the very existence of an upper bound on the run-time of the algorithm implies a limitation on the algorithm's usage of randomness, since the

[3]This is especially useful if the space of possible advice strings is of polynomial size. In our case, this can be possible to arrange via pseudorandom generators as discussed in Section I-C.

algorithm cannot perform tests on the randomness that require more time than its run-time. Impagliazzo and Wigderson [19] capitalize on that to show how to (conditionally) construct PRGs that "fool" any randomized algorithm that runs in a fixed polynomial time. For some algorithms, k-wise independent generators or ϵ-biased generators may suffice.

The disadvantage of PRGs is that one needs try all seeds to derandomize the algorithm, causing a slowdown. The slowdown is likely at least linear in the number of random bits that the algorithm uses [18]. For k-wise independent generators the slowdown is at least $\Omega(n^k)$ [22]. One case when only a poly-logarithmic slowdown can be achieved is almost k-wise independent generators for small k [25]. For those generators the slowdown is only $\mathrm{poly}(\log n, 2^k)$, but they are only suitable for a limited class of algorithms. In this work we obtain a poly-logarithmic slowdown for algorithms that use their randomness in a much stronger way.

It is interesting to note the relation between derandomization from sketching and pseudorandom generators. In many senses the two methods are *dual*. In PRGs, one shrinks the space of possible randomness strings. In derandomization from sketching one shrinks the space of possible inputs. PRGs need to work against non-uniform algorithms (since the input to the algorithm may give it "advice" helping it distinguish pseudorandom bits from truly random bits), whereas derandomization from sketching produces non-uniform algorithms (now the successful randomness is the advice). Of course, it is possible to combine PRGs and derandomization from sketching, so, e.g., one can amortize the cost of searching for a successful randomness string over fewer inputs, or have reduced search cost in a preprocessing phase.

D. Amplification

Derandomization from sketching as discussed above only relaxes the amplification task - instead of requiring a randomized algorithm with error probability below 2^{-n} as in Adleman's method, it requires an algorithm with error probability below $2^{-n'}$. In our applications $n' \approx \sqrt{n}$, so amplification by repetition would still incur a large slowdown. In this section we discuss our approach to amplification with little slowdown. The method has a *poly-logarithmic* slowdown in k when it amplifies the error probability to $2^{-\Omega(k)}$, as opposed to standard repetition that has a slowdown that is linear in k. In fact, in certain situations the slowdown is only *constant*! However, unlike repetition, our method does not work in all cases. It requires a quick check that approximates probabilistically the quality of a randomness string given to it as input.

Fix an input to the randomized algorithm. Assign a "grade" in $[0, 1]$ to each randomness string indicating the quality of the randomness. A "quick check" is a randomized procedure that given the randomness string r accepts with probability equal to the grade of r. For example, suppose that the algorithm is given as input a graph, and its task is to find a cut that contains at least $1/2 - \epsilon$ fraction of the edges in the graph, for some constant ϵ. The algorithm uses its randomness to pick the cut. The grade of the randomness is the fraction of edges in the cut. The randomness checker picks a random edge in the graph and checks whether it is in the cut, which takes $O(1)$ time.

In general, if the run-time of the algorithm is T and it has a quick check that runs in time t, then we show how to decrease the error probability from $1/3$ to $\exp(-k)$ in time roughly $k \cdot t + T$ instead of $k \cdot T$ of repetition. This follows from an algorithm for a stochastic multi-armed bandit problem that we define. In this problem, which we call the *biased coin problem*, there is a large pile of coins, and $2/3$ fraction of the coins are biased, meaning that they fall on heads with high probability. The coins are unmarked and the only way to discover information about a coin is to toss it. The task is to find one biased coin[4] with certainty $1 - e^{-\Omega(k)}$ using as few coin tosses as possible. The analogy between the biased coin problem and amplification is that the coins represent possible randomness strings for the algorithm, many of which are good. The task is to find one randomness string that is good with very high probability. Tossing a coin corresponds to a quick check. We show how to find a biased coin using only $\tilde{O}(k)$ coin tosses. Moreover, when there is a gap between the grades of good randomness strings and the grades of bad randomness strings, we show that only $O(k)$ coin tosses suffice. The algorithm for finding a biased coin can be interpreted as an algorithm for searching the space of randomness strings in order to find a randomness string of high grade. The number of coin tosses determines the run-time of the algorithm.

The biased coin problem is related to the stochastic multi-armed bandit problem studied in [10], [24], however, in the latter there might be only one biased coin, whereas in our problem we are guaranteed that a constant fraction of the coins are biased. This makes a big difference in the algorithms one would consider for each problem and in their performance. In the setup considered by [10], [24] one has to toss all coins, and

[4]We allow the bias of the output coin to be slightly smaller than the bias of the $2/3$ fraction of the coins that have high bias.

the algorithms focus on which coins to eliminate. In our setup it is likely that we find a biased coin quickly, and the focus is on certifying bias. In [10], [24] an $\Omega(k^2)$ lower bound is proved for the number of coin tosses needed to find a biased coin with probability $1-e^{-\Omega(k)}$, whereas we present an $\tilde{O}(k)$ upper bound for the case of a constant fraction of biased coins.

E. Previous Work

Questions on the cost of derandomization are not new in theoretical computer science. In particular, the question of whether one can derandomize algorithms with little slowdown is related to Luby's question [23] of whether one can save in the number of processors when derandomizing parallel algorithms. Unlike Luby, we focus on sequential algorithms.

The connection that we make between derandomization and sketching adds to a long list of connections that have been identified over the years between derandomization, compression, learning and circuit lower bounds, e.g., circuit lower bounds can be used for pseudorandom generators and derandomization [19]; learning goes hand in hand with compression, and can be used to prove circuit lower bounds [12]; simplification under random restrictions can be used to prove circuit lower bounds [28] and construct pseudorandom generators [17]. Sparsification of the distinguisher of a pseudorandom generator (e.g., for simple distinguishers like DNFs) can lead to more efficient pseudorandom generators and derandomizations [15]. Our connection differs from all those connections. In particular, previous connections are based on pseudorandom generators, while our approach is dual and focuses on shrinking the number of inputs.

The idea of saving in a union bound by only considering representatives is an old idea with countless appearances in math and theoretical computer science, including derandomization (one example comes from the notion of an ε-net and its many uses; another example is [15] we mentioned above). Our contribution is in the formulation of an oblivious verifier and in designing sketches and oblivious verifiers.

Our applications have Atlantic City[5] algorithms that run in sub-linear time and have a constant error probability. There are works that aim to derandomize sub-linear time algorithms. Most notably, there is a deterministic version of the Frieze-Kannan regularity lemma [13], [9], [8], [4], which is relevant to some of our applications but not to others. Regularity lemmas do not apply to problems such as free games which we discuss in Section I-F.

Another work is [29] that generates deterministic *average case* algorithms for decision problems with certain sub-linear run time (Zimand's work incurred a slowdown that was subsequently removed by Shaltiel [27]). We focus on worst-case algorithms.

F. Application: Free Games

In the full version of the paper [16], we demonstrate our techniques with applications for MAX-CUT on dense graphs, (approximate) CLIQUE on graphs that contain large cliques, free games (constraint satisfaction problems on dense bipartite graphs), and reducing the Reed-Muller list decoding problem to its unique decoding problem. All our algorithms run in nearly linear time in their input size, and all of them beat the current state of the art algorithms in one aspect or another. The biggest improvement is in the algorithm for free games that is more efficient by orders of magnitude than the best deterministic algorithms.

A *free game* $\mathcal{G}$ is defined by a complete bipartite graph $G = (X, Y, X \times Y)$, a finite alphabet Σ and constraints $\pi_e \subseteq \Sigma \times \Sigma$ for all $e \in X \times Y$. For simplicity we assume $|X| = |Y|$. A labeling to the vertices is given by $f_X : X \to \Sigma$, $f_Y : Y \to \Sigma$. The value achieved by f_X, f_Y, denoted $val_{f_X,f_Y}(\mathcal{G})$, is the fraction of edges that are satisfied by f_X, f_Y, where an edge $e = (x, y) \in X \times Y$ is satisfied by f_X, f_Y if $(f_X(x), f_Y(y)) \in \pi_e$. The value of the instance, denoted $val(\mathcal{G})$, is the maximum over all labelings $f_X : X \to \Sigma$, $f_Y : Y \to \Sigma$, of $val_{f_X,f_Y}(\mathcal{G})$. Given a game $\mathcal{G}$ with value $val(\mathcal{G}) \geq 1-\varepsilon$, the task is to find a labeling to the vertices $g_X : X \to \Sigma$, $g_Y : Y \to \Sigma$, that satisfies at least $1 - O(\varepsilon)$ fraction of the edges.

Free games have been studied in the context of one round two prover games (see [11] and subsequent works on parallel repetition of free games) and two prover AM [1]. They unify a large family of problems on dense bipartite graphs obtained by considering different constraints. For instance, for MAX-2SAT we have $\Sigma = \{T, F\}$, and π_e contains all (a, b) that satisfy $\alpha \vee \beta$ where α is either a or $\neg a$ and β is either b or $\neg b$. Note that on a small fraction of the edges the constraints can be "always satisfied", so one can optimize over any dense graph, not just over the complete graph (the density of the graph is crucial: if fewer than $\varepsilon |X| |Y|$ of the edges have non-trivial constraints, then any labeling satisfies $1 - \varepsilon$ fraction of the edges).

There are randomized algorithms for free games that have constant error probability [5], [3], [6], [1], as well as a derandomization that incurs a polynomial

[5]Atlantic City algorithms have a two-sided error, as opposed to Monte Carlo algorithms that have a one-sided error, and Las Vegas algorithms that never err but may decline to output a solution.

slowdown [5]. In addition, deterministic algorithms for free games of value 1 are known. In the full version of the paper [16], we show a randomized algorithm with exponentially small error probability in $|X|\,|\Sigma|$ and a non-uniform deterministic algorithm whose running time is similar to that of the randomized algorithms with constant error probability.

Theorem I.1. *The following hold:*

1) *There is a Las Vegas algorithm that given a free game $\mathcal{G}$ with vertex sets X, Y, alphabet Σ, and $val(\mathcal{G}) \geq 1 - \varepsilon_0$, and given $\varepsilon > 0$, finds a labeling to the vertices that satisfies $1 - \varepsilon_0 - O(\varepsilon)$ fraction of the edges, except with probability exponentially small in $|X|\,|\Sigma|$. The algorithm runs in time $\tilde{O}(|X|\,|Y|\,|\Sigma|^{O((1/\varepsilon^2)\log(|\Sigma|/\varepsilon))})$.*
2) *There is a deterministic non-uniform algorithm that given a free game $\mathcal{G}$ with vertex sets X, Y, alphabet Σ, and $val(\mathcal{G}) \geq 1 - \varepsilon_0$, and given $\varepsilon > 0$, finds a labeling to the vertices that satisfies $1 - \varepsilon_0 - O(\varepsilon)$ fraction of the edges. The algorithm runs in time $\tilde{O}(|X|\,|Y|\,|\Sigma|^{O((1/\varepsilon^2)\log(|\Sigma|/\varepsilon))})$.*

At the high level, for a free games algorithm with constant error probability, we sample a subset $V' \subseteq X$ of the vertices of size roughly $\frac{1}{\varepsilon^2}\log(|\Sigma|/\varepsilon)$. Then, we use brute force to find the best assignment induced by V'. In the full version, we show how to amplify this algorithm, and then derandomize it using oblivious verifiers.

II. Derandomization by Oblivious Verification

In this section we develop a technique for converting randomized algorithms to deterministic non-uniform algorithms. The derandomization technique is based on the notion of "oblivious verifiers", which are verifiers that deterministically test the randomness of an algorithm while accessing only a sketch (compressed version) of the input to the algorithm. If the verifier accepts, the algorithm necessarily succeeds on the input and the randomness. In contrast, the verifier is allowed to reject randomness strings on which the randomized algorithm works correctly, as long as it does not do so for too many randomness strings.

Definition II.1 (Oblivious verifier)**.** Suppose that A is a randomized algorithm that on input $x \in \{0,1\}^N$ uses $p(N)$ random bits. Let $s : \mathbb{N} \to \mathbb{N}$ and $\varepsilon : \mathbb{N} \to [0,1]$. An (s, ε)-*oblivious verifier* for A is a deterministic procedure that gets as input N, a sketch $\hat{x} \in \{0,1\}^{s(N)}$ and $r \in \{0,1\}^{p(N)}$, either accepts or rejects, and satisfies the following:

- Every $x \in \{0,1\}^N$ has a sketch $\hat{x} \in \{0,1\}^{s(N)}$.
- For every $x \in \{0,1\}^N$ and its sketch $\hat{x} \in \{0,1\}^{s(N)}$, for every $r \in \{0,1\}^{p(N)}$, if the verifier accepts on input $\hat{x}$ and r, then A succeeds on x and r.
- For every $x \in \{0,1\}^N$ and its sketch $\hat{x} \in \{0,1\}^{s(N)}$, the probability over $r \in \{0,1\}^{p(N)}$ that the verifier rejects is at most $\varepsilon(N)$.

For example, the sketch for a free game on $G = (X \cup Y, E)$ consists of the restriction of the game to a small random subset of Y. In the full version, we show that such a sketch suffices to estimate the value of the labelings considered by our algorithm for free games.

Note that ε of the oblivious verifier may be somewhat larger than the error probability of the algorithm A, but hopefully not much larger. We do not limit the run-time of the verifier, but the verifier has to be deterministic. Indeed, the oblivious verifiers we design run in deterministic exponential time. We do not limit the time for computing the sketch $\hat{x}$ from the input x either. Indeed, we use the probabilistic method in the design of our sketches. Crucially, the sketch depends on the input x, but is independent of r.

Our derandomization theorem shows how to transform a randomized algorithm with an oblivious verifier into a deterministic (non-uniform) algorithm whose run-time is not much larger than the run-time of the randomized algorithm. Its idea is as follows. An oblivious verifier allows us to partition the inputs so inputs with the same sketch are bundled together, and the number of inputs effectively shrinks. This allows us to apply a union bound, just like in Adleman's proof [2], but over many fewer inputs, to argue that there must exist a randomness string for (a suitable repetition of) the randomized algorithm that works for all inputs.

Theorem II.2 (Derandomizing by verifying from a sketch)**.** *For every $t \geq 1$, if a problem has a Las Vegas algorithm that runs in time T and a corresponding (s, ε)-oblivious verifier for $\varepsilon < 2^{-s/t}$, then the problem has a non-uniform deterministic algorithm that runs in time $T \cdot t$ and always outputs the correct answer.*

Proof: Consider the randomized algorithm that runs the given randomized algorithm on its input for t times independently, and succeeds if any of the runs succeeds. Its run-time is $T \cdot t$. For any input, the probability that the oblivious verifier rejects all of the t runs is less than $(2^{-s/t})^t = 2^{-s}$. By a union bound over the 2^s possible sketches, the probability that the oblivious verifier rejects for any of the sketches is less than $2^s \cdot 2^{-s} = 1$. Hence, there exists a randomness string that the oblivious verifier accepts no matter what the sketch is. On this randomness string the algorithm has to be correct no matter what the input is. The deterministic non-uniform algorithm invokes the repeated randomized algorithm on this randomness string. ■

Adleman's theorem can be seen as a special case of Theorem II.2, in which the sketch size is the trivial $s(N) = N$, the oblivious verifier runs the algorithm on the input and randomness and accepts if the algorithm succeeds, and the randomized algorithm has error probability $\varepsilon < 2^{-N/t}$.

The reason that we require that the algorithm is a Las Vegas algorithm in Theorem II.2 is that it allows us to repeat the algorithm and combine the answers from all invocations. Combining is possible by other means as well. E.g., for randomized algorithms that solve decision problems or for pseudo-deterministic algorithms (algorithms that typically return the same answer) one can combine by taking majority. For algorithms that return a list, one can combine the lists.

The derandomization technique assumes that the error probability of the algorithm is sufficiently low. To complement it, in Section III we develop an amplification technique to decrease the error probability. Interestingly, our applications are such that the error probability can be decreased without a substantial slowdown to a point at which our derandomization technique kicks in, but we do not know how to decrease the error probability sufficiently for Adleman's original proof to work without slowing down the algorithm significantly.

III. Amplification by Finding a Biased Coin

In this section we develop a technique that will allow us to significantly decrease the error probability of randomized algorithms without substantially slowing down the algorithms. The technique works by testing the random choices made by the algorithm and quickly discarding undesirable choices. It requires the ability to quickly estimate the desirability of random choices. The technique is based on a solution to the following problem.

Definition III.1 (Biased coin problem)**.** Let $0 < \eta, \zeta < 1$. In the biased coin problem one has a source of coins. Each coin has a bias, which is the probability that the coin falls on "heads". The bias of a coin is unknown, and one can only toss coins and observe the outcome. It is known that at least $2/3$ fraction[6] of the coins have bias at least $1 - \eta$. Given $n \geq 1$, the task is to find a coin of bias at least $1 - \eta - \zeta$ with probability at least $1 - \exp(-n)$ using as few coin tosses as possible.

[6] $2/3$ can be replaced with any constant larger than 0.

A similar problem was studied in the setup of multi-armed bandit problems [10], [24], however in that setup there might be only one coin with large bias, as opposed to a constant fraction of coins as in our setup. In the former setup, many more coin tosses might be needed (an $\Omega(n^2/\zeta^2)$ lower bound is proved in [24]).

A. Biased Coin and Amplification

The analogy between the biased coin problem and amplification is as follows: a coin corresponds to a random choice of the algorithm. Its bias corresponds to how desirable the random choice is. The assumption is that a constant fraction of the random choices are very desirable. The task is to find one desirable random choice with a very high probability. Tossing a coin corresponds to testing the random choice. The coin falls on heads in proportion to the quality of the random choice.

More formally, we will be able to amplify with little slowdown randomized algorithms that have a quick check as defined next:

Definition III.2 (Randomness checker)**.** Let Ω be a space of randomness strings. Let $grade : \Omega \to [0, 1]$ assign each randomness string a grade. A *randomness checker* is a randomized algorithm that given $r \in \Omega$ accepts with probability $grade(r)$.

Definition III.3 (Algorithm with quick check)**.** Let $t_{check} : \mathbb{N} \to \mathbb{N}$ and $0 < \eta, \zeta < 1$. We say that a randomized algorithm A has a (t_{check}, η, ζ)-quick check if for every input x, $|x| = n$, to the algorithm there is a function $grade_x : \Omega_n \to [0, 1]$ with a randomness checker, where Ω_n is the space of randomness strings on input size n.

- Randomization: For at least $2/3$ fraction of $r \in \Omega_n$ we have $grade_x(r) \geq 1 - \eta$.
- Approximation: If $grade_x(r) \geq 1 - \eta - \zeta$ then A is correct on x using randomness r.
- Quickness: The run-time of the checker is bounded by $t_{check}(n)$.

In the full version we extend the above definitions. The general definitions apply even if each randomness has several possible grades, and even if the randomness checker accepts only with probability that *approximates* the grade(s) of the randomness.

For example, consider an algorithm to approximate an instance of free games on $G = (X \cup Y, E)$ and an alphabet Σ to within ε. To achieve constant error probability, the algorithm samples $V' \subseteq X$ of size roughly $\frac{1}{\varepsilon^2} \log(|\Sigma|/\varepsilon)$. Then, for every possible assignment to V', the algorithm computes an *induced assignment* for the rest of the vertices: for each $y \in Y$, assign $\sigma \in \Sigma$ that maximizes the fraction of satisfied edges between y and V'. For each $x \in X$, assign $\sigma \in \Sigma$ that maximizes the fraction of satisfied edges between x and Y.

The algorithm uses randomness to pick V', and for every assignment to V' the randomness gets a different grade. The grade is the fraction of satisfied edges for the induced assignment. The randomness checker approximates the grade by first finding an assignment to Y, given the assignment chosen for V'. Then, it picks an $x \in X$ (in the full version, we actually pick a larger subset $X' \subseteq X$), and finds the best assignment for x. The checker returns the fraction of satisfied constraints between x and Y. Note that the run-time of the randomness checker is $O(|V'||Y|)$, which is significantly less than the run-time of the algorithm of constant error probability.

Suppose that the desired error probability for the amplified algorithm is $\exp(-k)$. Given an input we will show how to find a randomness string to plug into the basic algorithm in time roughly $k \cdot t_{check}$, as opposed to $k \cdot t$ where t is the run-time of A.

Lemma III.4 (Amplification via biased coin)**.** *For any $k \geq 1$, if A is a randomized algorithm with a (t_{check}, η, ζ)-quick check and that runs in time t for some problem, then there is a randomized algorithm A' for the same problem whose run-time is $t + \tilde{O}(kt_{check}/\zeta^2)$ and whose error probability is* $\exp(-k)$.

The lemma follows from Lemma III.5 that we prove in the sequel by using the quick check to "toss" the coin associated with the randomness string. The lemma does not imply anything new for the cut algorithm we mentioned in the example above, since its error probability was already exponentially small in the number of vertices. However, the lemma is useful for many other algorithms. In applications, we often don't have pure quick checks, but instead have algorithms which may simulate or approximate quick checks. A simulator is given a number k and its task is to simulate k applications of a randomness checker. Sometimes there is a bound K, such that only $k \leq K$ is allowed (e.g., the simulator picks a sample of the vertices, and cannot sample more than all the vertices).

B. The Gapped Case

Interestingly, if we knew that all coins have bias either at least $1-\eta$ or at most $1-\eta-\zeta$, it would be possible to solve the biased coin problem using only $O(n/\zeta^2)$ coin tosses. The algorithm is described in Figure 1. It tosses a random coin a small number of times and expects to witness about $1-\eta$ fraction heads. If so, it doubles the number of tosses, and tries again, until its confidence in the bias is sufficiently large. If the fraction of heads is too small, it restarts with a new coin. The algorithm has two parameters: i_0 that determines the initial number of tosses, and i_f that determines the final number of tosses.

The probability that the algorithm restarts at the i'th phase is exponentially small in $\zeta^2 k$ for $k = 2^i$: either the coin had bias at least $1-\eta$, and then there's an exponentially small probability in $\zeta^2 k$ that there were less than $(1-\eta-\zeta/2)k$ heads, or the coin had bias at most $1-\eta-\zeta$, and then there is probability exponentially small in $\zeta^2 k$ that the coin had at least $1-\eta-\zeta/2$ fraction heads in all the previous phases (whereas if this is phase $i = i_0$, then the probability that a coin with bias less than $1-\eta$ was picked in this case is constant, i.e., exponentially small in $\zeta^2 k$). Moreover, the number of coin tosses up to this step is at most $2k$. Hence, we maintain a linear relation (up to ζ^2 factor) between the number of coin tosses and the exponent of the probability. To get the error probability down to $\exp(-n)$ we only need $O(n/\zeta^2)$ coin tosses.

```
FIND-BIASED-COIN-GIVEN-GAP($n, \eta, \zeta$)
1  Set $i_0 = \log(1/\zeta^2) + \Theta(1)$;
   $i_f = \log(n/\zeta^2) + \Theta(1)$
   (constants picked appropriately).
2  Pick a coin at random.
3  for $i = i_0, i_0+1, \ldots, i_f$
4      Toss the coin for $k = 2^i$ times.
5      If the fraction of heads is less than
        $1-\eta-\zeta/2$, restart.
6  return coin.
```

Figure 1: An algorithm for finding a coin of bias at least $1-\eta-\zeta$ when all the coins either have bias at least $1-\eta$ or at most $1-\eta-\zeta$. The algorithm uses $O(n/\zeta^2)$ coin tosses and achieves error probability $\exp(-n)$.

C. The General Case

Counter-intuitively, adding coins of bias between $1-\eta-\zeta$ and $1-\eta$ – all acceptable outcomes of the algorithm – derails the algorithm we outlined above, as well as other algorithms. If one fixes a threshold like $1-\eta-\zeta/2$ for the fraction of heads one expects to witness, there might be a coin whose bias is close to the threshold. We might toss this coin a lot and then decide to restart with a new coin. One can also consider a competition-style algorithm like the ones studied in [10], [24] when one tries several coins each time, keeping the ones that fall on heads most often.

Such an algorithm may require $\Omega(n^2/\zeta^2)$ coin tosses, since coins can lose any short competition to coins with slightly smaller bias; then, such coins can lose to coins with slightly smaller bias, and so on, until we may end up with a coin of bias smaller than $1-\eta-\zeta$.

There is, however, an algorithm that uses only $\tilde{O}(n/\zeta^2)$ coin tosses. This algorithm decreases the threshold for the fraction of heads one expects to witness with respect to the number of coin tosses one already made for this coin. If the coin was already tossed a lot, the deviation of the number of heads from $1-\eta$ would have to be large for us to decide to restart with a new coin. The algorithm is described in Figure 2.

```
FIND-BIASED-COIN($n, \eta, \zeta$)
1  Set $i_0 = \log(1/\zeta^2) + \Theta(1)$;
   $i_f = \log(n/\zeta^2) + \Theta(\log\log(n/\zeta))$;
   $\beta = \zeta/i_f$.
2  Pick a coin at random.
3  for $i = i_0, i_0+1, \ldots, i_f$
4      Toss the coin for $k = 2^i$ times.
5      If the fraction of heads is less than
         $1-\eta-i\beta$, restart.
6  return coin.
```

Figure 2: An algorithm for finding a coin of bias at least $1-\eta-\zeta$ using $\tilde{O}(n/\zeta^2)$ coin tosses. The error probability is exponentially small in n.

Note that the deviation parameter β is picked so $1-\eta-i\beta \geq 1-\eta-\zeta$ for all $i \leq i_f$.

Lemma III.5. *Within $O((n/\zeta^2)\log^2(n/\zeta)) = \tilde{O}(n/\zeta^2)$ coin tosses,* FIND-BIASED-COIN *outputs a coin of bias at least $1-\eta-\zeta$ except with probability* $\exp(-n)$.

Proof: Suppose that the algorithm restarts at phase i. The number of coin tosses made by this point since the previous restart (if any) is $2^{i_0} + 2^{i_0+1} + \ldots + 2^i \leq 2^{i+1}$. Moreover, if the coin had bias smaller than $1-\eta-i\beta+\beta/2$, then, if $i > i_0$, by a Chernoff bound, the probability the coin passed the previous test, where it was supposed to have at least $1-\eta-(i-1)\beta$ fraction of heads, is at most $\exp(-\beta^2 2^{i-2})$. If $i = i_0$, there is probability less than $1/3$ that the coin was picked. If the coin had bias at least $1-\eta-i\beta+\beta/2$, then by the Chernoff bound, the probability it failed the current test, where it is supposed to have at least $1-\eta-i\beta$ fraction of heads, is at most $\exp(-\beta^2 2^{i-1})$. In any case, the ratio between the number of coin tosses and the exponent of the probability is $O(1/\beta^2)$. The value of i_f is chosen so that the error probability in the last iteration is $\exp(-n)$. By the choice of β, the coin tosses to exponent ratio is $O((1/\zeta^2)\log^2(n/\zeta))$. Therefore, the number of coin tosses one needs in order to reach error probability $\exp(-n)$ is $O((n/\zeta^2)\log^2(n/\zeta))$. ■

IV. Open Problems

- We believe that the question of what slowdown is incurred by deterministic vs. randomized algorithms deserves a great deal of attention from the research community.
- We obtained efficient non-uniform deterministic algorithms. It would be very interesting to convert them to uniform algorithms.
- What other algorithms can be derandomized using our method? Can more sophisticated sketching and sparsification techniques be used to handle algorithms on sparse graphs?
- What lower bound can one prove on the number of coin tosses needed to find a biased coin? What if the target bias is not known, yet it is known that a large fraction of the coins achieve that target? Solving the latter would yield an algorithm for FREE GAMES that handles games with general value, rather than value close to 1.
- Can one use the existence of an oblivious verifier (i.e., effectively fewer inputs to consider) to construct better psuedorandom generators?
- The run-times of our algorithms have $\text{poly}\log n$ factors coming from our algorithm for the biased coin problem and from the size of the sketches. Can they be eliminated?
- The minimum spanning tree (MST) problem has a randomized linear time algorithm achieving error probability exponentially small in the number of edges m [21]. Also, a result of Pettie and Ramachandran proves that a non-uniform linear time algorithm for MST would imply a uniform algorithm [26]. Therefore, an $(O(m), 2^{-\Omega(m)})$-oblivious verifier for such a minimum spanning tree algorithm would imply a linear time deterministic algorithm for MST. Finding such an oblivious verifier remains open.

Acknowledgements

Dana Moshkovitz is grateful to Sarah Eisenstat for her collaboration during the long preliminary stages of this work.

References

[1] S. Aaronson, R. Impagliazzo, and D. Moshkovitz. AM with multiple Merlins. In *2014 IEEE 29th Conference on Computational Complexity (CCC)*, pages 44–55, 2014.

[2] L. Adleman. Two theorems on random polynomial time. In *Proc. 19th IEEE Symp. on Foundations of Computer Science*, pages 75–83, 1978.

[3] N. Alon, W. F. de la Vega, R. Kannan, and M. Karpinski. Random sampling and approximation of MAX-CSPs. *Journal of Computer and System Sciences*, 67(2):212–243, 2003.

[4] N. Alon, R.A. Duke, H. Lefmann, V. Rödl, and R. Yuster. The algorithmic aspects of the regularity lemma. *Journal of Algorithms*, 16(1):80 – 109, 1994.

[5] S. Arora, D. Karger, and M. Karpinski. Polynomial time approximation schemes for dense instances of NP-hard problems. In *Proc. 27th ACM Symp. on Theory of Computing*, pages 284–293, 1995.

[6] B. Barak, M. Hardt, T. Holenstein, and D. Steurer. Subsampling mathematical relaxations and average-case complexity. In *Proc. 22nd Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 512–531, 2011.

[7] I. Baran, E. Demaine, and M. Patrascu. Subquadratic algorithms for 3SUM. In *Algorithms and Data Structures*, volume 3608 of *Lecture Notes in Computer Science*, pages 409–421. 2005.

[8] D. Dellamonica, S. Kalyanasundaram, D. Martin, V. Rödl, and A. Shapira. A deterministic algorithm for the Frieze-Kannan regularity lemma. *SIAM Journal on Discrete Math*, 26:15–29, 2012.

[9] D. Dellamonica, S. Kalyanasundaram, D. Martin, V. Rödl, and A. Shapira. An optimal algorithm for finding Frieze-Kannan regular partitions. *Combinatorics, Probability and Computing*, 24(2):407–437, 2015.

[10] E. Even-Dar, S. Mannor, and Y. Mansour. PAC bounds for multi-armed bandit and Markov decision processes. In *Computational Learning Theory*, volume 2375 of *Lecture Notes in Computer Science*, pages 255–270. 2002.

[11] U. Feige. Error reduction by parallel repetition - the state of the art, 1995.

[12] L. Fortnow and A. R. Klivans. Efficient learning algorithms yield circuit lower bounds. *Journal of Computer and System Sciences*, 75(1):27 – 36, 2009.

[13] A. Frieze and R. Kannan. The regularity lemma and approximation schemes for dense problems. In *Proc. 37th IEEE Symp. on Foundations of Computer Science*, pages 12–20, 1996.

[14] O. Goldreich. A sample of samplers - a computational perspective on sampling (survey). Technical report, ECCC Report TR97-020, 1997.

[15] P. Gopalan, R. Meka, and O. Reingold. DNF sparsification and a faster deterministic counting algorithm. *Computational Complexity*, 22(2):275–310, 2013.

[16] O. Grossman and D. Moshkovitz. Amplification and derandomization without slowdown. Technical report, ECCC Report TR15-158, 2015.

[17] R. Impagliazzo, R. Meka, and D. Zuckerman. Pseudorandomness from shrinkage. In *Proc. 53rd IEEE Symp. on Foundations of Computer Science*, pages 111–119, 2012.

[18] R. Impagliazzo, S. Shaltiel, and A. Wigderson. Reducing the seed length in the nisan-wigderson generator. *Combinatorica*, 26(6):647–681, 2006.

[19] R. Impagliazzo and A. Wigderson. P = BPP if E requires exponential circuits: Derandomizing the XOR lemma. In *Proc. 29th ACM Symp. on Theory of Computing*, pages 220–229, 1997.

[20] R. Impagliazzo and D. Zuckerman. How to recycle random bits. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pages 248–253, 1989.

[21] D. Karger, P. Klein, and R. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42:321–328, 1995.

[22] H. Karloff and Y. Mansour. On construction of k-wise independent random variables. *Combinatorica*, 17(1):91–107, 1997.

[23] M. Luby. Removing randomness in parallel computation without a processor penalty. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pages 162–173, 1988.

[24] S. Mannor and J. N. Tsitsiklis. The sample complexity of exploration in the multi-armed bandit problem. *J. Mach. Learn. Res.*, 5:623–648, 2004.

[25] J. Naor and M. Naor. Small-bias probability spaces: Efficient constructions and applications. *SIAM Journal on Computing*, 22(4):838–856, 1993.

[26] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *Journal of the ACM*, 49(1):16–34, 2002.

[27] R. Shaltiel. Weak derandomization of weak algorithms: Explicit versions of yao's lemma. *computational complexity*, 20(1):87–143, 2011.

[28] B. A. Subbotovskaya. Realizations of linear functions by formulas using +, *, -,. *Sov. Math. Dokl.*, 2:110–112, 1961.

[29] M. Zimand. Exposure-resilient extractors and the derandomization of probabilistic sublinear time. *computational complexity*, 17(2):220–253, 2008.