Amplification and Derandomization Without Slowdown*

Ofer Grossman† Dana Moshkovitz‡

July 20, 2020

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 similar run-time to the original randomized algorithms.

The amplification technique applies when there is a quick, probabilistic, test of the randomness. In this case, we show how to efficiently find *one* randomness string for the algorithm that works with very high probability, and apply the algorithm only on that randomness (In contrast, standard approaches suggest a number of possible randomness strings, and run the algorithm on all of them). The search of good randomness turns out to be a natural stochastic multi-armed bandit problem, which we define ("the biased coin problem") and analyze.

The derandomization technique applies when there is a verifier that can test the randomness to the algorithm while only inspecting a sub-linear size sketch of the input (the sketch may be hard to compute; the verifier may be inefficient and is allowed reject a small portion of the good randomness strings). In this case, we show how to apply Adleman's derandomization (from the proof of $BPP \subseteq P/poly$) more efficiently.

We demonstrate the techniques by showing applications for dense max-cut, approximate clique, free games, and going from list decoding to unique decoding for Reed-Muller codes.

*Extended Abstract appeared in the proceedings of FOCS 2016.

†ogrossma@mit.edu. Department of Mathematics, MIT.

‡danama@cs.utexas.edu. Department of Computer Science, UT Austin. This material is based upon work done at MIT and supported by the National Science Foundation under grants number 1218547 and 1452302.

1 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 [26], yielding a substantial slowdown.

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 [28], but the number of invocations – and hence the run-time – provably remains large (see, e.g., [21]).

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 1.3). 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 1.1 and we discuss the importance of non-uniform algorithms in Section 1.2.

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. 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.

1.1 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

[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.

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 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 branches of the algorithm. We bound the number of branches, trimming low probability branches. We then use the exponentially low error probability of our randomized algorithms, implying that with high probability *all* branches work correctly, so there is no importance to the exact branch that the algorithm picked. 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.

1.2 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.

Interestingly, for certain algorithmic problems the uniform and the non-uniform model are equivalent. One important example is the problem of finding an optimal deterministic minimum spanning tree algorithm where equivalence was shown in [39]. Indeed, this was the original motivation for our work. In fact, minimum spanning tree is part of a much wider phenomenon: 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 [9].

Furthermore, our non-uniform algorithms: (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). Hence, one can get rid of non-uniformity altogether by preprocessing, amortizing, or using non-determinism:

[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.

- *Amortization:* 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 of the search 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:* One can find a correct advice during a preprocessing step.
- *Non-determinism:* 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.

1.3 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., [33]). 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. 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 algorithm cannot perform tests on the randomness that require more time than its run-time. Impagliazzo and Wigderson [27] capitalize on that to show how to construct – based on hard functions that plausibly exist – pseudorandom generators that "fool" any randomized algorithm that runs in a fixed polynomial time. One can use other limitations on the algorithm to construct unconditional pseudorandom generators. For instance, some algorithms only require that k-tuples of random bits are independent, and k-wise independent generators fool them [29]. Some algorithms only perform linear operations on their random bits, and ϵ-biased generators fool them [38].

The disadvantage of pseudorandom generators is that one needs to go through all their seeds to derandomize the algorithm, and this incurs a slowdown. Alternatively, one can parallelize the computation for different seeds, and this incurs an increase in the number of processors [33]. In general the slowdown (or increase in the number of processors) is likely at least linear in the number of random bits that the algorithm uses [26]. For k-wise independent generators the slowdown is at least $\Omega(n^k)$ [31]. For ϵ-biased generators the slowdown is at least linear [6]. The slowdown is at least linear in almost all known constructions of pseudorandom generators, including pseudorandom generators for polynomials [45], polynomial thresholds functions [37], regular branching programs [10], and most others. One case when only a poly-logarithmic slowdown can be achieved is almost k-wise independent generators for small k [38]. For those generators the slowdown is only

[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 1.3.

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 pseudorandom generators, one shrinks the space of possible randomness strings. In derandomization from sketching one shrinks the space of possible inputs. Pseudorandom generators 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 pseudorandom generators 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.

1.4 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.

For a randomized algorithm, certain randomness strings (those which lead to incorrect outputs) are "bad" randomness stings. For example, consider the following algorithm for finding a cut where at least $|E|/3$ of the edges cross the cut: pick a random cut, and check if more than $|E|/3$ of the edges are cut by it (it is not too hard to derandomize this algorithm using standard techniques, but we briefly consider it here anyway for the sake of a very simple concrete example. Note that later in the paper we consider a *different* and more involved max-cut algorithm, one that provides better approximation for dense graphs, and show how to apply our techniques there). Since each edge is cut with probability $\frac{1}{2}$, the expected cut size is $|E|/2$, and one can show that with enough repetitions with high probability a good cut will be found. First, note that in this algorithm, a randomness is "bad" if the cut corresponding to the randomness contains fewer than $|E|/3$ edges. We can assign a "grade" to a randomness r corresponding to the fraction of edges that cross the cut defined by r. So, randomness strings with a low grade are "bad", and the algorithm will not succeed when using them.

Naively, in order to amplify the above algorithm (that is, decrease its error probability), one would have to run the algorithm multiple times. However, that may be wasteful: if the randomness chosen is bad, then we are wasting a lot of computation time to check the size of the randomness's cut. Our amplification idea is essentially a way to get around this. Instead of running the algorithm with the randomness, we run some quick tests on the randomness to estimate its grade. We term such a test of the randomness's grade a "quick check". If the grade of the randomness is low (and hence we have reason to suspect the chosen randomness is "bad"), then we discard the randomness and restart. In this max-cut approximation example, a randomness checker can simply involve sampling few (say, 100) edges, and checking if they cross the cut (and, say, outputting 1 if more than 33 edges are in the cut, and otherwise outputting 0). If fewer than 33 of our sampled edges are cut, we have reason to believe our randomness is bad, so we discard it and restart. If many of the edges are cut, we may choose to try using the randomness (or, check it again to increase our confidence in the randomness).

It's not clear how many times one should run a quick check on a random string before using it or discarding it. One of our contributions is providing an algorithm that shows how to use quick checks to amplify the success probability of an algorithm (this is the algorithm for the "biased coin problem").

More formally, 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 [14, 34], 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 [14, 34] one has to toss all coins, and 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 [14, 34] 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.

1.5 Previous Work

There are many works that are related to certain aspects of the current paper.

[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.

Questions on the cost of low error probability are of course not new, and appear in many guises in theoretical computer science. In particular, the question of whether one can derandomize algorithms with little slowdown is related to Luby's question [33] on whether one can save in the number of processors when converting a randomized algorithm to a deterministic parallel algorithm. Unlike Luby, who considered parallel algorithms, we focus on standard, 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 [27]; learning goes hand in hand with compression, and can be used to prove circuit lower bounds [16]; simplification under random restrictions can be used to prove circuit lower bounds [41] and construct pseudorandom generators [25]. Sparsification of the distinguisher of a pseudorandom generator (e.g., for simple distinguishers like DNFs) can lead to more efficient pseudorandom generators and derandomizations [23]. 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 [23] 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 [44, 19, 13, 12, 5, 17, 18], which is relevant to some of our applications but not to others (more on that when we discuss the individual applications in Section 1.6). Another work is [46] 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 [40]). We focus on worst-case algorithms.

1.6 Applications

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. The algorithm for Max-Cut can efficiently handle sparser graphs than the best deterministic algorithm, the algorithm for (approximate) Clique can efficiently handle smaller cliques than the best deterministic algorithm; and the algorithm for the Reed-Muller code achieves similar run-time as sophisticated algebraic algorithms despite being much simpler. In general, our focus is on demonstrating the utility and versatility of the techniques and not on obtaining the most efficient algorithm for each problem. In the open problems section we point to several aspects where we leave room for improvement.

[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.

1.6.1 Max Cut on Dense Graphs

Given a graph $G = (V, E)$, a cut in the graph is defined by $C \subseteq V$. The value of the cut is the fraction of edges $e = (u, v) \in E$ such that $u \in C$ and $v \in V - C$. We say that a graph is γ-dense if it contains $\gamma |V|^2 /2$ edges. For simplicity we assume that the graph is regular, so every vertex has degree $\gamma |V|$. Given a regular γ-dense graph that has a cut of value at least $1 - \varepsilon$ for $\varepsilon < 1/4$, we'd like to find a cut of value roughly $1 - \varepsilon$. Understanding this problem on general (non-dense) graphs is an important open problem: (a weak version of) the Unique Games Conjecture [32]. However, for dense graphs, it is possible to construct a cut of value $1 - \varepsilon - \zeta$ efficiently [11, 7, 22, 36]. The best randomized algorithms are an algorithm of Mathieu and Schudy [36] that runs in time $O(|V|^2 + 2^{O(1/\gamma^2\zeta^2)})$ and an algorithm of Goldreich, Goldwasser and Ron [22] that runs in time $O(|V| (1/\gamma\zeta)^{O(1/\gamma^2\zeta^2)} + (1/\gamma\zeta)^{O(1/\gamma^3\zeta^3)})$ (Note that the algorithm of [22] runs in sub-linear time. This is possible since it is an Atlantic City algorithm). Both algorithms have constant error probability. We obtain a Las Vegas algorithm with exponentially small error probability, and deduce a deterministic non-uniform algorithm. This is the simplest application of our techniques. It uses the biased coin algorithm, but does not require any sketches.

Theorem 1.1. *There is a Las Vegas algorithm that given a γ-dense graph G that has a cut of value at least $1 - \varepsilon$ for $\varepsilon < 1/4$, and given $\zeta < 1/4 - \varepsilon$, finds a cut of value at least $1 - \varepsilon - O(\zeta)$, except with probability exponentially small in $|V|^2$. The algorithm runs in time $\tilde{O}(|V|^2 (1/\zeta)^{O(1/\gamma^2+1/\zeta^2)})$. It also implies a non-uniform deterministic algorithm with the same run-time.*

Note that run-time $\Omega(\gamma |V|^2)$ is necessary for a deterministic algorithm, since the input size is $\gamma |V|^2$. A deterministic $O(|V|^2 2^{\mathrm{poly}(1/\gamma,1/\zeta)})$-time algorithm follows from a recent deterministic version of the Frieze-Kannan regularity lemma [44, 19, 13, 12, 5, 17, 18] (At the time our work was originally published, the best dependence on $1/\gamma$ and $1/\zeta$ was *doubly* exponential).

1.6.2 Approximate Clique

The input is $0 < \varepsilon, \rho < 1$ and an undirected graph $G = (V, E)$ for which there exists a set $C \subseteq V$, $|C| \geq \rho |V|$, that spans a clique. The goal is to find a set $D \subseteq V$, $|D| \geq \rho |V|$, whose edge density is at least $1-\varepsilon$, i.e., if $E(D) \subseteq E$ is the set of edges whose endpoints are in D, then $|E(D)| / \binom{|D|}{2} \geq 1-\varepsilon$. Goldreich, Goldwasser and Ron [22] gave a randomized $O(|V| (1/\varepsilon)^{O(1/(\rho^3\varepsilon^2))})$ time algorithm for this problem with constant error probability (Note that this is a sub-linear time algorithm. This is possible since it is an Atlantic City algorithm). A deterministic $O(|V|^2 2^{\mathrm{poly}(1/\rho,1/\varepsilon)})$ time algorithm follows from a deterministic version of the Frieze-Kannan regularity lemma [44, 19, 13, 12, 5, 17, 18] (the last of of these papers appeared after this work was initially published, and was in fact motivated by this work. At the time our work was published, the best dependence on $1/\rho$ and $1/\varepsilon$ was *doubly* exponential). We obtain a randomized algorithm with exponentially small error probability in $|V|$, and use sketching to obtain a non-uniform deterministic algorithm. Our algorithms have better dependence in ρ and ε than the existing deterministic algorithm, and can therefore handle efficiently graphs with smaller cliques than the existing deterministic algorithm and output denser sets.

Theorem 1.2. *The following hold:*

1. *There is a Las Vegas algorithm that given $0 < \rho, \varepsilon < 1$, and a graph $G = (V, E)$ with a clique on $\rho |V|$ vertices, finds a set of $\rho |V|$ vertices and density at least $1 - \varepsilon$, except with probability exponentially small in $|V|$. The algorithm runs in time $\tilde{O}(|V|^2 2^{O(1/(\rho^3\varepsilon^2))})$.*

2. *There is a deterministic non-uniform algorithm that given $0 < \rho, \varepsilon < 1$, and a graph $G = (V, E)$ with a clique on $\rho|V|$ vertices, finds a set of $\rho|V|$ vertices and density at least $1 - \varepsilon$. The algorithm runs in time $\tilde{O}(|V|^2 2^{O(1/(\rho^3\varepsilon^2))})$.*

The sketch for approximate clique consists of all the edges that touch a small random set of vertices. We show that such a sketch suffices to estimate the density of the sets considered by the algorithm and the quality of the random samples of the algorithm.

1.6.3 Free Games

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 researched in the context of one round two prover games (see [15] and many 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 [7, 4, 8, 1], as well as a derandomization that incurs a polynomial slowdown [7]. In addition, deterministic algorithms for free games of value 1 are known [35]. 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 1.3. *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 $\tilde{O}(|X||Y||\Sigma|^{O((1/\varepsilon^2)\log(|\Sigma|/\varepsilon))})$.*

The sketch of a free game consists of the restriction of the game to a small random subset of Y. We show that the sketch suffices to estimate the value of the labelings considered by the algorithm and the random samples the algorithm makes.

1.6.4 From List Decoding to Unique Decoding of Reed-Muller Code

Definition 1.4 (Reed-Muller code)**.** The Reed-Muller code defined by a finite field $\mathbb{F}$ and natural numbers m and d consists of all m-variate polynomials of degree at most d over $\mathbb{F}$.

Let $0 < \epsilon < \rho < 1$. In the list decoding to unique decoding problem for the Reed-Muller code, the input is a function $f : \mathbb{F}^m \to \mathbb{F}$ and the goal is to output a list of $l = O(1/\epsilon)$ functions $g_1, \ldots, g_l : \mathbb{F}^m \to \mathbb{F}$, such that for every m-variate polynomial p of degree at most d over $\mathbb{F}$ that agrees with f on at least ρ fraction of the points $x \in \mathbb{F}^m$, there exists g_i that agrees with p on at least $1 - \epsilon$ fraction of the points $x \in \mathbb{F}^m$.

There are randomized, self-correction-based, algorithms for this problem (see [43] and the references there). There are also deterministic list decoding algorithms for the Reed-Solomon and Reed-Muller codes that can solve this problem: The algorithms of Sudan [42] and Guruswami-Sudan [24] run in large polynomial time, as they involve solving a system of linear equations and factorization of polynomials. There are efficient implementations of these algorithms that run in time $\tilde{O}(|\mathbb{F}^m|)$ (see [3] and the references there), but they involve deeper algebraic technique. Our contribution is simple, combinatorial, algorithms, randomized and deterministic, with nearly-linear run-time. This application too relies on the biased coin algorithm but does not require sketching.

Theorem 1.5. *Let $\mathbb{F}$ be a finite field, let d and $m > 3$ be natural numbers and let $0 < \rho, \epsilon < 1$, such that $d \leq |\mathbb{F}|/10$, $\epsilon > \sqrt[3]{2/|\mathbb{F}|}$ and $\rho > \epsilon + 2\sqrt{d/|\mathbb{F}|}$. There is a randomized algorithm that given $f : \mathbb{F}^m \to \mathbb{F}$, finds a list of $l = O(1/\rho)$ functions $g_1, \ldots, g_l : \mathbb{F}^m \to \mathbb{F}$, such that for every m-variate polynomial p of degree at most d over $\mathbb{F}$ that agrees with f on at least ρ fraction of the points $x \in \mathbb{F}^m$, there exists g_i that agrees with p on at least $1 - \epsilon$ fraction of the points $x \in \mathbb{F}^m$. The algorithm has error probability exponentially small in $|\mathbb{F}^m| \log |\mathbb{F}|$ and it runs in time $\tilde{O}(|\mathbb{F}^m| \operatorname{poly}(|\mathbb{F}|))$. It implies a deterministic non-uniform algorithm with the same run-time.*

Note that the standard choice of parameters for the Reed-Muller code has $|\mathbb{F}| = \operatorname{poly}\log |\mathbb{F}^m|$, and in this case our algorithms run in nearly linear time $\tilde{O}(|\mathbb{F}^m|)$.

2 Preliminaries

2.1 Conventions and Inequalities

Lemma 2.1 (Chernoff bounds)**.** *Let $X_1, \ldots, X_n$ be i.i.d random variables taking values in $\{0, 1\}$. Let $1 > \varepsilon > 0$. Then,*

$$\Pr\left[\frac{1}{n}\sum X_i \geq \frac{1}{n}\sum \mathbf{E}\left[X_i\right] + \varepsilon\right] \leq e^{-2\varepsilon^2 n}, \ \ \Pr\left[\frac{1}{n}\sum X_i \leq \frac{1}{n}\sum \mathbf{E}\left[X_i\right] - \varepsilon\right] \leq e^{-2\varepsilon^2 n}.$$

The same inequalities hold for random variables taking values in $[0, 1]$ (Hoeffding bound). The multplicative version of the Chernoff bound states:

$$\Pr\left[\sum X_i \geq (1+\varepsilon) \cdot \sum \mathbf{E}\left[X_i\right]\right] \leq e^{-\varepsilon^2 \sum \mathbf{E}[X_i]/3}, \ \ \Pr\left[\sum X_i \leq (1-\varepsilon) \cdot \sum \mathbf{E}\left[X_i\right]\right] \leq e^{-\varepsilon^2 \sum \mathbf{E}[X_i]/2}.$$

When we say that a quantity is *exponentially small in k* we mean that it is of the form $2^{-\Omega(k)}$. We use $\exp(-n)$ to mean e^{-n}.

2.2 Non-Uniform and Randomized Algorithms

Definition 2.2 (Non-uniform algorithm)**.** A non-uniform algorithm that runs in time $t(n)$ is given by a sequence $\{C_n\}$ of Boolean circuits, where for every $n \geq 1$, the circuit C_n gets an input of size n and satisfies $|C_n| \leq t(n)$.

Alternatively, a non-uniform algorithm that runs in time $t(n)$ on input of size n is given an advice string $a = a(n)$ of size at most $t(n)$ (note that $a(n)$ depends on n but not on the input!). The algorithm runs in time $t(n)$ given the input and the advice.

The class of all languages that have non-uniform polynomial time algorithms is called P/ poly.

There are two main types of randomized algorithms: the strongest are Las Vegas algorithms that may not return a correct output with small probability, but would report their failure. Atlantic City algorithms simply return an incorrect output a small fraction of the time.

Definition 2.3 (Las Vegas algorithm)**.** A *Las Vegas* algorithm that runs in time $t(n)$ on input of size n is a randomized algorithm that always runs in time at most $t(n)$, but may, with a small probability return $\perp$. In any other case, the algorithm returns a correct output.

The probability that a Las Vegas algorithm returns $\perp$ is called its *error probability.* In any other case we say that the algorithm succeeds.

Definition 2.4 (Atlantic City algorithm)**.** An *Atlantic City* algorithm that runs in time $t(n)$ on input of size n is a randomized algorithm that always runs in time at most $t(n)$, but may, with a small probability, return an incorrect output.

The probability that an Atlantic City algorithm returns an incorrect output is called its *error probability.* In any other case we say that the algorithm succeeds.

Note that a Las Vegas algorithm is a special case of Atlantic City algorithms. Atlantic City algorithms that solve decision problems return the same output the majority of the time. For search problems we have the following notion:

Definition 2.5 (Pseudo-deterministic algorithm, [20])**.** A *Pseudo-deterministic* algorithm is an Atlantic City algorithm that returns the same output except with a small probability, called its *error probability.*

3 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 3.1 (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 $\varepsilon(N)$.

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 3.2 (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 3.2, 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 3.2 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 4 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.

4 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 4.1 (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.

A similar problem was studied in the setup of multi-armed bandit problems [14, 34], 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 [34]).

4.1 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 a quick randomized check of the random choice. The coin falls on heads in proportion to the quality of the random choice. Next we will make the formal definitions; then we will follow with an example.

Definition 4.2 (Randomness checker)**.** Let Ω be a space of randomness strings. Let $grade : \Omega \rightarrow [0, 1]$ assign each randomness string a grade. A *randomness checker* is a randomized algorithm that given $r \in \Omega$ accepts with probability $grade(r)$.

Definition 4.3 (Algorithm with quick check)**.** Let $t_{check} : \mathbb{N} \rightarrow \mathbb{N}$ 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 $grade_x : \Omega_n \rightarrow [0, 1]$ 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 $t_{check}(n)$.

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

For example, suppose that the algorithm is given as input a graph, and its task is to find a cut that contains at least (roughly) a half of the edges in the graph. Consider the algorithm that picks a random cut and – if the cut contains at least half of the edges – outputs it. Let the grade of the randomness be the fraction of edges in the cut (so, a grade above $1/2$ corresponds to "good" randomness). The randomness checker picks a random edge in the graph and checks whether it is in the cut, which takes $O(1)$ time. The idea here is to somehow use this fact so that instead of having to amplify by running the algorithm many times, instead we can pick a randomness, and run some quick checks on it. If the algorithm fails too many of the quick tests, we have reason to believe the randomness was bad, and instead of computing the size of the cut, we can simply discard the randomness and resample. It's not clear at what point to discard a randomness, or decide to use it, and our algorithm for the biased coin problem shows a way to find randomnesses with a high grade without too much overhead in runtime. Translated to this toy max-cut example, it shows us how to find a cut with at least $1/2 - \epsilon$ fraction of the edges in the graph efficiently, without having to count the sizes of many cuts, but instead picking cuts, and sampling only *few* edges (i.e., running quick checks) in a clever way so as not to waste much time on randomnesses corresponding to small cuts.

Suppose that the desired error probability for the amplified algorithm is $\exp(-k)$. Given an input to the randomized algorithm 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 4.4 (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 4.5 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). In Section 4.4 we discuss various extensions that are useful for the applications in this paper.

4.2 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 have been 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, η, ζ)

Set $i_0 = \log(1/\zeta^2) + \Theta(1)$; $i_f = \log(n/\zeta^2) + \Theta(1)$ (constants picked appropriately).
Pick a coin at random.
for $i = i_0, i_0 + 1, \ldots, i_f$
    Toss the coin for $k = 2^i$ times.
    If the fraction of heads is less than $1-\eta-\zeta/2$, restart.
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 $\exp(-n)$.

4.3 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 [14, 34] 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.

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

Lemma 4.5. *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. If the algorithm returns a coin in Step 6, then it must have tossed the coin for more than $16n/\zeta^2$ times and the fraction of heads is at least $1-\eta-\zeta/2$. By the Chernoff bound, the probability that this happens when the coin's bias is less than $1-\eta-\zeta$ is at most $\exp(-n)$. It

```
Find-Biased-Coin(n, η, ζ)
1  Set i_0 = log(1/ζ^2) + Θ(1);  i_f = log(n/ζ^2) + Θ(log log(n/ζ));  β = ζ/(2 i_f).
2  Pick a coin at random.
3  for i = i_0, i_0 + 1, ..., i_f
4       Toss the coin for k = 2^i times.
5       If the fraction of heads is less than 1 − η − iβ, 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.

remains to analyze the number of coin tosses in runs the end with restarting. We will show that, for any possible invocation of Find-Biased-Coin, the probability of a restart after T coin tosses is exponentially small in $\tilde{\Theta}(T/\zeta^2)$. Note that the lemma would follow: let the number of coin tosses in runs that end with restart be $T_1, T_2, \ldots, T_k$. Then the total number of coin tosses is $\sum T_i$ and the total probability of this happening is exponentially small in $\tilde{\Theta}(\sum T_i/\zeta^2)$.

Suppose that the algorithm restarts at iteration i. The number of coin tosses made by this point since the previous restart (if any) is $2^{i_0} + 2^{i_0+1} + \ldots + 2^i \leq 2^{i+1}$. There are two cases regarding the bias of the coin:

1. The coin has bias smaller than $1-\eta-i\beta+\beta/2$. If $i > i_0$, by a Chernoff bound, the probability the coin passed the test of Step 5 in iteration $i-1$, where the coin 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.

2. The coin has bias at least $1-\eta-i\beta+\beta/2$. By the Chernoff bound, the probability the coin failed the test of Step 5 in phase i, 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 $O(1/\beta^2)$. By the choice of β, the coin tosses to exponent ratio is $O((1/\zeta^2)\log^2(n/\zeta))$.

□

4.4 Extensions

Suppose that the randomized algorithm we'd like to amplify picks randomness and then considers several different ways to use it. For instance, the randomized algorithm for Max-Cut of Section 1.6.1 picks a sample of the vertices, and then considers all possible cuts of the sample; each defining a different global cut of the graph. The randomized algorithm for Clique of Section 1.6.2 picks a sample of the vertices and considers all possible sub-cliques within the sample; each defining a different global set of vertices, depending on what vertices were connected to the sub-cliques. Some of the ways to use the randomness may be *faulty*: for example, the sample may have a sub-clique that is not connected to many vertices outside the sample, and therefore cannot define a large set of vertices. In this section we develop the machinery to handle such cases.

We consider a general setting where coins are divided into groups, and rather than directly tossing coins, we simulate tossing. The simulation may fail or may produce results that are inconsistent with the true bias of the coin. Some of the coins may be faulty, and their tossing may fail arbitrarily. In this case, we imagine the quick check of the randomness outputs $\perp$. For other coins, the probability that tossing fails is small. For any coin, the probability that the coin toss does not fail and is inconsistent with the true bias is small. The coins are partitioned into groups of size g each. The bias of a group is the maximum bias among non-faulty coins in the group, and is 0 if all the coins in the group are faulty. At least $2/3$ fraction of the groups have bias at least $1-\eta$. The task is to find a group of coins of bias at least $1-\eta-\zeta$.

The formal requirements from a simulated coin toss are as follows:

Definition 4.6. Simulated coin tosses satisfy the following (in this setting, each coin outputs heads, tails, or $\perp$):

- For any coin, when tossing the coin k times, there is exponentially small probability in $\beta^2 k$ for the following event: the tosses do not all fail (i.e., $\perp$ is not outputted by all the coin toss), yet the fraction of tosses that fall on heads deviates from the true bias by more than an additive $\beta/4$ for β as in Figure 3.
- For non-faulty coins, the probability that tossing the coin fails is exponentially small in $\beta^2 k$.

Note that the simulation has to be very accurate, since the deviation $\beta/4$ is sub-constant. We describe a modified biased coin algorithm in Figure 3.

```
Find-Biased-Coin-in-Group(n, g, η, ζ)
1  Set i_f = log((n + log g)/ζ^2) + Θ(log log((n + log g)/ζ));   β = ζ/(2 i_f).
2  Set i_0 = log(log g/β^2) + Θ(1), where the constant term is sufficiently large.
3  Pick a group of coins at random.
4  for i = i_0, i_0 + 1, ..., i_f
5        Simulate tossing each one of the g coins in the group for k = 2^i times.
6        If the maximum fraction of heads per coin is less than 1 − η − iβ, restart.
7  return group of coins.
```

Figure 3: An algorithm for finding a group of coins of bias at least $1-\eta-\zeta$, where the coins are partitioned into groups of size g each. The error probability is exponentially small in n.

Lemma 4.7 (Generalized biased coin)**.** *If* Find-Biased-Coin-in-Group *restarts at a certain phase, then either in this phase or in the previous, the reported fraction of heads deviates by more than $\beta/2$ from the true bias for one of the non-faulty coins in the group, or it is the first phase and a group of bias at most $1-\eta-\beta/2$ was picked.*

As a result, within $O((ng\log g/\zeta^2)\log^2((n+\log g)/\zeta)) = \tilde{O}(ng/\zeta^2)$ coin tosses Find-Biased-Coin-in-Group *outputs a coin of bias at least $1-\eta-\zeta$ except with probability* $\exp(-n)$.

Proof. If the algorithm returns a group of coins in Step 7, then it must have tossed the g coins for more than $16n/\zeta^2$ times and the max fraction of heads is at least $1-\eta-\zeta/2$. By the Chernoff bound, the probability that this happens when the biases are all less than $1-\eta-\zeta$ is at most $\exp(-n)$. It remains to analyze the number of coin tosses in runs the end up with restarting.

Suppose that the algorithm restarts at phase i. The number of coin tosses made by this point is $g \cdot (2^{i_0} + 2^{i_0+1} + \ldots + 2^i) \leq g \cdot 2^{i+1}$.

Suppose that the group had bias smaller than $1 - \eta - i\beta + \beta/2$. If $i > i_0$, the probability that the coins passed the previous test, where at least one of them was supposed to have at least $1-\eta-(i-1)\beta$ fraction of heads, is $g\cdot(\exp(-\beta^2 2^{i-4})+\exp(-\Omega(\beta^2 2^{i-1})))$ (where $\beta/4$ of the deviation and $\exp(-\Omega(\beta^2 2^{i-1}))$ of the error probability can be attributed to the simulation). Note that this probability is exponentially small in $\beta^2 2^i$ when $\log g$ is sufficiently smaller than $\beta^2 2^{i_0-1}$ (here we use the choice of i_0). If $i = i_0$, the probability that a group of bias smaller than $1-\eta$ was picked is less than $1/3$.

On the other hand, if the group has bias at least $1-\eta-i\beta+\beta/2$, then the probability it failed the current test, where one of the coins is supposed to have at least $1-\eta-i\beta$ fraction of heads, is at most $\exp(-\beta^2 2^{i-3}) + \exp(-\Omega(\beta^2 2^i))$ (again, $\beta/4$ of the deviation and $\exp(-\Omega(\beta^2 2^i))$ of the error probability can be attributed to the simulation).

In any case, the ratio between the number of coin tosses and the exponent of the probability is $O(g \log g/\beta^2)$. By the choice of β, the coin tosses to exponent ratio is $O((g\log g/\zeta^2)\log^2((n + \log g)/\zeta))$.

□

5 Max Cut on Dense Graphs

In this section we show the application to Max-Cut on dense graphs. This is our simplest example. It relies on the biased coin algorithm, and does not require any sketches.

5.1 A Simple Randomized Algorithm

First we describe a simple randomized algorithm for dense Max-Cut based on the sampling idea of Fernandez de la Vega [11] and Arora, Karger and Karpinski [7]. We remark that Mathieu and Schudy [36] have similar, but more efficient, randomized algorithms, however, for the sake of simplicity, we stick to the simplest algorithm with the easiest analysis.

The main idea of the algorithm is as follows. We sample a small $S \subseteq V$ and enumerate over all possible S-cuts $H \subseteq S$. Each S-cut induces a cut $C_{S,H} \subseteq V$ as follows.

Definition 5.1 (Induced cut)**.** Let $G = (V, E)$. Let $S \subseteq V$ and $H \subseteq S$. We define $C_{S,H} \subseteq V$ as follows: for every $v \in V$ let $v \in C_{S,H}$ if the fraction of edges $e = (v, s) \in E$ with $s \in S - H$ is larger than the fraction of edges $e = (v, s) \in E$ with $s \in H$.

We will argue below that if there is a cut in G with value at least $1-\varepsilon$ and H is the restriction of that cut to S, then the induced cut is likely to approximately achieve the optimal value. Note that we rely on density when we hope that the edges that touch the small set S span most of the vertices in the graph.

Lemma 5.2 (Sampling)**.** *Let $G = (V, E)$ be a regular γ-dense graph that has a cut of value at least $1-\varepsilon$ for $\varepsilon < 1/4$. Then for $\zeta < 1/4 - \varepsilon$ and for a uniform $S \subseteq V$,*

$$|S| = \max\left\{\lceil \log(2/\zeta^2)/\zeta^2\rceil, \lceil 2\log(2/\zeta^2)/\gamma^2\rceil\right\},$$

with probability at least $1-\zeta$*, there exists* $H \subseteq S$ *such that the value of the cut* $C_{S,H}$ *is at least* $1-\varepsilon-10\zeta$*.*

Proof. Let C^* be the optimal cut in G. Let $H \subseteq S$ be the restriction of C^* to S. Denote by V' the set of all $v \in V$ such that at least $1/2+\zeta$ fraction of the edges that touch v contribute to the value of C^*. Note that $|V'| \geq (1-4\zeta)|V|$. By γ-density and regularity, the degree of all vertices is $\gamma|V|$. By a Chernoff bound, except with probability $\zeta^2/2$ over S, at least $(\gamma/2)|S|$ of the vertices in S are neighbors of v. The sample of v's neighbors is uniform and hence by another Chernoff bound, except with probability $\zeta^2/2$ over S, the vertex v is assigned by $C_{S,H}$ to the same side as C^* assigns it. Therefore, except with probability ζ over the random choice of S, at least $1-\zeta$ fraction of the vertices $v \in V'$ are assigned by $C_{S,H}$ the same as C^*. This means that at least $1-5\zeta$ fraction of the vertices $v \in V$ are assigned by $C_{S,H}$ the same as C^*. Therefore, the fraction of edges that: (i) contribute to the value of C^*, and (ii) have both their endpoints assigned by $C_{S,H}$ the same as C^*, is at least $1-\varepsilon-2\cdot 5\zeta = 1-\varepsilon-10\zeta$. □

5.2 A Randomized Algorithm With Exponentially Small Error Probability

We describe an analogy between finding a cut of high value and finding a biased coin. We think of sampling $S \subseteq V$ as picking a group of coins, and picking $H \subseteq S$ as picking a coin in the group. The bias of the coin is the value of the cut $C_{S,H}$. Therefore a biased coin directly corresponds to a desirable cut. One tosses a coin by picking an edge $(u,v) \in E$ uniformly at random, computing whether $u \in C_{S,H}$ and whether $v \in C_{S,H}$, and checking whether the edge contributes to the value of the cut. Note that checking whether a vertex belongs to $C_{S,H}$ is computed in time $|S|$. The coin toss algorithm is described in Figure 4. This algorithm, at the high level, simply figures out what cut the randomness corresponds to, and then picks an edge at random and tests whether the edge crosses the cut. If the edge does cross the cut, this is evidence that the cut is good. The algorithm based on finding a biased coin is described in Figure 5.

Max-Cut-Toss-Coin($G=(V,E),S,H$)
1 Pick $e=(u,v) \in E$ uniformly at random.
2 **return** "heads" iff $u \in C_{S,H}$ and $v \notin C_{S,H}$ or vice versa.

Figure 4: A coin toss picks an edge at random and checks whether it contributes to the value of the cut $C_{S,H}$.

Note the similarity in structure between the amplified algorithm and the algorithm for finding a biased coin from Section 4. This is because the amplified algorithm is simply using the algorithm for finding a biased coin, where instead of coin flips we use Max-Cut-Toss-Coin.

This proves Theorem 1.1, which is repeated below for convenience. Note that for a sufficiently small error probability exponentially small in $|V|^2$ it follows that there exists a randomness string on which the algorithm succeeds, no matter what the input is.

Theorem 5.3. *There is a Las Vegas algorithm that given a* γ*-dense graph* G *that has a cut of value at least* $1-\varepsilon$ *for* $\varepsilon<1/4$*, and given* $\zeta<1/4-\varepsilon$*, finds a cut of value at least* $1-\varepsilon-O(\zeta)$*, except with probability exponentially small in* $|V|^2$*. The algorithm runs in time* $\tilde{O}(|V|^2\,(1/\zeta)^{O(1/\gamma^2+1/\zeta^2)})$*. It also implies a non-uniform deterministic algorithm with the same run-time.*

Find-Cut($G = (V, E), \varepsilon, \zeta$)

Set $s = \max\left\{\lceil \log(2/\zeta^2)/\zeta^2 \rceil, \lceil 2\log(2/\zeta^2)/\gamma^2 \rceil\right\}$, where γ is the density of G.
Set $i_f = \log((|V|^2 + s)/\zeta^2) + \Theta(\log\log((|V| + s)/\zeta))$; $\beta = \zeta / i_f$.
Set $i_0 = \log(s/\beta^2) + \Theta(1)$.
Sample $S \subseteq V$, $|S| = s$.
for $i = i_0, i_0 + 1, \ldots, i_f$
    for all $H \subseteq V$
        Invoke Max-Cut-Toss-Coin(G, S, H) for $k = 2^i$ times.
    If the fraction of heads is less than $1 - \varepsilon - 10\zeta - i\beta$ for all H, restart.
return cut $C_{S,H}$ with value at least $1 - \varepsilon - 11\zeta$ if exists.

Figure 5: An algorithm for finding a cut of value $1-\varepsilon-O(\zeta)$ in a regular γ-dense graph that has a cut of value $1-\varepsilon$. The error probability of the algorithm is exponentially small in $|V|^2$.

6 Approximate Clique

6.1 An Algorithm With Constant Error Probability

In this section we describe a randomized algorithm with constant error probability for finding an approximate clique in a graph that has a large clique. The algorithm is a simplified version of an algorithm and analysis by Goldreich, Goldwasser and Ron [22]. We rely on the algorithm and the analysis when we design a randomized algorithm with error probability $\exp(-\Omega(|V|))$ and again when we design a deterministic algorithm.

The main idea of the algorithm is as follows. We first find a small random subset U' of the large clique C by sampling vertices U from V and enumerating over all possibilities for $C \cap U$. The intuition is that now we would like to find other vertices that are part of the large clique C. A natural test for whether a vertex $v \in V$ is in the clique is whether v is connected to all the vertices in U'. This, however, is not a sound test, since the clique might have many vertices that neighbor it but do not neighbor one another. A better test checks whether v neighbors all of U', as well as many of the vertices that neighbor all of U'. Vertices that neighbor all of U' are likely to neighbor almost all of the clique.

The algorithm is described in Figure 6. It picks $U \subseteq V$ at random, considers all possible sub-cliques $U' \subseteq U$, $|U'| \geq (\rho/2)\,|U|$, computes $\Gamma(U')$ the set of vertices that neighbor all of U', computes for every vertex in $\Gamma(U')$ the fraction of vertices in $\Gamma(U')$ that neighbor it, and considers $S_{U'}$ the set of $\rho\,|V|$ vertices in $\Gamma(U')$ with largest fractions of neighbors. The algorithm outputs a sufficiently dense set among all sets $S_{U'}$, if such exists.

The algorithm runs in time $\exp(k_0/\rho) \cdot O(|V|^2)$. Next we analyze the probability it is correct. By a Chernoff bound, we have $|U \cap C| \geq (\rho - \varepsilon)\,|U|$, except with probability $1/10$. Pick a uniformly random order on the vertices. Let us focus on the event $|U \cap C| \geq (\rho - \varepsilon)\,|U|$ and U' that is the first $(\rho/2)\,|U|$ elements in $U \cap C$ according to the random order. Note that the elements of U' are uniformly and independently distributed in C. Let $\Gamma(U') \subseteq V$ contain all the vertices that neighbor all of U'.

Lemma 6.1. *With probability $1 - e^{-25/\varepsilon}$ over the choice of U', the fraction of $v \in \Gamma(U')$ that*

Find-Approximate-Clique-Constant-Error($G = (V, E), \rho, \varepsilon$)

Sample $U \subseteq V$, $|U| = \lceil k_0/\rho \rceil$, for $k_0 = 100/\varepsilon^2$.
for all sub-cliques $U' \subseteq U$, $|U'| \geq (\rho/2)\,|U|$,
  Compute $\Gamma(U')$ the set of vertices that neighbor all of U'.
  For each $v \in \Gamma(U')$ compute the fraction f_v of vertices in $\Gamma(U')$ that neighbor v.
  Let $S_{U'} \subseteq \Gamma(U')$ contain the $\rho\,|V|$ vertices with largest f_v.
return set $S_{U'}$ of density at least $1 - 2\varepsilon/\rho$ if such exists.

Figure 6: Randomized algorithm with constant error probability for finding an approximate clique.

neighbor less than $1 - \varepsilon$ *fraction of* C *is at most* $e^{-25/\varepsilon}$.

Proof. Consider $v \in V$ that has less than $1-\varepsilon$ neighbors in C. For v to be in $\Gamma(U')$ the set U' must miss all of the non-neighbors of v. Since U' is a uniform sample of C, this happens with probability $(1-\varepsilon)^{|U'|} \leq e^{-50/\varepsilon}$. The lemma follows. □

Let us focus on U' for which the fraction of $v \in \Gamma(U')$ that neighbor less than $1 - \varepsilon$ fraction of C is at most $e^{-25/\varepsilon}$. Lemma 6.1 guarantess that such a U', which we call *good*, is picked with constant probability. Next we show that an average vertex in C neighbors most of $\Gamma(U')$.

Lemma 6.2 (Density for C)**.** *For good* U'*, the average number of neighbors a vertex* $c \in C$ *has in* $\Gamma(U')$ *is at least* $(1 - 2\varepsilon) \cdot |\Gamma(U')|$.

Proof. Since U' is good, more than $1-e^{-25/\varepsilon}$ fraction of $\Gamma(U')$ neighbor at least $1-\varepsilon$ fraction of C. Hence, the average fraction of $\Gamma(U')$ neighbors a uniform vertex in C has is at least $1 - 2\varepsilon$ (using $e^{-25/\varepsilon} \leq \varepsilon$). □

We can now prove the correctness of Find-Approximate-Clique-Constant-Error.

Lemma 6.3. *With probability at least* $1-e^{-25/\varepsilon}$, Find-Approximate-Clique-Constant-Error, *when invoked on* $0 < \rho, \varepsilon < 1$*, and a graph* $G = (V, E)$ *with a clique on* $\rho\,|V|$ *vertices, picks* $S_{U'}$ *such that* $(1/|S_{U'}|) \cdot \sum_{v \in S_{U'}} f_v \geq 1 - 2\varepsilon$*, and returns a set of density at least* $1 - 2\varepsilon/\rho$.

Proof. For good U', by Lemma 6.2, $(1/|C|) \sum_{v \in C} f_v \geq 1 - 2\varepsilon$. Since $S_{U'}$ takes the $\rho\,|V|$ vertices with largest f_v and $|C| \geq \rho\,|V|$, we have $(1/|S_{U'}|) \cdot \sum_{v \in S_{U'}} f_v \geq 1 - 2\varepsilon$. Therefore, the density within $S_{U'}$ is at least $1 - 2\varepsilon/\rho$, and so is the density of the set returned by the algorithm. □

Next we show how to transform the randomized algorithm with constant error probability from Section 6.1 into an algorithm with error probability that is exponentially small in $|V|$ *without increasing the run-time by more than poly-logarithmic factors*. The algorithm applies the biased coin algorithm from Section 4.

6.2 Finding an Approximate Clique as Finding a Biased Coin

The analogy between finding a biased coin and finding an approximate clique is as follows: Picking U picks a group of coins. There is a coin for every $U' \subseteq U$, $|U'| \geq (\rho/2)\,|U|$. The coin is faulty if

$|\Gamma(U')| < \rho\,|V|$. A coin corresponds to the set $S_{U'}$ of the $\rho\,|V|$ vertices in $\Gamma(U')$ with largest number of neighbors in $\Gamma(U')$ (when the coin is faulty, pad the set with dummy vertices with 0 neighbors). The bias of the coin $bias_{U'}$ is the expectation, over the choice of a random vertex $v \in S_{U'}$, of the fraction of vertices in $\Gamma(U')$ that neighbor v. With at least $2/3$ probability, one of the coins in the group – the one associated with a good U' in the sense of Section 6.1 – has $bias_{U'} \geq 1 - 2\varepsilon$. Moreover, any U' with $bias_{U'} \geq 1 - c\varepsilon$ corresponds to a set of density at least $1 - c\varepsilon/\rho$.

Had we found the vertices in each $S_{U'}$, we could have tossed a coin by picking a vertex at random from $S_{U'}$ and a vertex at random from $\Gamma(U')$ and letting the coin fall on heads if there is an edge between the two vertices. Unfortunately, finding the vertices in $S_{U'}$ may take $\Omega(|V|^2)$ time, so we refrain from doing that. We settle for a simulated toss – where with high probability the coin falls on heads with probability close to its bias. In Section 4.4 we extended the biased coin algorithm to simulated tosses. In Figure 7 we describe the algorithm for tossing a coin enough times so the probability of γ-deviation from the true bias is exponentially small in k (the number of coin tosses is implicit). The algorithm runs in time $O(k\,|V|\,|U'|\,\mathrm{poly}(1/\rho, 1/\gamma))$.

Clique-Coin-Toss$(G = (V, E), U', \rho, k, \gamma)$

Compute $\Gamma(U')$.
if $|\Gamma(U')| < \rho\,|V|$
    Fail.
Sample $V' \subseteq V$, $|V'| = \lceil k/(\rho\gamma^2)\rceil$.
For all $v \in V'$ compute the fraction f_v of $\Gamma(U')$ vertices that neighbor v.
Let $S_{U',V'} \subseteq V' \cap \Gamma(U')$ contain the $\rho\,|V'|$ vertices with largest f_v.
return $bias_{U'}^{V'} \doteq (1/\rho\,|V'|) \sum_{v \in S_{U',V'}} f_v$ heads.

Figure 7: An algorithm for tossing the coin associated with U', where the coin falls on heads with probability $\Theta(\gamma)$-close to its bias except with probability exponentially small in k.

In the next lemma we prove that $bias_{U'}^{V'}$ is likely to approximate $bias_{U'}$ well. For future use we phrase a more general statement than we need here, addressing U' that defines a slightly faulty coin as well.

Lemma 6.4. *Assume that $|\Gamma(U')| \geq (1-\gamma')\rho\,|V|$, where $\gamma' = \Theta(\gamma)$ and $\gamma, \gamma' \leq 1/4$. For a uniform $V' \subseteq V$, except with probability exponentially small in $\rho\gamma^2\,|V'|$,*

$$\left| bias_{U'}^{V'} - bias_{U'} \right| \leq 3\gamma + 2\gamma'.$$

Proof. By a multiplicative Chernoff bound, except with probability exponentially small in $\rho\gamma^2\,|V'|$, there are $(1 \pm \gamma \pm \gamma')\rho\,|V'|$ vertices in $V' \cap S_{U'}$. Let us focus on this event.

By a Hoeffding bound, except with probability exponentially small in $\rho\gamma^2\,|V'|$, we have

$$\left| \frac{1}{|V' \cap S_{U'}|} \sum_{v \in V' \cap S_{U'}} f_v - \frac{1}{|S_{U'}|} \sum_{v \in S_{U'}} f_v \right| \leq \gamma.$$

Hence,

$$\left| \frac{1}{\rho\,|V'|} \sum_{v \in V' \cap S_{U'}} f_v - \frac{1}{|S_{U'}|} \sum_{v \in S_{U'}} f_v \right| \le 3\gamma + 2\gamma'.$$

The lemma follows. □

The algorithm for finding an approximate clique using Clique-Coin-Toss is described in Figure 8. Note that the coin tossing algorithm satisfies the conditions of simulated tossing (Definition 4.6). Lemma 6.1 and Lemma 6.2 ensure that with constant probability over the choice of U, for U' as specified in Section 6.1, we have $bias_{U'} \ge 1 - 2\varepsilon$ for one of the $U' \subseteq U$. Moreover, a coin with bias at least $1 - c\varepsilon$ yields a set which is at least $1 - c\varepsilon/\rho$-dense, and this set can be computed in $O(|V|^2)$ time. Therefore, the algorithm in Figure 8 gives an algorithm for finding an approximate clique that errs with probability exponentially small in $|V|$ and runs in time $\tilde{O}(|V|^2\, 2^{O(1/\varepsilon^2\rho)})$. This proves part of Theorem 1.2 repeated below for convenience (note that ε in Theorem 1.2 is replaced with $O(\varepsilon/\rho)$ here).

Theorem 6.5. *There is a Las Vegas algorithm that given a graph $G = (V, E)$ with a clique on $\rho\,|V|$ vertices and given $0 < \rho, \varepsilon < 1$, finds a set of $\rho\,|V|$ vertices and density $1 - O(\varepsilon/\rho)$, except with probability exponentially small in $|V|$. The algorithm runs in time $\tilde{O}(|V|^2\, 2^{O(1/(\varepsilon^2\rho))})$.*

The remainder of the section constructs an oblivious verifier for Find-Approximate-Clique and uses it to prove the second part of Theorem 1.2 (a deterministic algorithm). First we describe the sketch and its properties, then we devise an oblivious verifier for Clique-Coin-Toss, and finally we describe the verifier for Find-Approximate-Clique.

Find-Approximate-Clique($G = (V, E), \rho, \varepsilon$)

Set $u = \lceil 100/(\varepsilon^2\rho) \rceil$.
Set $i_f = \log((|V| + u)/\varepsilon^2) + \Theta(\log\log((|V| + u)/\varepsilon))$; $\beta = \varepsilon/i_f$.
Set $i_0 = \log(u/\beta^2) + \Theta(1)$.
Sample $U \subseteq V$, $|U| = u$.
for $i = i_0, i_0 + 1, \ldots, i_f$
  Set $k = 2^i$.
  for all $U' \subseteq U$, $|U'| \ge (\rho/2)\,|U|$
    Clique-Coin-Toss($G, U', \rho, \beta^2 k, \gamma = \beta/100$). If fails, skip this U'.
  If the fraction of heads is less than $1 - 2\varepsilon - i\beta$ for all (non-skipped) U', restart.
return set $S_{U'}$ of density at least $1 - 3\varepsilon/\rho$ if such exists.

Figure 8: An algorithm for finding an approximate clique in a graph $G = (V, E)$ that contains a clique on $\rho\,|V|$ vertices. The error probability of the algorithm is exponentially small in $|V|$.

6.3 A Sketch for Approximate Clique

The sketch for a given G contains, for some carefully chosen set R of poly$(\log|V|, 1/\varepsilon, 1/\rho)$ vertices, the bipartite graph $G_R = (R, V, E_R)$ that contains all the edges of G that at least one of their endpoints falls in R. The set R is chosen so it allows the verifier to estimate the f_v's corresponding to different sets $U' \subseteq V$. Note that the size of the sketch is $|R|\,|V|$.

Let $U' \subseteq U$. For every $v \in V$ we denote by f_v the fraction of vertices in $\Gamma(U')$ that neighbor v. For $V' \subseteq V$, let $S_{U',V'} \subseteq V' \cap \Gamma(U')$ denote the $\rho\,|V'|$ elements $v \in V' \cap \Gamma(U')$ with largest f_v (pad with dummy vertices with 0 neighbors if needed). Let $bias_{U'}^{V'} \doteq (1/\rho\,|V'|)\sum_{v \in S_{U',V'}} f_v$. For $v \in V$ let $\tilde{f}_v$ denote the fraction of $\Gamma(U') \cap R$ vertices that neighbor v among all vertices in $\Gamma(U') \cap R$. For $V' \subseteq V$, let $\tilde{S}_{U',V'}$ be the $\rho\,|V'|$ vertices $v \in V'$ with largest $\tilde{f}_v$ (pad with dummy vertices with 0 neighbors if needed). Let $\tilde{bias}_{U'}^{V'} \doteq (1/\rho\,|V'|)\sum_{v \in \tilde{S}_{U',V'}} \tilde{f}_v$.

In the lemma we use u, ρ, γ from Find-Approximate-Clique in Figure 8.

Lemma 6.6 (Sketch)**.** *There exists $R \subseteq V$, $|R| = O(u \log |V| / \rho\gamma^2)$, such that for every $U' \subseteq V$, $|U'| \le u$,*

1. *If $|\Gamma(U')| \ge \rho\,|V|$, then $|R \cap \Gamma(U')| \ge (1-\gamma)\rho\,|R|$, whereas if $|\Gamma(U')| < (1-2\gamma)\rho\,|V|$, then $|R \cap \Gamma(U')| < (1-\gamma)\rho\,|R|$.*
2. *Suppose that $|\Gamma(U')| \ge (1-2\gamma)\rho\,|V|$. Then, for every $v \in V$, we have $\left|\tilde{f}_v - f_v\right| \le \gamma$.*
3. *Suppose that $|\Gamma(U')| \ge (1-2\gamma)\rho\,|V|$. Then, $\left|bias_{U'}^{R} - bias_{U'}\right| \le 7\gamma$.*

Proof. Pick $R \subseteq V$ uniformly at random. Let $U' \subseteq U$, $|U'| \le u$. By a multiplicative Chernoff bound, if $|\Gamma(U')| \ge \rho\,|V|$, then $|R \cap \Gamma(U')| \ge (1-\gamma)\rho\,|R|$, except with probability exponentially small in $\rho\gamma^2\,|R|$. If $|\Gamma(U')| < (1-2\gamma)\rho\,|V|$, then $|R \cap \Gamma(U')| \le (1-\gamma)\rho\,|R|$ except with probability exponentially small in $\rho\gamma^2\,|R|$.

Suppose that $|\Gamma(U')| \ge (1-2\gamma)\rho\,|V|$. Let $v \in V$. By a multiplicative Chernoff bound, except with probability exponentially small in $\rho\gamma^2\,|R|$, we have $|\Gamma(U') \cap R| \ge (1-3\gamma)\,|R|$. By a Chernoff bound, except with probability exponentially small in $\rho\gamma^2\,|R|$, we have

$$\left|\tilde{f}_v - f_v\right| \le \gamma.$$

By a union bound over all v and by the choice of $|R|$, the last inequality holds for all $v \in V$ except with probability exponentially small in $\rho\gamma^2\,|R|$.

By Lemma 6.4, if $|\Gamma(U')| \ge (1-2\gamma)\rho\,|V|$, except with probability exponentially small in $\rho\gamma^2\,|R|$, we have

$$\left|bias_{U'}^{R} - bias_{U'}\right| \le 7\gamma.$$

Since there are less than $|V|^u$ choices for U', it follows from a union bound that there exists R for which all three items hold for all $U' \subseteq V$, $|U'| \le u$. ☐

Lemma 6.7. *Suppose that $|\Gamma(U')| \ge (1-2\gamma)\rho\,|V|$. For any $V' \subseteq V$,*

$$\left|\tilde{bias}_{U'}^{V'} - bias_{U'}^{V'}\right| \le 2\gamma.$$

Proof. By Lemma 6.6, the contribution from $v \in V'$ in $\tilde{S}_{U',V'} \cap S_{U',V'}$ is at most γ since $\left|\tilde{f}_v - f_v\right| \le \gamma$. It remains to bound the contribution from other elements $v \in V'$ that are either in $\tilde{S}_{U',V'} - S_{U',V'}$ or in $S_{U',V'} - \tilde{S}_{U',V'}$. Pair those vertices arbitrarily, and consider a single pair $v_2 \in \tilde{S}_{U',V'} - S_{U',V'}$

and $v_1 \in S_{U',V'} - \tilde{S}_{U',V'}$. We know that $f_{v_1} \geq f_{v_2} \geq \tilde{f}_{v_2} - \gamma$, so $\tilde{f}_{v_2} - f_{v_1} \leq \gamma$. Similarly, $\tilde{f}_{v_2} \geq \tilde{f}_{v_1} \geq f_{v_1} - \gamma$, so $f_{v_1} - \tilde{f}_{v_2} \leq \gamma$. In any case, $\left| f_{v_1} - \tilde{f}_{v_2} \right| \leq \gamma$. The triangle inequality implies the lemma. □

Corollary 6.8. *For every $U' \subseteq V$, $|U'| \leq u$, either $|R \cap \Gamma(U')| < (1-2\gamma)\rho|V|$, or*

$$\left| \tilde{bias}^R_{U'} - bias_{U'} \right| \leq 9\gamma.$$

Proof. If $|R \cap \Gamma(U')| \geq (1-2\gamma)\rho|V|$, by Lemma 6.6, we have $\left| bias^R_{U'} - bias_{U'} \right| \leq 7\gamma$. By Lemma 6.7, $\left| \tilde{bias}^R_{U'} - bias^R_{U'} \right| \leq 2\gamma$. The claim follows. □

Interestingly, our construction of the sketch is randomized, yet it will allow us to obtain a deterministic algorithm. The reason is that we only need the existence of a sketch describing an input so we can take a union bound over all possible sketches.

6.4 Obliviously Checking V'

Next we show how we can check the sample V' of Clique-Coin-Toss using the sketch. The oblivious verifier receives a sketch G_R of the graph G, the rest of the input of Clique-Coin-Toss and the randomness V' used by the algorithm. The verifier accepts iff $bias^{V'}_{U'}$ is approximately $bias_{U'}$. It uses the sketch to approximate $bias_{U'}$ via $\tilde{bias}^R_{U'}$. It is described in Figure 9.

Oblivious-Verifier-Clique-Coin-Toss$(G_R, U', \rho, k, \gamma, V')$
1 If $|R \cap \Gamma(U')| < (1-\gamma)\rho|R|$
2 Fail.
3 For all $v \in V'$ compute the fraction $\tilde{f}_v$ of vertices in $\Gamma(U') \cap R$ that neighbor v.
4 Let $\tilde{S}_{U',V'}$ be the $\rho|V'|$ vertices $v \in V'$ with largest $\tilde{f}_v$.
5 Let $\tilde{bias}^{V'}_{U'} \doteq (1/\rho|V'|)\sum_{v \in \tilde{S}_{U',V'}} \tilde{f}_v$.
6 Accept iff $\left| \tilde{bias}^{V'}_{U'} - \tilde{bias}^R_{U'} \right| \leq 18\gamma$.

Figure 9: An oblivious verifier for Clique-Coin-Toss.

Recall that by Lemma 6.6, if the coin is non-faulty and $|\Gamma(U')| \geq \rho|V|$, then $|R \cap \Gamma(U')| \geq (1-\gamma)\rho|R|$, so Oblivious-Verifier-Clique-Coin-Toss does not fail in Step 2. Moreover, if $|\Gamma(U')| < (1-2\gamma)\rho|V|$, then $|R \cap \Gamma(U')| < (1-\gamma)\rho|R|$, and Oblivious-Verifier-Clique-Coin-Toss necessarily fails.

Lemma 6.9. *The following hold:*

1. *If* Oblivious-Verifier-Clique-Coin-Toss *accepts then $|\Gamma(U')| \geq (1-2\gamma)\rho|V|$ and V' sampled by* Clique-Coin-Toss *satisfies $\left| bias^{V'}_{U'} - bias_{U'} \right| \leq 29\gamma$.*

2. *If $|R \cap \Gamma(U')| \geq (1-\gamma)\rho|V|$, then for a randomly chosen V' the probability that* Oblivious-Verifier-Clique-Coin-Toss *rejects is exponentially small in k (where k is a parameter that is given as input to* Clique-Coin-Toss *and* Oblivious-Verifier-Clique-Coin-Toss*, and is proportional to the size of V').*

Proof. Toward the second item, suppose that $|R \cap \Gamma(U')| \geq (1-\gamma)\rho|V|$. By Lemma 6.7, we have $\left|\tilde{bias}_{U'}^{V'} - bias_{U'}^{V'}\right| \leq 2\gamma$. By Corollary 6.8, we have $\left|\tilde{bias}_{U'}^{R} - bias_{U'}\right| \leq 9\gamma$. By Lemma 6.4, we have $\left|bias_{U'}^{V'} - bias_{U'}\right| \leq 7\gamma$, except with probability exponentially small in k. The low probability of rejection follows.

Toward the first item, suppose that Oblivious-Verifier-Clique-Coin-Toss accepts, so $|R \cap \Gamma(U')| \geq (1-\gamma)\rho|R|$ and $\left|\tilde{bias}_{U'}^{V'} - \tilde{bias}_{U'}^{R}\right| \leq 18\gamma$. By Lemma 6.7, we have $\left|\tilde{bias}_{U'}^{V'} - bias_{U'}^{V'}\right| \leq 2\gamma$. By Corollary 6.8, we have $\left|\tilde{bias}_{U'}^{R} - bias_{U'}\right| \leq 9\gamma$. Thus, we have $\left|bias_{U'}^{V'} - bias_{U'}\right| \leq 29\gamma$. □

6.5 An Oblivious Verifier for Approximate Clique

The verifier is unable to follow the execution of Find-Approximate-Clique nor compute its output, since it can't tell exactly how many heads Clique-Coin-Toss yields. The verifier can be probably approximately correct about the fraction of heads, but it is likely that during the execution of Find-Approximate-Clique some of its predictions would be false, thereby changing the course of execution. It may seem that under these conditions the verifier cannot check the randomness of the algorithm, but this is not so. The key idea is that the verifier is not limited computationally and can try all possible executions of the algorithm (i.e., the outcomes of all possible restart decisions). The verifier discards executions that occur with sufficiently low probability. Thanks to the exponentially low error probability of the algorithm, with high probability the algorithm succeeds on *all* remaining executions. Hence, the identity of the exact execution of the algorithm is unimportant.

Remark 6.1. *The algorithm* Find-Approximate-Clique *uses its randomness as a stream of random bits, and uses independent randomness between restarts. The oblivious verifier for a single execution simulates a possible run of the algorithm and follows the algorithm in its use of the randomness. Different executions use the same randomness.*

The verifier maintains a set $\mathcal{G}$ of possible input graphs G that are consistent with the execution up to this step (initially $\mathcal{G}$ contains all the input graphs that are consistent with the sketch). If the set of inputs becomes empty, then the execution is designated *infeasible*. Otherwise the execution is designated *feasible*. Additionally, the verifier maintains *counter* such that the probability of the execution is at most exponentially small in *counter* (initially, *counter* $= 0$). If *counter* becomes too large, the verifier rejects the execution. If none of the feasible executions get rejected, the verifier accepts. The verifier for a single execution (a single fixing of guesses) is described in Figure 10. Note that we use the shorthand OVCCT for Oblivious-Verifier-Clique-Coin-Toss and that the verifier uses the parameters i_0, i_f, β of the algorithm. The final verifier is described in Figure 11.

Next we analyze Oblivious-Verifier-Approximate-Clique.

Lemma 6.10. *If* Oblivious-Verifier-Approximate-Clique *accepts on a sketch of a graph G*

Oblivious-Verifier-Clique-Execution($\mathcal{G}, G_R = (R, V, E_R), \rho, \varepsilon, r, counter$)

if $\mathcal{G} = \phi$
//Recall $\mathcal{G}$ is the set of possible input graphs consistent with the current execution.
    return infeasible.
if $counter > |V|$
    return reject.
Extract from r the sample $U \subseteq V$ of the algorithm.
//The randomness r encodes which subset of the vertices the algorithm will sample.
if $\tilde{bias}^R_{U'} < 1 - 2\varepsilon - \beta/2 + 9\gamma$ for all U' such that $|R \cap \Gamma(U')| \geq (1-\gamma)\rho\,|V|$
//$\tilde{bias}^R_{U'}$ is the approximation for $bias_{U'}$
    $counter \leftarrow counter + 25/\varepsilon$.
for $i = i_0, i_0 + 1, \ldots, i_f$
    Set $k = 2^i$.
    for all $U' \subseteq U$, $|U'| \geq (\rho/2)\,|U|$
        Extract from r the randomness V' for Clique-Coin-Toss.
        OVCCT$(G_R, U', \rho, \beta^2 k, \gamma = \beta/100, V')$
//The obilivious verifier simulates the coin toss
    if $\exists U', |R \cap \Gamma(U')| \geq (1-\gamma)\rho\,|V|$ such that OVCCT rejects
        $counter \leftarrow counter + \beta^2 k - u$.
//We maintain that the probability of this execution is exponentially small in the counter.
    Guess if $\max\left\{ bias^{V'}_{U'} \,\middle|\, |\Gamma(U')| \geq \rho\,|V| \right\} < 1 - 2\varepsilon - i\beta$ and update $\mathcal{G}$ accordingly.
    if guessed true
        Restart while maintaining $\mathcal{G}$ and the value of $counter$.
return accept iff $\exists U' \subseteq U$, $|R \cap \Gamma(U')| \geq (1-\gamma)\rho\,|V|$, such that $\tilde{bias}^R_{U'} \geq 1 - 3\varepsilon - 9\gamma$.

Figure 10: An oblivious verifier for a single execution of Find-Approximate-Clique (an execution is defined by the outcomes of guesses). We use the shorthand OVCCT for Oblivious-Verifier-Clique-Coin-Toss.

and randomness r, then Find-Approximate-Clique[7] *necessarily finds $U' \subseteq V$ with $bias_{U'} \geq 1 - 3\varepsilon - 18\gamma$ when invoked on G and r with the same parameters ρ and ε.*

Proof. If Oblivious-Verifier-Approximate-Clique accepts with G's sketch and randomness r, then, in particular, the execution of Oblivious-Verifier-Clique-Execution with the guesses that correspond to the run of Find-Approximate-Clique on G and r results in U such that for some $U' \subseteq U$, $|R \cap \Gamma(U')| \geq (1-\gamma)\rho\,|V|$, it holds that $\tilde{bias}^R_{U'} \geq 1 - 3\varepsilon - 9\gamma$. By Corollary 6.8, $bias_{U'} \geq 1 - 3\varepsilon - 18\gamma$. □

Next we argue that Oblivious-Verifier-Approximate-Clique rejects with probability exponentially small in $|V|$.

[7] Note that we argue about a version of Find-Approximate-Clique that checks a relaxed condition on the density such that the condition is satisfied by a coin of bias $1 - 3\varepsilon - 18\gamma$ (as opposed to the version of Figure 8).

> $\textsc{Oblivious-Verifier-Approximate-Clique}(G_R = (R, V, E_R), \rho, \varepsilon, r)$
> 1 Let $\mathcal{G}$ contain all the graphs that are consistent with G_R.
> 2 Try all guesses in $\textsc{Oblivious-Verifier-Clique-Execution}(\mathcal{G}, G_R, \rho, \varepsilon, r, 0)$.
> 3 Accept iff all feasible executions accept.

Figure 11: The final oblivious verifier for approximate clique.

Lemma 6.11. *The following hold:*

- *For all sequences of guesses, the probability that* Oblivious-Verifier-Clique-Execution *rejects is exponentially small in* $|V|$.
- Oblivious-Verifier-Clique-Execution *makes a number of guesses that is sufficiently smaller than* $|V|$ *(the ratio between the number of guesses and* $|V|$ *can be made arbitrarily small by lowering* ε *by a constant factor and by increasing* i_0 *by a constant).*

The correctness of the final oblivious verifier follows from a union bound over all possible choices of guesses. If the number of guesses is sufficiently smaller than $|V|$, the probability that Oblivious-Verifier-Approximate-Clique rejects is exponentially small in $|V|$.

In order to show the two items above we show that three invariants hold. The invariants are in Lemmas 6.12, 6.13 and 6.14.

Lemma 6.12. *Throughout the execution of* Oblivious-Verifier-Clique-Execution*:* $\mathcal{G}$ *contains all the graphs that are consistent with the sketch and guesses so far.*

Proof. The invariant holds since $\mathcal{G}$ is initialized to contain all graphs consistent with the sketch, and since after each guess $\mathcal{G}$ is updated to contain only those graphs in $\mathcal{G}$ that are consistent with the guess. □

Lemma 6.13. *Throughout the execution of* Oblivious-Verifier-Clique-Execution*: If the counter reaches a value c, then the sequence of results of guesses made by* Oblivious-Verifier-Clique-Execution *has probability exponentially small in c.*

Proof. counter is initially 0. We will show that every increase of the counter by some x happens right after an if-statement whose probability if being true is exponentially small in x.Z Note that there are two steps in which the counter increases (step 7 and step 14) we analyze them one at a time below.

The increase in step 7 is justified as follows. When this step occurs, $\tilde{bias}^R_{U'} < 1 - 2\varepsilon - \beta/2 + 9\gamma$ for all U' such that $|R \cap \Gamma(U')| \geq (1-\gamma)\rho\,|V|$. By Corollary 6.8, for all U' such that $|\Gamma(U')| \geq (1-2\gamma)\rho\,|V|$, we have $\left|\tilde{bias}^R_{U'} - bias_U\right| \leq 9\gamma$. Hence, when step 7 occurs for all U' with $|\Gamma(U')| \geq \rho\,|V|$, it holds that $bias_{U'} < 1 - 2\varepsilon$ (here we also use Lemma 6.6). By Lemma 6.3, for every graph in $\mathcal{G}$, the probability that this happens is $e^{-25/\varepsilon}$, and therefore the increase in *counter* is justified. The increase in Step 14 follows from the correctness of Oblivious-Verifier-Clique-Coin-Toss (Note that we take a union bound over 2^u possible U'). □

Lemma 6.14. *If* Oblivious-Verifier-Clique-Execution *restarts (that is, the if-statement on Step 16 evaluates to true) in phase i then counter increases by at least $\beta^2 2^{i-1} - u$ either in the restart phase or in the previous phase, or it is the first phase and counter increases by $25/\varepsilon$.*

Proof. Suppose that there is a restart in phase i. Let $k = 2^i$. By Lemma 4.7, either in this iteration, or in the previous iteration, there exists a non-faulty coin in the group that yielded a fraction of heads that deviates from its bias by an additive $\beta/2$, or it's the first phase and a group with bias at most $1 - 2\varepsilon - \beta/2$ was picked. Next we handle each of these cases.

1. Suppose that $i = i_0$ and $\max\{bias_{U'} \mid |\Gamma(U')| \geq \rho |V|\} < 1 - 2\varepsilon - \beta/2$. By Lemma 6.6, $|R \cap \Gamma(U')| \geq (1-\gamma)\rho |R|$, and, hence, by Corollary 6.8, $\left|\tilde{bias}^R_{U'} - bias_{U'}\right| \leq 9\gamma$. This implies that $\tilde{bias}^R_{U'} < 1 - 2\varepsilon - \beta/2 + 9\gamma$ and that *counter* increases in step 7.
2. Suppose that there exists $U' \subseteq U$ with $|\Gamma(U')| \geq \rho |V|$ such that $bias^{V'}_{U'}$ deviates from $bias_{U'}$ by at least $\beta/2$ in phase i or $i-1$. By Lemma 6.9 (and Lemma 6.6), since $29\gamma < \beta/2$ and Clique-Coin-Toss does not fail on U', we know that Clique-Coin-Toss rejects and that *counter* increases in Step 14.

The lemma follows. □

Next we prove Lemma 6.11 from the invariants in Lemmas 6.12, 6.13 and 6.14.

Proof. (of Lemma 6.11 from invariants) Rejection is caused either by *counter* reaching $|V|$, which happens with probability exponentially small in $|V|$ by Lemma 6.13, or by Find-Approximate-Clique, running on any of the graphs in $\mathcal{G}$ (recall that $\mathcal{G} \neq \phi$), returning, by Corollary 6.8, U' such that $bias_{U'} < 1 - 3\varepsilon$. We already saw that the latter has probability exponentially small in $|V|$ for any graph in $\mathcal{G}$ in Section 6.2.

The second item follows from Lemma 6.14 and since *counter* increases by large increments and *counter* $\leq |V|$. The increments are either at least $\beta^2 2^{i_0 - 1} - u$ (which corresponds to the constant term in i_0) or $25/\varepsilon$. Note also that no more than i_f "false" guesses can occur in a row – otherwise the for-loop of Step 8 terminates and so the execution for this set of guesses terminates as well. □

A non-uniform deterministic algorithm for approximate clique follows, concluding the proof of Theorem 1.2 (note that ε in Theorem 1.2 is replaced with $O(\varepsilon/\rho)$ here).

Theorem 6.15. *There is a deterministic non-uniform algorithm that given $0 < \rho, \varepsilon < 1$ and a graph $G = (V, E)$ with a clique on $\rho |V|$ vertices, finds a set of $\rho |V|$ vertices and density $1 - O(\varepsilon/\rho)$. The algorithm runs in time $\tilde{O}(|V|^2\, 2^{O(1/(\varepsilon^2 \rho))})$.*

Remark 6.2. *There is a an alternative way to prove Theorem 6.15 based on the biased coin algorithm. It involves a more complicated randomized algorithm that can achieve lower error probability and no sketching. First we devise a coin tossing algorithm that achieves error probability exponentially small in k for any $1 \leq k \leq |V|$ while running in time $O(k\,|U'|\,\mathrm{poly}(1/\rho, 1/\gamma))$. This algorithm picks $V' \subseteq V$ like* Clique-Coin-Toss*, but then instead of computing $bias^{V'}_{U'}$ directly, it estimates $bias^{V'}_{U'}$ by picking for every $v \in V'$ a small independent sample to estimate the fraction of $\Gamma(U')$ vertices that neighbor it. The observation is that it suffices that the estimates for most $v \in V'$ are accurate in order for the estimate for $bias^{V'}_{U'}$ to be accurate. By repeating this coin tossing algorithm r times, we can obtain a coin tossing algorithm that runs in time $O(kr\,|U'|\,\mathrm{poly}(1/\rho, 1/\gamma))$*

and achieves error probability exponentially small in kr for any $r \geq 1$. Via the biased coin algorithm, we obtain a Las Vegas algorithm that runs in time $\tilde{O}(|V|^2 2^{O(1/(\varepsilon^2\rho))})$ and achieves error probability exponentially small in $|V|^2$. This implies a deterministic non-uniform algorithm that runs in the same time.

7 Free Games

7.1 A Simple Randomized Algorithm

First we describe a simple randomized algorithm with constant error probability for free games based on the sampling idea in the MAX-CUT algorithm. A similar algorithm appeared in [1]. The main idea of the algorithm is as follows. We sample a small $S \subseteq X$ and enumerate over all possible labelings $h : S \to \Sigma$. Each labeling induces a labeling to all vertices as follows.

Definition 7.1 (Induced labeling)**.** Let $\mathcal{G}$ be a free game on a graph $G = (X, Y, X \times Y)$, alphabet Σ and constraints $\{\pi_e\}$. Let $S \subseteq X$ and let $h : S \to \Sigma$ be a labeling to S.

- The induced labeling $f_{S,h,Y} : Y \to \Sigma$ is defined as follows: For every $y \in Y$ let $f_{S,h,Y}(y)$ be the label $\sigma \in \Sigma$ that maximizes the fraction of edges $e = (s, y) \in S \times \{y\}$ such that $(h(s), \sigma) \in \pi_e$ (ties are broken arbitrarily).
- The induced labeling $f_{S,h,X} : X \to \Sigma$ is defined as follows: For every $x \in X$ let $f_{S,h,X}(x)$ be the label $\sigma \in \Sigma$ that maximizes the fraction of edges $e = (x, y) \in \{x\} \times Y$ such that $(\sigma, f_{S,h,Y}(y)) \in \pi_e$ (ties are broken arbitrarily).

Note that for each $y \in Y$ computing $f_{S,h,Y}(y)$ takes time $O(|\Sigma|\,|S|)$. For each $x \in X$ computing $f_{S,h,X}(x)$ takes time $O(|\Sigma|\,|Y|)$.

We will argue below that if h is the restriction of an optimal assignment to $\mathcal{G}$, then the induced labeling is likely to approximately achieve the value of $\mathcal{G}$.

Lemma 7.2 (Sampling)**.** *Let $\mathcal{G}$ be a free game on a graph $G = (X, Y, X \times Y)$ and with alphabet Σ. Let $\varepsilon, \delta > 0$. Then for a uniform $S \subseteq X$, $|S| = \lceil \log(|\Sigma|/\varepsilon\delta)/\varepsilon^2 \rceil$, with probability at least $1 - \delta$, there exists $h : S \to \Sigma$ such that $f_{S,h,X}$, $f_{S,h,Y}$ satisfy at least $val(\mathcal{G}) - 2\varepsilon$ fraction of the edges in G.*

Proof. There is a labeling $f_X^* : X \to \Sigma$, $f_Y^* : Y \to \Sigma$ that achieves the value of $\mathcal{G}$, namely, $val_{f_X^*, f_Y^*}(\mathcal{G}) = val(\mathcal{G})$. Let $h : S \to \Sigma$ be the restriction of f_X^* to S. Let $y \in Y$. Let $\sigma \in \Sigma$. By a Chernoff bound except with probability $\varepsilon\delta/|\Sigma|$, the fraction of edges $e = (s, y) \in S \times \{y\}$ such that $(h(s), \sigma) \in \pi_e$, is the same up to an additive ε as the fraction of edges $e = (x, y) \in X \times \{y\}$ such that $(f_X^*(x), \sigma) \in \pi_e$. By a union bound over all $\sigma \in \Sigma$, for each $y \in Y$, except with probability at most $\varepsilon\delta$ over S, this holds for all $\sigma \in \Sigma$. In other words, for a uniform S, the expected fraction of $y \in Y$ for which this does not hold is at most $\varepsilon\delta$. Thus, with probability at most δ over the choice of S, for at least $1 - \varepsilon$ fraction of the $y \in Y$, this holds. Therefore, except for at most δ fraction of the S, we have $val_{f_X^*, f_{S,h,Y}}(\mathcal{G}) \geq val_{f_X^*, f_Y^*}(\mathcal{G}) - 2\varepsilon = val(\mathcal{G}) - 2\varepsilon$. The lemma follows noticing that $val_{f_{S,h,X}, f_{S,h,Y}}(\mathcal{G}) \geq val_{f_X^*, f_{S,h,Y}}(\mathcal{G})$. □

7.2 A Randomized Algorithm With Exponentially Small Error Probability

We think of sampling $S \subseteq X$ as picking a group of coins. The group has a coin per $h : S \to \Sigma$. The bias of the coin is the fraction of edges satisfied by $f_{S,h,X}$ and $f_{S,h,Y}$. One tosses a coin k times by picking roughly k vertices $X' \subseteq X$ and estimating the success of $f_{S,h,X}$ and $f_{S,h,Y}$ on edges that touch X'. The coin tossing algorithm is described in Figure 12. For a deviation parameter γ dictating by how much the coin toss deviates from the actual bias and for k dictating the error probability, the toss runs in time $k\,|Y| \cdot \mathrm{poly}(|\Sigma|, 1/\varepsilon, 1/\gamma)$.

Free-Toss-Coin($\mathcal{G}, S, h, k, \gamma$)

Compute $f_{S,h,Y}(y)$ for all $y \in Y$.
Pick $X' \subseteq X$, $|X'| = \lceil (k + |S| \log |\Sigma|)/\gamma^2 \rceil$, uniformly at random.
For all $x \in X'$ and $\sigma \in \Sigma$ compute the fraction of $y \in Y$ so $(\sigma, f_{S,h,Y}(y)) \in \pi_{(x,y)}$.
For all $x \in X'$ let $g_{S,h,x}$ be the max over $\sigma \in \Sigma$ of the fractions.
return $bias_{S,h}^{X'} \doteq (1/|X'|) \sum_{x \in X'} g_{S,h,x}$ fraction heads.

Figure 12: A coin toss picks vertices at random and estimates how many of the edges that touch the sample are satisfied by $f_{S,h,X}$ and $f_{S,h,Y}$.

Let $bias_{S,h}$ be the fraction of edges satisfied by $f_{S,h,X}$ and $f_{S,h,Y}$. For $X' \subseteq X$, let $bias_{S,h}^{X'}$ be the fraction of edges that touch X' and are satisfied by $f_{S,h,X}$, $f_{S,h,Y}$. For each $x \in X$ let $g_{S,h,x}$ be the fraction of edges that touch x and are satisfied by $f_{S,h,X}$, $f_{S,h,Y}$. We have $bias_{S,h} = (1/|X|) \sum_{x \in X} g_{S,h,x}$, and $(1/|X'|) \sum_{x \in X'} g_{S,h,x} = bias_{S,h}^{X'}$. The following lemma shows that the estimate $bias_{S,h}^{X'}$ of the coin tossing algorithm typically doesn't deviate much from the actual bias $bias_{S,h}$.

Lemma 7.3. *Except with probability exponentially small in k, for all $h : S \to \Sigma$,*

$$\left| bias_{S,h}^{X'} - bias_{S,h} \right| \le \gamma.$$

Proof. By a Hoeffding bound, except with probability exponentially small in $k + |S| \log |\Sigma|$ we have,

$$\left| \frac{1}{|X'|} \sum_{x \in X'} g_{S,h,x} - bias_{S,h} \right| \le \gamma.$$

The lemma follows from a union bound over all $h : S \to \Sigma$.

□

Using an algorithm for the biased coin problem we get an algorithm for finding a good labeling to a free game. The algorithm is described in Figure 13.

The algorithm proves part of Theorem 1.3 repeated here for convenience.

Theorem 7.4. *Given a free game $\mathcal{G}$ with vertex sets X, Y, alphabet Σ, and $val(\mathcal{G}) \ge 1 - \varepsilon_0$ and given $\varepsilon > 0$, the algorithm* Find-Labeling *finds a labeling for $\mathcal{G}$ that achieves value at least $1 - \varepsilon_0 - 3\varepsilon$ except with probability exponentially small in $|X|\,|\Sigma|$. The algorithm runs in time $\tilde{O}(|X|\,|Y|\,|\Sigma|^{O((1/\varepsilon^2)\log(|\Sigma|/\varepsilon))})$.*

Find-Labeling($\mathcal{G} = (G = (X, Y, X \times Y), \Sigma, \{\pi_e\}), \varepsilon_0, \varepsilon$)

Set $s = \lceil \log(|\Sigma| / \varepsilon^2) / \varepsilon^2 \rceil$.
Set $i_f = \log((|X| |\Sigma| + s \log |\Sigma|)/\varepsilon^2) + \Theta(\log \log((|X| |\Sigma| + s \log |\Sigma|)/\varepsilon))$; $\beta = \varepsilon / i_f$.
Set $i_0 = \log(s \log |\Sigma| / \beta^2) + \Theta(1)$.
Sample $S \subseteq X$, $|S| = s$.
for $i = i_0, i_0 + 1, \ldots, i_f$
    Set $k = 2^i$.
    for all $h : S \to \Sigma$
        Free-Toss-Coin($\mathcal{G}, S, h, \beta^2 k, \gamma = \beta/80$).
    If the fraction of heads is less than $1 - \varepsilon_0 - 2\varepsilon - i\beta$ for all h, restart.
return labeling $f_{S,h,X}$, $f_{S,h,Y}$ with value at least $1 - \varepsilon_0 - 3\varepsilon$, if such exists.

Figure 13: An algorithm for finding a good labeling to a free game $\mathcal{G}$ with $val(\mathcal{G}) \geq 1 - \varepsilon_0$. The error probability of the algorithm is exponentially small in $|X| |\Sigma|$.

In the remainder of the section we construct an oblivious verifier for free games and prove the rest of Theorem 1.3, namely show a deterministic non-uniform algorithm.

7.3 A Sketch for Free Games

In this section we show that free games can be sketched using $\tilde{O}(|X| |\Sigma|^2 \operatorname{poly}(1/\varepsilon))$ bits (as opposed to $O(|X| |Y| |\Sigma|^2)$ bits needed to describe the entire input game). The idea is to store the sub-game $\mathcal{G}_R$ induced on a small and carefully chosen set $R \subseteq Y$, that is, store all the constraints of the form $\pi_{(x,y)}$ for $x \in X$ and $y \in R$. We will show that this allows us to estimate the bias of every coin, as well as the value of the labeling induced by the coin on the random samples the algorithm makes.

Let $x \in X$. Let $g^R_{S,h,x}$ be the maximum over $\sigma \in \Sigma$ of the fraction of vertices $y \in R$ such that $(\sigma, f_{S,h,Y}(y)) \in \pi_{(x,y)}$. We use the notation $bias^{X,R}_{S,h} \doteq (1/|X|) \sum_{x \in X} g^R_{S,h,x}$ and $bias^{X',R}_{S,h} \doteq (1/|X'|) \sum_{x \in X'} g^R_{S,h,x}$ for the bias of S, h as witnessed by R.

Lemma 7.5 (Free game sketch)**.** *Let s be as in Figure 13. Then, there exists $R \subseteq Y$, $|R| = \lceil s(s+1) \log |\Sigma| \log |X| / \gamma^2 \rceil$, such that for all $S \subseteq X$, $|S| = s$, for all $h : S \to \Sigma$, for every $x \in X$ we have*

$$\left| g^R_{S,h,x} - g_{S,h,x} \right| \leq \gamma.$$

As a result, $\left| bias^{X,R}_{S,h} - bias_{S,h} \right| \leq \gamma$, *and, for all* $X' \subseteq X$, $\left| bias^{X',R}_{S,h} - bias^{X'}_{S,h} \right| \leq \gamma$.

Proof. Pick uniformly at random $R \subseteq Y$ of the specified size. Let $S \subseteq X$, $|S| = s$, $h : S \to \Sigma$, $x \in X$. By applying a Chernoff bound for every $\sigma \in \Sigma$ and taking a union bound over $\sigma \in \Sigma$, we get that $\left| g^R_{S,h,x} - g_{S,h,x} \right| \leq \gamma$ except with probability smaller than $|\Sigma|^{-s} |X|^{-(s+1)}$. By a union bound over all S, h and x, we get that there exists $R \subseteq Y$ of the specified size such that $\left| g^R_{S,h,x} - g_{S,h,x} \right| \leq \gamma$ always holds. The claims about bias follow. □

7.4 Oblivious Verifier for Free Games

We design an oblivious verifier for Find Labeling, which we call Oblivious-Verifier-Free-Execution. The verifier gets the sketch of the input and the randomness of the algorithm Find Labeling, and follows the execution of the algorithm by guessing when it decides to restart. The final verifier, Oblivious-Verifier-Free-Game, checks all possible guesses. During its run Oblivious-Verifier-Free-Execution maintains *counter* recording the low probability events it witnessed. If *counter* ever reaches a value larger than $|X|\,|\Sigma|$, the verifier rejects. The verifier also maintains $\mathcal{H}$ the family of all free games consistent with the execution so far. If $\mathcal{H}$ becomes empty, the execution is designated as *infeasible*. Initially, *counter* $= 0$ and $\mathcal{H}$ contains all free games consistent with the sketch. The final verifier checks that, no matter what were the guesses, all feasible executions of Oblivious-Verifier-Free-Execution accept. We argue that since the algorithm has error probability that is exponentially small in $|X|\,|\Sigma|$, the probability that the final verifier rejects is exponentially small in $|X|\,|\Sigma|$ as well.

The verifier for a single execution (a single set of guesses) is described in Figure 14. Note that it uses the parameters i_0, i_f, β of the algorithm. The final verifier is described in Figure 15.

Next we analyze the oblivious verifier. We start by showing that when it accepts, Find-Labeling finds a high quality labeling (even if slightly of lower quality than in Figure 13).

Lemma 7.6. *If* Oblivious-Verifier-Free-Game *accepts on the sketch of a game* $\mathcal{G}$ *and on randomness* r*, then* Find-Labeling[8] *produces a labeling that satisfies at least* $1 - \varepsilon_0 - 3\varepsilon - 2\gamma$ *fraction of the edges when invoked on* $\mathcal{G}$ *and* r *and with the same parameters* $\varepsilon_0, \varepsilon$.

Proof. Let $\mathcal{G}$ be the input game to Find-Labeling. If Oblivious-Verifier-Free-Game accepts with the sketch $\mathcal{G}_R$, then, in particular, the execution of Oblivious-Verifier-Free-Execution with the guesses that correspond to the run of Find-Labeling accepts. By Lemma 7.5, Find-Labeling finds a labeling that satisfies $1 - \varepsilon_0 - 3\varepsilon - 2\gamma$ fraction of the edges. □

In order to argue that Oblivious-Verifier-Free-Game rejects with probability exponentially small in $|X|\,|\Sigma|$ we prove the following.

Lemma 7.7. *The following hold:*

- *For any guesses, the probability that* Oblivious-Verifier-Free-Execution *rejects is exponentially small in* $|X|\,|\Sigma|$.
- Oblivious-Verifier-Free-Execution *makes a number of guesses that is sufficiently smaller than* $|X|\,|\Sigma|$ *(the ration between the number of guesses and* $|X|\,|\Sigma|$ *can be made arbitrarily small by multiplying* ε *by a suitable constant and by increasing the constant term of* i_0*).*

The correctness of the final verifier follows from a union bound over all possible guesses. If the number of guesses is sufficiently small, then there is an exponentially small probability in $|X|\,|\Sigma|$ that the final verifier rejects.

In order to prove Lemma 7.7, we show the following invariants.

Lemma 7.8. *The following invariants are maintained throughout the run of* Oblivious-Verifier-Free-Execution*:*

[8] Note that we argue about a version of Find-Labeling that checks a relaxed condition that is satisfied by a coin of bias $1 - \varepsilon_0 - 3\varepsilon - 2\gamma$ (as opposed to the version of Figure 13).

Oblivious-Verifier-Free-Execution($\mathcal{H}, \mathcal{G}_R, \varepsilon_0, \varepsilon, r, counter$)

1 **if** $\mathcal{H} = \phi$
//$\mathcal{H}$ is the set of possible inputs consistent with the current execution.
2 **return** infeasible.
3 **if** $counter > |X|\,|\Sigma|$
4 **return** reject.
5 Extract from r the sample $S \subseteq X$ of the algorithm.
//The randomness r encodes which subset S is sampled.
6 **if** $\max_h bias_{S,h}^{X,R} < 1 - \varepsilon_0 - 3\varepsilon - \beta/2 + \gamma$
//If our approximation of the bias is too low, we increase the counter to account for this low probability event
7 $counter \leftarrow counter + \log(1/\varepsilon)$
8 **for** $i = i_0, i_0 + 1, \ldots, i_f$
9 $k \leftarrow 2^i$.
10 **for** all $h : S \to \Sigma$
11 Extract from r the sample $X' \subseteq X$ of the algorithm.
12 Compute $bias_{S,h}^{X',R}$.
13 **if** $\exists h, \left| bias_{S,h}^{X',R} - bias_{S,h}^{X,R} \right| > 3\gamma$
14 $counter \leftarrow counter + \beta^2 k - s \log |\Sigma|$.
//We maintain that the probability of this execution is exponentially small in the counter.
15 Guess if $\max_h bias_{S,h}^{X'} < 1 - \varepsilon_0 - 2\varepsilon - i\beta$ and update $\mathcal{H}$ accordingly.
16 **if** guessed true
17 Restart while maintaining $\mathcal{H}$ and the value of $counter$.
18 **return** accept iff $\max_h bias_{S,h}^{X,R} \geq 1 - \varepsilon_0 - 3\varepsilon - \gamma$.

Figure 14: An oblivious verifier for a single execution of Find-Labeling (an execution is defined by the outcomes of guesses). The final oblivious verifier checks that all feasible executions are accepted.

1. *$\mathcal{H}$ consists of all the free games that are consistent with the sketch and the guesses so far.*
2. *If $counter = c$ then the sequence of guesses made by* Oblivious-Verifier-Free-Execution *has probability exponentially small in c.*
3. *If* Oblivious-Verifier-Free-Execution *restarts in phase i, then either in the current phase or in the previous counter increases by at least $\beta^2 2^{i-1} - s \log |\Sigma|$, or $i = i_0$ and counter increases by $\log(1/\varepsilon)$.*

Proof. It's clear that Invariant 1 holds. Next we show that Invariant 2 holds. Initially *counter* is set to 0. The increase in Step 7 is justified by Lemma 7.2 and the design of the sketch (Lemma 7.5). The increase in Step 14 is justified by Lemma 7.3 and the design of the sketch (Lemma 7.5).

Next we show that Invariant 3 holds. Suppose that there is a restart in phase i. Let $k = 2^i$. Lemma 4.7 ensures that either in the current phase or in the previous one $\max_h bias_{S,h}^{X'}$ deviates

OBLIVIOUS-VERIFIER-FREE-GAME($\mathcal{G}_R, \varepsilon_0, \varepsilon, randomness$)
1 Let $\mathcal{H}$ contain all the free games that are consistent with $\mathcal{G}_R$.
2 Try all guesses in OBLIVIOUS-VERIFIER-FREE-EXECUTION($\mathcal{H}, \mathcal{G}_R, \varepsilon_0, \varepsilon, randomness, 0$).
3 Accept iff all feasible executions accept.

Figure 15: The final oblivious verifier for free games.

from $\max_h bias_{S,h}$ by more than an additive $\beta/2$, or it's the first phase and $\max_h bias_{S,h}$ is smaller than $1 - \varepsilon_0 - 2\varepsilon - \beta/2$. We handle both cases:

1. Suppose that this is the first phase and $\max_h bias_{S,h} < 1 - \varepsilon_0 - 2\varepsilon - \beta/2$. By the design of the sketch (Lemma 7.5), we have $\max_h bias^{X,R}_{S,h} < 1 - \varepsilon_0 - 2\varepsilon - \beta/2 + \gamma$, and hence *counter* increases as required in Step 7.

2. Suppose that either in the current phase or in the previous one $\max_h bias^{X'}_{S,h}$ deviates from $\max_h bias_{S,h}$ by more than an additive $\beta/2$. By the design of the sketch (Lemma 7.5), either in this phase or in the previous there exists $h : S \to \Sigma$ such that $\left| bias^{X',R}_{S,h} - bias^{X,R}_{S,h} \right| > 3\gamma$. In this case, *counter* increases appropriately in Step 14 of the appropriate phase.

□

We can now prove Lemma 7.7 from the invariants of Lemma 7.8.

Proof. (of Lemma 7.7 from Lemma 7.8) Let us show that the first item in Lemma 7.7 follows from Invariant 2. The verifier only rejects if $counter > |X|\,|\Sigma|$, or if it reached Step 18 and rejected. The invariant ensures that the probability that $counter > |X|\,|\Sigma|$ is exponentially small in $|X|\,|\Sigma|$. Theorem 7.4 ensures that the probability that $\max_h bias_{S,h} < 1 - \varepsilon_0 - 3\varepsilon$ is exponentially small in $|X|\,|\Sigma|$. The design of the sketch (Lemma 7.5) ensures that if $\max_h bias_{S,h} \geq 1 - \varepsilon_0 - 3\varepsilon$, then $\max_h bias^{X,R}_{S,h} \geq 1 - \varepsilon_0 - 3\varepsilon - \gamma$.

The second item follows from Invariant 3, since $counter < |X|\,|\Sigma|$, and, *counter* increases in large increments: Invariant 6.14 ensures that the increments only depend on ε and i_0, and can be made arbitrarily large by multiplying ε by a small constant and by adding to i_0 a large constant. □

A non-uniform deterministic algorithm for free games follows, concluding the proof of Theorem 1.3.

Theorem 7.9. *There is a deterministic non-uniform algorithm that given a free game $\mathcal{G}$ with vertex sets X, Y, alphabet Σ and $val(\mathcal{G}) \geq 1 - \varepsilon_0$, finds a labeling to the vertices that satisfies $1 - \varepsilon_0 - O(\varepsilon)$ fraction of the edges. The algorithm runs in time $\tilde{O}(|X|\,|Y|\,|\Sigma|^{O((1/\varepsilon^2)\log(|\Sigma|/\varepsilon))})$.*

We remark that there is an alternative way to prove Theorem 7.9 similarly to Remark 6.2.

8 From List Decoding to Unique Decoding of Reed-Muller Code

8.1 A Randomized Algorithm With Error Probability Roughly $1/|\mathbb{F}|$

Let $\mathbb{F}$ be a finite field, and let $m > 3$ and $d < |\mathbb{F}|$ be natural numbers. First we describe a randomized algorithm with error probability $|\mathbb{F}|^{-\Omega(1)}$ for reducing the Reed-Muller list decoding problem with parameters $\mathbb{F}$, m, d to the Reed-Muller unique decoding problem. The algorithm is based on the idea of self-correction (see, e.g., [43] and the references there). The algorithm is described in Figure 16. On input $f : \mathbb{F}^m \to \mathbb{F}$ it picks a random line ℓ and finds all the polynomials that agree with f on about ρ fraction of the points in ℓ. We'll show that if f agrees with an m-variate polynomial p on ρ fraction of the points in $\mathbb{F}^m$, then, except with probability roughly $1/|\mathbb{F}|$ over the choice of ℓ, there is about ρ fraction of the points on ℓ on which f agrees with p. Hence, the restriction of p to ℓ is likely to be one of the polynomials that the algorithm finds. The algorithm outputs a list of functions $g_1, \ldots, g_k : \mathbb{F}^m \to \mathbb{F}$. Each g_i corresponds to one of the polynomials in the line list. The algorithm computes each g_i by iterating over all $z \in \mathbb{F}^m$ and considering the plane s spanned by ℓ and z. Again, except for fraction roughly $1/|\mathbb{F}|$ of the $z \in \mathbb{F}^m$, there is about ρ fraction of the points on s on which f agrees with p. The algorithm sets $g_i(z) = f(z)$ if there is a unique polynomial that agrees with f on about ρ fraction of the points in s and with g_i's polynomial on ℓ.

SELF-CORRECT(f, ρ, ϵ)

1 Pick uniformly at random $x, y \in \mathbb{F}^m$, $y \neq \vec{0}$.
2 Find all univariate $p^{(1)}_{x,y}, \ldots, p^{(k)}_{x,y}$ so $\left|\left\{t \in \mathbb{F} \mid f(x+ty) = p^{(j)}_{x,y}(t)\right\}\right| \geq (\rho - \epsilon) \cdot |\mathbb{F}|$.
3 **for** $z \in \mathbb{F}^m$ such that $z - x, y$ are independent
4   Find $q^{(1)}, \ldots, q^{(k')}$: $\left|\left\{t_1, t_2 \in \mathbb{F} \mid f(x + t_1 y + t_2(z-x)) = q^{(j)}(t_1, t_2)\right\}\right| \geq (\rho - \epsilon) \cdot |\mathbb{F}|^2$.
5   **for** $1 \leq i \leq k$
6   **if** $\exists!\ 1 \leq j \leq k'$, $p^{(i)}_{x,y}(t) = q^{(j)}(t, 0)$ for all $t \in \mathbb{F}$
7   Set $g_i(z) = q^{(j)}(0, 1)$.
8 **return** $g_1, \ldots, g_k$.

Figure 16: A randomized algorithm with error probability $|\mathbb{F}|^{-\Omega(1)}$ that finds $g_1, \ldots, g_k : \mathbb{F}^m \to \mathbb{F}$, $k \leq O(1/\rho)$, such that for every polynomial p of degree at most d that agrees with f on ρ fraction of the points in $\mathbb{F}^m$ there is g_i that agrees with p on at least $1 - \epsilon$ fraction of the points.

Steps 2 and 4 that require list decoding of Reed-Solomon code can be performed in time $\text{poly}(|\mathbb{F}|)$. Therefore, the run time of the algorithm is $O(|\mathbb{F}^m| \,\text{poly}(|\mathbb{F}|))$. A standard choice of parameters is $|\mathbb{F}| = \text{poly}\log|\mathbb{F}^m|$, and it leads to a run-time of $\tilde{O}(|\mathbb{F}^m|)$. Next we prove the correctness of the algorithm.

We'll need the following lemma about list decoding for polynomials.

Lemma 8.1 (List decoding)**.** *Fix a finite field $\mathbb{F}$ and natural numbers m and $d < |\mathbb{F}|$. Let $f : \mathbb{F}^m \to \mathbb{F}$. Then, for any $\rho \geq 2\sqrt{\frac{d}{|\mathbb{F}|}}$, if $q_1, \ldots, q_k : \mathbb{F}^m \to \mathbb{F}$ are* different *polynomials of degree at most d, and for every $1 \leq i \leq k$, the polynomial q_i agrees with f on at least ρ fraction of the points, i.e., $\Pr_{x \in \mathbb{F}^m}[q_i(x) = f(x)] \geq \rho$, then $k \leq \frac{2}{\rho}$.*

Proof. Let $\rho \geq 2\sqrt{\frac{d}{|\mathbb{F}|}}$, and assume on way of contradiction that there exist $k = \lfloor \frac{2}{\rho} \rfloor + 1$ different polynomials $q_1, \ldots, q_k : \mathbb{F}^m \to \mathbb{F}$ as stated.

For every $1 \leq i \leq k$, let $A_i \doteq \{x \in \mathbb{F}^m \mid q_i(x) = f(x)\}$. By inclusion-exclusion,

$$|\mathbb{F}^m| \geq \left|\bigcup_{i=1}^{k} A_i\right| \geq \sum_{i=1}^{k} |A_i| - \sum_{i \neq j} |A_i \cap A_j|$$

By Schwartz-Zippel, for every $1 \leq i \neq j \leq k$, $|A_i \cap A_j| \leq \frac{d}{|\mathbb{F}|} \cdot |\mathbb{F}^m|$. Therefore, by the premise,

$$|\mathbb{F}^m| \geq k\rho\,|\mathbb{F}^m| - \binom{k}{2}\frac{d}{|\mathbb{F}|}\,|\mathbb{F}^m|$$

On one hand, since $k > \frac{2}{\rho}$, we get $k\rho > 2$. On the other hand, since $\frac{2}{\rho} \leq \sqrt{\frac{|\mathbb{F}|}{d}}$ and $d \leq |\mathbb{F}|$, we get $\binom{k}{2} \leq \frac{|\mathbb{F}|}{d}$. This results in a contradiction. □

Let p be an m-variate polynomial of degree at most d over $\mathbb{F}$ that agrees with f on at least ρ fraction of the points $x \in \mathbb{F}^m$. We will show that most likely one of the g_i's that the algorithm outputs is very close to p.

In the following lemma we argue that the restriction of p to the line defined by x and y is likely to appear in the line list.

Lemma 8.2 (Sampling)**.** *Except with probability $\rho/(\epsilon^2\,|\mathbb{F}|)$ over the choice of x and y, for at least $(\rho - \epsilon)\,|\mathbb{F}|$ elements $t \in \mathbb{F}$ we have $f(x + ty) = p(x + ty)$.*

Proof. The lemma follows from a second moment argument. Let $A \subseteq \mathbb{F}^m$, $|A| = \rho\,|\mathbb{F}^m|$ contain elements $z \in \mathbb{F}^m$ such that $f(z) = p(z)$. For uniform $x, y \in \mathbb{F}^m$, for $t \in \mathbb{F}$ let X_t be an indicator random variable for $x + ty \in A$. Let $X = (1/\,|\mathbb{F}|)\sum X_t$. We have $\mathbf{E}[X] = \rho$. For every $t \neq t' \in \mathbb{F}$ we have that X_t and $X_{t'}$ are independent. Hence, $\mathbf{E}\left[X^2\right] = (1/\,|\mathbb{F}|)^2 \sum_{t,t'} \mathbf{E}[X_t X_{t'}] = (1/\,|\mathbb{F}|)^2 \sum_t \mathbf{E}[X_t] = \rho/\,|\mathbb{F}|$. By Chebychev inequality,

$$\Pr[|X - \rho| > \epsilon] \leq \frac{\mathbf{E}\left[X^2\right]}{\epsilon^2} = \frac{\rho}{\epsilon^2\,|\mathbb{F}|}.$$

The lemma follows. □

The same holds for the planes defined by most $z \in \mathbb{F}^m$.

Lemma 8.3 (Sampling)**.** *Except with probability $\rho/(\epsilon^2\,|\mathbb{F}|)$ over the choice of z, x and y, for at least $(\rho - \epsilon)\,|\mathbb{F}|^2$ elements $t_1, t_2 \in \mathbb{F}$, we have $f(x + t_1 y + t_2(z - x)) = p(x + t_1 y + t_2(z - x))$.*

From Lemmas 8.2 and 8.3 it follows that except with probability roughly $1/\,|\mathbb{F}|$ restrictions of p appear both in the line list and in most planes lists. Next we'll argue that for most $z \in \mathbb{F}^m$ it's unlikey that the restriction of p to the plane is not the unique polynomial in the plane list that agrees with p on the line.

Lemma 8.4. *The probability over the choice of x, y and z that there are $1 \leq j < i \leq k'$ such that $q^{(j)}(t, 0) \equiv q^{(i)}(t, 0)$ even though $q^{(j)} \not\equiv q^{(i)}$, is at most $4d/((\rho - \epsilon)^2\,|\mathbb{F}|)$.*

Proof. Fix $1 \leq j < i \leq k'$. Suppose that one first picks the three dimensional subspace s and then picks $x, y, z \in \mathbb{F}^m$ such that $\{x + t_1 y + t_2(z - x) \mid t_1, t_2 \in \mathbb{F}\}$. The probability over the choice of x and y, that $q^{(j)}$ agrees with $q^{(i)}$ on the line $\{x + ty \mid t \in \mathbb{F}\}$ is at most $d / |\mathbb{F}|$. By Lemma 8.1, there are at most $2/(\rho - \epsilon)$ polynomials in the list of s. Taking a union bound over all choices of $1 \leq j < i \leq k'$ results in the lemma. □

Hence, except with probability $O(\delta)$ over x, y for $\delta = \max\left\{d/(\rho^2 |\mathbb{F}|), \rho/(\epsilon^2 |\mathbb{F}|)\right\}$, the restriction of p to the line $\{x + ty \mid t \in \mathbb{F}\}$ appears as $p^{(i)}_{x,y}$, and, moreover, g_i agrees with p on all but $O(\delta)$ fraction of the z (note that only a small fraction $|\mathbb{F}|^2 / |\mathbb{F}^m|$ of the $z \in \mathbb{F}^m$ satisfy that $z - x, y$ are dependent).

8.2 Finding Approximate Codewords as Finding a Biased Coin

We describe an analogy between finding a list of approximate polynomials and finding a biased coin. We think of picking a line and finding a list decoding of f on the line as picking a coin. The coin picking algorithm is described in Figure 17. We think of sampling $z \in \mathbb{F}^m$ and checking whether the line list decoding is consistent with the list decoding on the subspace defined by z and the line as a coin toss that falls on "heads" if there is consistency. The coin tossing algorithm is described in Figure 18.

RM-Pick-Coin(f, ρ, ϵ)

Pick uniformly at random $x, y \in \mathbb{F}^m$, $y \neq \vec{0}$.
Find univariate polynomials $p^{(1)}_{x,y}, \ldots, p^{(k)}_{x,y}$ so $\left|\left\{t \in \mathbb{F} \mid f(x + ty) = p^{(j)}_{x,y}(t)\right\}\right| \geq (\rho - \epsilon) \cdot |\mathbb{F}|$.
return $x, y, p^{(1)}_{x,y}, \ldots, p^{(k)}_{x,y}$.

Figure 17: A coin corresponds to a line in $\mathbb{F}^m$ and the list decoding of f on the line.

RM-Toss-Coin$(f, \rho, \epsilon, x, y, p^{(1)}_{x,y}, \ldots, p^{(k)}_{x,y})$

Pick uniformly $z \in \mathbb{F}^m$ independent of x, y.
Find $q^{(1)}, \ldots, q^{(k')}$ so $|\{t_1, t_2 \in \mathbb{F} \mid f(x + t_1 y + t_2(z - x)) = q(t_1, t_2)\}| \geq (\rho - \epsilon) \cdot |\mathbb{F}|^2$.
for $1 \leq i \leq k$
    if $\neg\exists! 1 \leq j \leq k'$, $p^{(i)}_{x,y}(t) \equiv q^{(j)}(t, 0)$
        return "tails".
return "heads".

Figure 18: A coin toss corresponds to picking a uniform $z \in \mathbb{F}^m$ and checking whether the line list decoding is consistent with the list decoding on the subspace defined by the line and z.

Lemmas 8.2, 8.3 and 8.4 ensure that a biased coin is picked with at least a constant probability for sufficiently large $\rho > \epsilon > 0$ and sufficiently small $d < |\mathbb{F}|$. Note that both picking a coin and tossing it take short time poly$(|\mathbb{F}|)$. Hence, if we use $\tilde{O}(|\mathbb{F}^m|)$ coin tosses to find a biased coin and

RM-Interpolate$(f, \rho, \epsilon, x, y, p^{(1)}_{x,y}, \ldots, p^{(k)}_{x,y})$

for $z \in \mathbb{F}^m$ independent of x, y

  Find $q^{(1)}, \ldots, q^{(k')}$ so $|\{t_1, t_2 \in \mathbb{F} \mid f(x + t_1 y + t_2(z - x)) = q(t_1, t_2)\}| \geq (\rho - \epsilon) \cdot |\mathbb{F}|^2$.

  for $1 \leq i \leq k$

    if $\exists!\ 1 \leq j \leq k',\ p^{(i)}_{x,y}(t) = q^{(j)}(t, 0)$ for all $t \in \mathbb{F}$

      Set $g_i(z) = q^{(j)}(0, 1)$.

return $g_1, \ldots, g_k$.

Figure 19: An algorithm that uses a biased coin (given by $x, y, p^{(1)}_{x,y}, \ldots, p^{(k)}_{x,y}$) to find a short list of functions $g_1, \ldots, g_k : \mathbb{F}^m \to \mathbb{F}$ such that for every polynomial p of degree at most d that agrees with f on ρ fraction of the points in $\mathbb{F}^m$ there is g_i that agrees with p on at least $1 - \epsilon$ fraction of the points.

$|\mathbb{F}| = \text{poly}\log|\mathbb{F}^m|$, then we get an algorithm with $\tilde{O}(|\mathbb{F}^m|)$ run-time. Moreover, using a biased coin one can compute a short list of approximate polynomials as in Figure 19.

Lemma 8.5. *Assume that $x, y, p^{(1)}_{x,y}, \ldots, p^{(k)}_{x,y}$ define a biased coin (namely, a coin that falls on "heads" with probability at least $1 - O(\delta)$ for δ as in Section 8.1), and that ϵ is smaller than $O(\delta)$ from Section 8.1. Then, for every m-variate polynomial p of degree at most d over $\mathbb{F}$ there exists g_i in the list computed by* RM-Interpolate *that agrees with p on at least $1 - O(\delta)$ fraction of the points.*

Proof. Let p be an m-variate polynomial of degree at most d over $\mathbb{F}$ that agress with f on at least ρ fraction of the points $x \in \mathbb{F}^m$. Assume on way of contradiction that none of $p^{(1)}_{x,y}, \ldots, p^{(k)}_{x,y}$ is p restricted to the line $\{x + ty \mid t \in \mathbb{F}\}$ (otherwise, we are done as we argued in Section 8.1). Take ϵ sufficiently smaller than $O(\delta)$ in the lemma. For at least ϵ fraction of the $z \in \mathbb{F}^m$ such that $z - x, y$ are independent, the fraction of points $x + t_1 y + t_2(z - x)$ with $t_1, t_2 \in \mathbb{F}$ on which f agrees with p, is at least $\rho - \epsilon$. For those choices of z, the coin falls on "tails", hence the bias of the coin is at most $1 - \epsilon$, which is a contradiction. □

Therefore, Find-Biased-Coin when using RM-Pick-Coin and RM-Toss-Coin, and when followed by RM-Interpolate to obtain the list of approximate polynomials from the coin, solves the list decoding to unique decoding problem for the Reed-Muller code in time $\tilde{O}(|\mathbb{F}^m| \,\text{poly}(|\mathbb{F}|))$ and with error probability exponentially small in $|\mathbb{F}^m| \log |\mathbb{F}|$. Since the input f, ρ, ϵ is of size $|\mathbb{F}^m| \log |\mathbb{F}|$, we also get a deterministic non-uniform algorithm that runs in similar time. For convenience, we repeat Theorem 1.5 that we just proved.

Theorem 8.6. *Let $\mathbb{F}$ be a finite field, let d and $m > 3$ be natural numbers and let $0 < \rho, \epsilon < 1$, such that $d \leq |\mathbb{F}|/10$, $\epsilon > \sqrt[3]{2/|\mathbb{F}|}$ and $\rho > \epsilon + 2\sqrt{d/|\mathbb{F}|}$. There is a randomized algorithm that given $f : \mathbb{F}^m \to \mathbb{F}$, finds a list of $l = O(1/\rho)$ functions $g_1, \ldots, g_l : \mathbb{F}^m \to \mathbb{F}$, such that for every m-variate polynomial p of degree at most d over $\mathbb{F}$ that agrees with f on at least ρ fraction of the points $x \in \mathbb{F}^m$, there exists g_i that agrees with p on at least $1 - \epsilon$ fraction of the points $x \in \mathbb{F}^m$. The algorithm has error probability exponentially small in $|\mathbb{F}^m| \log |\mathbb{F}|$ and it runs in time $\tilde{O}(|\mathbb{F}^m| \,\text{poly}(|\mathbb{F}|))$. It implies a deterministic non-uniform algorithm with the same run-time.*

9 Open Problems

- 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? The applications in this paper have Atlantic City algorithms that run in sub-linear time, but we do not think that the method is limited to such problems. It will be interesting to find concrete examples.
- 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?
- Are there deterministic algorithms for Max-Cut on dense graphs that run in time $\tilde{O}(|V|^2 + (1/\varepsilon)^{O(1/\gamma\varepsilon^2)})$ or even $O(|V|^2 + 2^{O(1/\gamma\varepsilon^2)})$ instead of $\tilde{O}(|V|^2\,(1/\varepsilon)^{O(1/\gamma\varepsilon^2)})$? Recall that the randomized algorithm of Mathieu and Schudy [36] runs in time $O(|V|^2 + 2^{O(1/\gamma^2\varepsilon^2)})$. Are there deterministic algorithms for (approximate) Clique that run in time $\tilde{O}(|V|^2 + 2^{O(1/(\rho^3\varepsilon^2))})$ instead of $\tilde{O}(|V|^2\, 2^{O(1/(\rho^3\varepsilon^2))})$?
- 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 [30]. Also, a result of Pettie and Ramachandran proves that a non-uniform linear time algorithm for MST would imply a uniform algorithm [39]. 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. Many thanks to Scott Aaronson, Noga Alon, Bernard Chazelle, Shiri Chechik, Shayan Oveis Gharan, Oded Goldreich, Kristoffer Arnsfelt Hansen, Russell Impagliazzo, David Karger, Kasper Green Larsen, Guy Moshkovitz, Michal Moshkovitz, Richard Peng, Seth Pettie, Vijaya Ramachandran, Aaron Sidford, Dan Spielman, Bob Tarjan, Virginia Vassilevska Williams, Avi Wigderson and Ryan Williams for discussions. Thanks to anonymous reviewers for helpful comments for improving the paper's exposition.

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] M. Alekhnovich. Linear diophantine equations over polynomials and soft decoding of Reed-Solomon codes. *IEEE Transactions on Information Theory*, 51(7):2257–2265, 2005.

[4] 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.

[5] 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.

[6] N. Alon, O. Goldreich, J. Håstad, and R. Peralta. Simple constructions of almost k-wise independent random variables. *Random Structures and Algorithms*, 3:289–304, 1992. Addendum: Random Structures and Algorithms 4:119-120, 1993.

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

[8] B. Barak, M. Hardt, T. Holenstein, and D. Steurer. Subsampling mathematical relaxations and average-case complexity. In *Proceedings of the Twenty-second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '11, pages 512–531, 2011.

[9] 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. Springer Berlin Heidelberg, 2005.

[10] M. Braverman, A. Rao, R. Raz, and A. Yehudayoff. Pseudorandom generators for regular branching programs. In *Proc. 51st IEEE Symp. on Foundations of Computer Science*, pages 40–47, 2010.

[11] W. Fernandez de la Vega. Max-cut has a randomized approximation scheme in dense graphs. *Random Struct. Algorithms*, 8(3):187–198, 1996.

[12] 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.

[13] 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.

[14] 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. Springer Berlin Heidelberg, 2002.

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

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

[17] J. Fox, L. M. Lovász, and Y. Zhao. On regularity lemmas and their algorithmic applications. *Combinatorics, Probability, and Computing*, 26:481–505, 2017.

[18] J. Fox, L. M. Lovász, and Y. Zhao. A fast new algorithm for weak graph regularity. *arXiv preprint arXiv:1801.05037*, 2018.

[19] 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.

[20] E. Gat and S. Goldwasser. Probabilistic search algorithms with unique answers and their cryptographic applications. *Electronic Colloquium on Computational Complexity (ECCC)*, 18:136, 2011.

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

[22] O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. *Journal of the ACM*, 45(4):653–750, 1998.

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

[24] V. Guruswami and M. Sudan. Improved decoding of Reed-Solomon and algebraic-geometry codes. *IEEE Transactions on information theory*, 45:1757–1767, 1999.

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

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

[27] 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.

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

[29] A. Joffe. On a set of almost deterministic k-independent random variables. *Annals of Probability*, 2(1):161–162, 1974.

[30] 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.

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

[32] S. Khot. On the power of unique 2-prover 1-round games. In *Proc. 34th ACM Symp. on Theory of Computing*, pages 767–775, 2002.

[33] 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.

[34] 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.

[35] P. Manurangsi and D. Moshkovitz. Approximating dense max 2-CSPs. In *APPROX-RANDOM*, 2015.

[36] C. Mathieu and W. Schudy. Yet another algorithm for dense max cut: go greedy. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*, pages 176–182, 2008.

[37] R. Meka and D. Zuckerman. Pseudorandom generators for polynomial threshold functions. *SIAM Journal on Computing*, 42(3):1275–1301, 2013.

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

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

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

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

[42] M. Sudan. Decoding of Reed Solomon codes beyond the error-correction bound. *Journal of Complexity*, 13(1):180–193, 1997.

[43] M. Sudan, L. Trevisan, and S. P. Vadhan. Pseudorandom generators without the XOR lemma. *J. Comput. Syst. Sci.*, 62(2):236–266, 2001.

[44] E. Szemerédi. Regular partitions of graphs. In *Problmes combinatoires et thorie des graphes (Colloq. Internat. CNRS, Univ. Orsay, Orsay, 1976)*, pages 399–401, 1978.

[45] E. Viola. The sum of D small-bias generators fools polynomials of degree D. *Computational Complexity*, 18(2):209–217, 2009.

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