Uniform Generation of NP-witnesses using an NP-oracle

Mihir Bellare* Oded Goldreich† Erez Petrank‡

May 1998

Abstract

A *Uniform Generation procedure* for $\mathcal{NP}$ is an algorithm which given any input in a fixed NP-language, outputs a uniformly distributed NP-witness for membership of the input in the language. We present a Uniform Generation procedure for $\mathcal{NP}$ that runs in probabilistic polynomial-time with an NP-oracle. This improves upon results of Jerrum, Valiant and Vazirani, which either require a Σ_2^{P} oracle or obtain only almost uniform generation. Our procedure utilizes ideas originating in the works of Sipser, Stockmeyer, and Jerrum, Valiant and Vazirani.

*Dept. of Computer Science & Engineering, University of California at San Diego, 9500 Gilman Drive, La Jolla, California 92093, USA. E-Mail: mihir@cs.ucsd.edu. URL: http://www-cse.ucsd.edu/users/mihir. Supported in part by NSF CAREER Award CCR-9624439 and a 1996 Packard Foundation Fellowship in Science and Engineering.

†Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel. E-Mail: oded@wisdom.weizmann.ac.il. Work done while being on a sabbatical leave at MIT.

‡IBM Haifa Research Lab, MATAM, Haifa 31905, Israel. E-Mail: erezp@haifa.vnet.ibm.com. URL: http://www.cs.princeton.edu/~Erez.

1 Introduction

Fix an $\mathcal{NP}$ language L and an $\mathcal{NP}$-relation R defining it. (Thus, $L = \{ x \mid \exists w \text{ such that } R(x,w) = 1 \}$, it being understood that R is polynomial time computable and $R(x,w)$ can equal 1 only if $|w| \leq p(|x|)$, for some fixed polynomial p.) We consider the following problem:

> **Uniform Generation of $\mathcal{NP}$-witnesses**
> GIVEN: $x \in L$
> OUTPUT: A string w uniformly distributed in $R_x \stackrel{\text{def}}{=} \{ w \mid R(x,w) = 1 \}$.[1]

This was first considered by Jerrum, Valiant and Vazirani [15], who showed that it could be accomplished in probabilistic polynomial time given access to a Σ_2^{P} oracle.

The same paper also considered a weaker version of the problem, called "almost uniform generation". Here, the requirement is that the output distribution of the algorithm be statistically close to the uniform distribution on R_x. (To be specific, let's say the distance should be exponentially small in $|x|$.) They showed that this could be accomplished more efficiently, in probabilistic, polynomial time given access to an $\mathcal{NP}$-oracle.

In this paper we provide a procedure that has the "best of both worlds:" it accomplishes uniform generation of $\mathcal{NP}$-witnesses in probabilistic polynomial time with access to an $\mathcal{NP}$ oracle.

Note it is not hard to see that any algorithm for uniform (or almost uniform) generation of $\mathcal{NP}$-witnesses must be probabilistic and must have at least $\mathcal{NP}$ power.

Jerrum et. al. [15] obtained their results by reducing uniform generation to the problem of approximate counting. The latter problem can be solved using the "hashing paradigm" introduced by Sipser [19] and employed in previous works on this problem [20, 15] (see Section 4.1 for a more complete description of the history.). In contrast, we directly apply the "hashing paradigm" to the problem of uniform generation, rather than utilizing the above reduction.

Our investigation of the process of uniform generation is to some extent the outcome of our uses for it. Over the last few years, there has been a body of work in the area of interactive proof systems and knowledge complexity that has exploited uniform generation to develop efficient decision procedures for certain languages based on their interactive proof or knowledge complexity [5, 11, 3, 10]. The complexity of the procedure is the crucial issue in these works, and it depends largely on the complexity of uniform generation. We survey these applications in Section 4.2. In one case that we note, our new procedure leads to an improved result, but it turns out that for the bulk of them, almost uniform generation suffices. Yet, even in these cases, something is gained: conceptually, and in terms of analysis, it is simpler to use uniform generation. One can now do so without losing in the complexity.

2 Preliminaries

We begin with some general notation, then describe the most basic versions of the uniform generation problems, then describe extensions and enhancements, and finally some tools.

2.1 Notation and conventions

If S is a probability space then $x \stackrel{R}{\leftarrow} S$ denotes the operation of selecting an element uniformly at random according to S. If S is a set we let $\mathsf{Unif}(S)$ denote the probability space which puts a

[1] An algorithm for this problem is allowed, for technical reasons, to fail with some small probability, and this refers only to its output given that it did not fail.

uniform distribution on S. We typically use the shorthand $x \stackrel{R}{\leftarrow} S$ for $x \stackrel{R}{\leftarrow} \mathsf{Unif}(S)$. If S_1, S_2 are probability spaces then their statistical difference is

$$\|S_1 - S_2\| \;=\; \frac{1}{2}\sum_w \left|\Pr\left[\, x = w \;:\; x \stackrel{R}{\leftarrow} S_1 \,\right] - \Pr\left[\, x = w \;:\; x \stackrel{R}{\leftarrow} S_2 \,\right]\right| \,.$$

If A is a probabilistic algorithm then we denote by $A(x, y, \cdots; R)$ the output of A on inputs $x, y, \ldots$ and coin tosses R. We denote by $A(x, y, \cdots)$ the probability space that assigns to each possible output the probability it is generated, taken over the choice of R. We allow A to output a special symbol $\perp$ to indicate "failure." If f is a real-valued function of the inputs, We say A has failure probability at most $f(\cdot)$ —where f is a real-valued function of the inputs— if

$$\Pr\left[\, o = \perp \;:\; o \stackrel{R}{\leftarrow} A(x, y, \ldots) \,\right] \;\le\; f(x, y, \ldots) \,,$$

for all inputs $x, y, \ldots$. We are interested in the distribution over A's outputs when it does not fail. We let $\mathrm{Succ}(A, x, y, \ldots)$ denote the set of all random tapes R for which $A(x, y, \ldots; R) \neq \perp$.

Definition 2.1 If $A(\cdot, \cdot, \ldots)$ is a probabilistic algorith, we let $A^{\mathrm{succ}}(x, y, \ldots)$ be the probability space in which strings are assigned the probability of appearing as outputs of $A(x, y, \ldots)$ conditioned on the algorithm not failing; in other words, the probability of a string w under $A^{\mathrm{succ}}(x, y, \ldots)$ is

$$\Pr\left[\, A(x, y, \ldots; R) = w \;:\; R \stackrel{R}{\leftarrow} \mathrm{Succ}(A, x, y, \ldots) \,\right] \,.$$

An algorithm can take an oracle, to emphasize which we might call it an oracle algorithm. To supply an algorithm with an $\mathcal{NP}$ oracle means to supply it with an oracle for some $\mathcal{NP}$-complete language, which to be specific we fix to be SAT.

NP-RELATIONS. Let $R(\cdot, \cdot)$ be a binary relation. We say that R is an $\mathcal{NP}$-relation if it is polynomial time computable and, moreover, there exists a polynomial p such that $R(x, w) = 1$ implies $|w| \le p(|x|)$. For any $x \in \{0,1\}^*$ we let $R_x = \{\, w \in \{0,1\}^* \mid R(x, w) = 1 \,\}$ denote the *witness set* of x. We let $L_R = \{\, x \in \{0,1\}^* \mid R_x \neq \emptyset \,\}$ denote the language defined by R. Note that a language L is in $\mathcal{NP}$ iff there exists an NP-relation R such that $L = L_R$. We say that R is $\mathcal{NP}$-complete if L_R is $\mathcal{NP}$-complete.

2.2 Uniform Generation

The basic problem, as considered by [15], fixes some $\mathcal{NP}$-relation R and seeks a probabilistic algorithm that on input $x \in L_R$ outputs a string distributed uniformly in R_x. For technical reasons we allow algorithm to fail some fraction of the time. It will indicate this by outputting some special symbol $\perp$. Thus, the actual requirement is that the output distribution be uniform over R_x conditioned on the event that the output not be the special failure symbol $\perp$.

Definition 2.2 A *generator* for an $\mathcal{NP}$-relation R is a (possibly probabilistic, oracle) algorithm G that takes as input a string $x \in L_R$. It outputs either an element $w \in R_x$, or the special symbol $\perp$ indicating failure, and is required to have failure probability at most c for some constant c strictly less than 1.

Definition 2.3 Let G be a generator for $\mathcal{NP}$-relation R. We say that G is a *uniform generator for R* if for every $x \in L_R$ we have $G^{\mathrm{succ}}(R, x) = \mathsf{Unif}(R_x)$.

In other words, conditioned on the event that G does not fail, its output is distributed uniformly in R_x. (Refer to Definition 2.1 for the notation G^{succ}.)

Our results pertain to uniform generation. For the sake of discussing the history in the area, however, we also define the weaker notion of almost uniform generation.

Definition 2.4 Let $\delta\colon \mathsf{N} \to [0,1]$ and let G be a generator for $\mathcal{NP}$. We say that G is a $\delta(\cdot)$*-uniform generator for* R if for every $x \in L_R$ we have

$$\|G^{\mathrm{succ}}(x) - \mathsf{Unif}(R_x)\| \;\leq\; \delta(|x|) \ .$$

If $\delta(\cdot) \leq 2^{-(\cdot)}$ we call G an *almost uniform generator.*

Note a 0-uniform generator for R is a uniform generator for R.

2.3 Extensions and variations

Above we have described the most basic form of the problem. For applications, we often want additional properties. Since they are easily obtainable from the basic procedures, however, we make them the subject only of these remarks.

Lowering the failure probability. The first concern is the failure probability, which in applications we often need to be much lower than a constant. Standard error reduction techniques work. Namely, given a $\delta(\cdot)$-uniform generator U_1 for R and a security parameter k we can build a $\delta(\cdot)$-uniform generator U_2 for R with failure probability $f(k) = 2^{-k}$, and running time k times that of the original generator. Just repeat the execution of the original generator (on input x), each repetition with new coins, until some execution yields a non $\perp$ output, or we have exceeded k executions. In the former case output whatever was obtained, and in the latter output $\perp$. It is easy to see this yields a $\delta(\cdot)$-uniform generator with failure probability 2^{-k}.

Universal generators. Another issue is that in applications we often need the uniform generation ability for not one, but many $\mathcal{NP}$-relations, these being determined in some dynamic way, say via another algorithm. For this reason, we consider a slightly more general problem. We will not fix R, but provide it as input to the uniform generator. We will denote by $\langle R\rangle$ a "description" of R, meaning, specifically, a deterministic algorithm M that on input x, w, halts in $\mathrm{poly}(|x| + |w|)$ steps with output $R(x, w)$. The generator is given $\langle R\rangle$ as input and must then perform like a generator for R. We call such a generator *universal.*

For notational simplicity we view R as fixed in our constructions. However here, as with all known constructions of uniform or $\delta(\cdot)$-uniform generators, it is easy to see that the construction can work just as well if R is an input and not fixed, meaning the construction extends to one of a universal generator.

If one does not want to go back to the proof, here is a general reduction of universal generation to generation for a fixed $\mathcal{NP}$ relation. Let U be a universal $\mathcal{NP}$ relation, namely $U((M, x, 1^t), y)$ is 1 iff $M(x, y)$ halts within t steps with output 1. Let G be a uniform generator for U. Then one can get a universal generator via G, by running G on input $(\langle R\rangle, x, 1^t)$ to generate uniformly in R_x, where t is an appropriate time bound.

Separating parameters. A more minor point is that the function $\delta(\cdot)$ which measures the quality of a $\delta(\cdot)$-uniform generator is viewed as a function of the length of x. We can also make it a function of an independent security parameter k, so that for a given x, even a short one, we can get a distribution closer and closer to $\mathsf{Unif}(R_x)$ by using larger and larger values of k. We will not do this explicitly here.

2.4 Hashing

Let $H(n, m, t)$ denote a collection of t-wise independent hash functions of n bits to m bits. This means that for any $y_1, \ldots, y_t \in \{0,1\}^m$ and any distinct $x_1, \ldots, x_t \in \{0,1\}^n$ we have

$$\Pr\left[\, h(x_1) = y_1 \ \wedge \ \ldots \ \wedge \ h(x_t) = y_t \ : \ h \stackrel{R}{\leftarrow} H(n,m,t) \,\right] \ = \ 2^{-mt} \ .$$

To be concrete, we can use an implementation based on degree $t-1$ polynomials over a finite field. A hash function h in the family is described by a sequence $(a_0, \ldots, a_{t-1})$ of elements in the finite field $F = \mathrm{GF}(2^{\max(n,m)})$. The function h takes input $x \in \{0,1\}^n$, interprets it as an element of F under some fixed embedding of $\{0,1\}^n$ in F, computes $\sum_{j=0}^{t-1} a_j x^j$ in F, and outputs the first m bits of the result. The distribution over $H(n,m,t)$ is that induced by a random choice of $(a_0, \ldots, a_{t-1})$.

Note functions have description size $O(t \max(n,m)) = \mathrm{poly}(t,n,m)$, are $\mathrm{poly}(n,m,t)$ time computable, and can be identified from their description, properties we will need in the constructions.

For $\alpha \in \{0,1\}^m$ we let $h^{-1}(\alpha) = \{\, y \in \{0,1\}^n \mid h(y) = \alpha \,\}$.

2.5 Coin tossing

Since we are interested in picking elements uniformly from arbitrary sized sets we must look a little more closely at how coin tossing is modeled and achieved. The standard model for probabilistic algorithms allows algorithms to toss coins, namely the primitive operation is to be able to pick a random bit. But what if we want to pick an element at random out of a set of size three, eg. $\{1,2,3\}$? In fact, a coin tossing primitive does not allow one to always halt in polynomial time with output uniformly distributed in a set of size three. But we can take advantage of the fact that we are allowed to fail with a small probability. It is easy to see that there is a procedure C which runs in time $\mathrm{poly}(k)$, has $C^{\mathrm{succ}} = \mathsf{Unif}(\{1,2,3\})$, and failure probability at most 2^{-k}. This generalizes to any polynomial sized set which is explicitly given. Thus we will assume in the following that we can pick uniformly at random from any explicitly given polynomial sized set, with the understanding that the sum of the accumulated errors can be made small and included in the bound on the overall failure probability of the algorithm.

Notice this indicates that the need to have a failure probability in a uniform generation process is inherent from the model of coin tossing.

3 The procedure

The goal of this section is to establish the following:

Theorem 3.1 *Let R be an $\mathcal{NP}$-relation. Then there is a uniform generator for R which is implementable in probabilistic, polynomial time with an $\mathcal{NP}$-oracle.*

We will now prove this theorem. Fix an $\mathcal{NP}$-relation R. We let $n = |x|$ and assume $R_x \subseteq \{0,1\}^n$, meaning all witnesses have the same length, and this is the length of the input. (This is without loss of generality: it can be achieved by suitable padding.) Below, we begin by describing some facts about hash function induced partitions of R_x which guide the choice of parameters in our procedure. The high level procedure is then discussed, and following that we explain how its constituent subroutines are implemented.

3.1 Partitions

For $h\colon \{0,1\}^n \to \{0,1\}^m$, $x \in L_R$ and $\alpha \in \{0,1\}^m$ we define

$$R_{x,h,\alpha} \ = \ \{\, y \in R_x \mid h(y) = \alpha \,\} \ = \ h^{-1}(\alpha) \cap R_x \ .$$

Namely, the set of pre-images of α under h that fall in R_x. As we range over $\alpha \in \{0,1\}^m$ (with x, h fixed) these sets partition R_x. We call them the cells of the partition. Our uniform generation procedure will utilize properties of this partition. To this end we make the following:

Definition 3.2 We say that a map $h\colon \{0,1\}^n \to \{0,1\}^m$ *makes small cells* in R_x if

$$|R_{x,h,\alpha}| \;<\; 2n^2 \quad \text{for all } \alpha \in \{0,1\}^m .$$

We say it *makes non-trivial cells* in R_x if

$$|R_{x,h,\alpha}| \;\geq\; n^2/2 \quad \text{for all } \alpha \in \{0,1\}^m .$$

If h is drawn at random from $H(n,m,t)$ for $t \geq 1$, then the expected size of a cell $R_{x,h,\alpha}$ is $|R_x|/2^m$, for any $\alpha \in \{0,1\}^m$. We would like the partition to be "well balanced," meaning *all* cells are about the expected size. (Specifically, between 1/2 and 3/2 of the expected size.) The following lemma upper bounds the chance that the partition of R_x induced by a random t-wise independent hash function is not well balanced, as a function of $|R_x|, m, t$.

Lemma 3.3 *Assume $R_x \subseteq \{0,1\}^n$. Let m be an integer, $t \geq 4$ an even integer, and let $\beta = t2^m/|R_x|$. Then*

$$\Pr\left[\,\exists \alpha \in \{0,1\}^m \text{ such that } |R_{x,h,\alpha}| < \frac{|R_x|}{2^{m+1}} \text{ or } |R_{x,h,\alpha}| > \frac{3|R_x|}{2^{m+1}} \;:\; h \stackrel{R}{\leftarrow} H(n,m,t)\right]$$
$$<\; 2^{m+3} \cdot [4\beta(1+\beta)]^{t/2} \;.$$

The proof uses standard t-wise independence techniques and can be found in Appendix A. The goal of our procedure will be to find h that always makes small cells in R_x, and with high probability makes non-trivial cells. The latter will exploit the above lemma by choosing appropriate values of m, t to make small the probability that the partition is not well balanced. Specifically, the next lemma shows a choice of parameters which reduce the probability of getting a partition whose size deviates significantly from n^2.

Lemma 3.4 *Assume $R_x \subseteq \{0,1\}^n$ and $|R_x| > 2n^2$. Let $i = \log_2 |R_x|$ and $\ell = 2\log_2 n$ and assume these are integers. Let $m = i - \ell$ and $t = n$ and assume t is even and at least 4. Then*

$$\Pr\left[\,\exists \alpha \in \{0,1\}^m \text{ such that } |R_{x,h,\alpha}| < n^2/2 \text{ or } |R_{x,h,\alpha}| > 3n^2/2 \;:\; h \stackrel{R}{\leftarrow} H(n,m,t)\right] \;<\; 0.1 \,.$$

Proof: Notice the choice of parameters guarantees that $2^\ell = n^2$ and $2^i = |R_x|$ and $2^m = 2^i/2^\ell = |R_x|/n^2$ and $m > 0$. Apply Lemma 3.3. We have $|R_x|/2^{m+1} = n^2/2$ and $\beta = t2^m/|R_x| = t/n^2 = 1/n$ which gives us

$$\Pr\left[\,\exists \alpha \in \{0,1\}^m \text{ such that } |R_{x,h,\alpha}| < n^2/2 \text{ or } |R_{x,h,\alpha}| > 3n^2/2 \;:\; h \stackrel{R}{\leftarrow} H(n,m,t)\right]$$
$$<\; 2^{m+3} \cdot \left[\frac{4}{n}\left(1 + \frac{1}{n}\right)\right]^{n/2}$$
$$\leq\; 2^{n+3} \cdot \left(\frac{8}{n}\right)^{n/2}$$

But this is at most 0.1 for large enough n. ▮

This lemma guides the choice of $2n^2$ as the value based on which the following procedure "pivots."

3.2 High level procedure

The following generator G takes as input $x \in L_R$. Recall that $R_x \subseteq \{0,1\}^n$. Let $\ell \stackrel{\text{def}}{=} 2\lceil \log_2 n \rceil$. We describe the algorithm at a high level, and later explain how exactly the individual steps can be implemented in probabilistic polynomial time with an $\mathcal{NP}$ oracle.

Algorithm $G(x)$

1. If $|R_x| \le 2n^2$ then compute a listing $y_1, \ldots, y_s$ of the members of the set R_x, select j at random from $\{1, \ldots, s\}$, output y_j, and halt. Else go on to the next step. // Lemma 3.5 indicates how, in polynomial time with an $\mathcal{NP}$-oracle, to test the size of R_x and obtain the set when its size is at most $2n^2$.
2. Find an $i \in \{\ell, \ldots, n\}$ and a $h \in H(n, i-\ell, n)$ such that
   - h makes small cells in R_x
   - With probability at least 0.9 it is the case that h makes non-trivial cells in R_x.

   // See Definition 3.2 for the meaning of the terms. The probability of 0.9 is over the choices of the procedure. See Lemma 3.6 for how to do this in the desired complexity.
3. Select α at random from $\{0,1\}^{i-\ell}$ and compute a listing $y_1, \ldots, y_s$ of the member of the set $R_{x,h,\alpha}$. // Since h makes small cells in R_x we know that $0 \le s \le 2n^2$. See Lemma 3.7 for how to do this in the desired complexity.
4. Pick j at random from $\{1, \ldots, 2n^2\}$. If $j \le s$ output y_j and halt, else output $\perp$ and halt.

In the next subsection we will prove the above mentioned lemmas which show how to implement the above steps in probabilistic, polynomial time with an $\mathcal{NP}$ oracle. Then in Section 3.4 we will conclude the proof of Theorem 3.1 by showing that G is a uniform generator for R.

3.3 Implementation of subroutines

To simplify the exposition we assume $n^2 = 2^\ell$ (rather than $\ell = 2\lceil \log_2 n \rceil$) and also that $\log_2 |R_x|$ is an integer (meaning $|R_x|$ is a power of two). A more careful analysis removes these assumptions.

STEP 1. Lemma 3.5 below indicates how to execute the first step of the high level procedure in probabilistic polynomial time with an $\mathcal{NP}$-oracle. Namely run $M_1^{S_1,S_2}(x, 1^{2n^2})$, where the algorithm M_1 and the $\mathcal{NP}$ sets S_1, S_2 are as in the lemma. If this returns 0 then go to step two of the high level procedure. Else it returns R_x with the guarantee that this set has size at most $2n^2$, as the first step of the high level procedure requires.

Lemma 3.5 *There is a polynomial time oracle algorithm M_1 and sets $S_1, S_2 \in \mathcal{NP}$ such that $M_1^{S_1,S_2}(x, 1^m)$ outputs 0 if $|R_x| > m$, and outputs the set R_x otherwise, for any $x \in \{0,1\}^*$ and $m \in \mathsf{N}$.*

Proof: Let

$$
\begin{aligned}
S_1 &= \{\, (x', 1^k) \mid \exists\, y_1, \ldots, y_k \text{ such that } y_1, \ldots, y_k \text{ are distinct and } \forall i \in [k] : R(x', y_i) = 1 \,\} \\
S_2 &= \{\, (x', 1^{s'}, 1^{i'}, 1^{j'}) \mid \exists\, y_1, \ldots, y_{s'} \text{ such that } y_1 \prec \cdots \prec y_{s'} \text{ and } \forall i \in [s'] : R(x', y_i) = 1 \\
&\qquad \text{and } 1 \le j' \le |y_{i'}| \text{ and } 1 \le i' \le s' \text{ and the } j'\text{-th bit of } y_{i'} \text{ is } 0 \,\}\,.
\end{aligned}
$$

Here $\prec$ denotes some ordering (eg. lexicographic) on $\{0,1\}^*$. These sets are certainly in $\mathcal{NP}$ given that R is an $\mathcal{NP}$-relation. M_1 begins be executing the following code, which outputs 0 if $|R_x| > m$, and otherwise computes the cardinality s of R_x.

If $(x, 1^{m+1}) \in S_1$ **then output** 0 and **halt**
Else
  $s \leftarrow 0$
  While $(x, 1^{s+1}) \in S_1$ **do** $s \leftarrow s+1$

If $s = 0$ then R_x is empty, so M_1 outputs the empty set $\emptyset$ and halts. If $s > 0$ then M_1 finds the elements $y_1, \ldots, y_s$ of R_x one by one, and bit by bit for each one. It does so using queries to S_2 in the natural way:

For $i = 1, \ldots, s$ **do**
  For $j = 1, \ldots, n$ **do** // Recall that all witnesses have length $n = |x|$ by assumption
    If $(x, 1^s, 1^i, 1^j) \in S_2$ **then** $y_{i,j} \leftarrow 0$ **else** $y_{i,j} \leftarrow 1$
  $y_i \leftarrow y_{i,1}y_{1,2} \cdots y_{i,n}$
Output $\{y_1, \ldots, y_s\}$

It is easy to verify that M_1 runs in poly(n, m) time and has the claimed properties. ▮

Step 2. The following lemma indicates how to find the hash function h to satisfy the properties required in Step 2 of the procedure G.

Lemma 3.6 *There is a probabilistic, polynomial time algorithm M_2 and a set $S_3 \in \mathcal{NP}$ such that given $x \in L_R$ for which $|R_x| > 2n^2$, algorithm $M_2^{S_3}$ outputs a pair (i, h) such that*
(1) *$\ell \leq i \leq n$ and $h \in H(n, i-\ell, n)$*
(2) *h makes small cells in R_x*
(3) *With probability at least 0.9 it is the case that h makes non-trivial cells in R_x.*
Here $\ell = 2\log_2 n$.

Proof: Let

$$\begin{aligned} S_3 \;=\; \{\, (x', h', 1^m) \mid \exists\, \alpha, y_1, \ldots, y_k & \text{ such that } k = 2|x'|^2 \text{ and } y_1, \ldots, y_k \text{ are distinct and} \\ & h' \in H(|x'|, m, |x'|) \text{ and } \forall j \in [k] \,:\, R(x', y_j) = 1 \text{ and } \forall j \in [k] \,:\, h'(y_j) = \alpha \,\}\,. \end{aligned}$$

This set is certainly in $\mathcal{NP}$. (We use here that h' can be evaluated in poly(n, m) time if it is in $H(n, m, n)$, and membership in the latter set can be tested in polynomial time.) Procedure M_2 goes through the values $i = \ell, \ldots, n-1$ seeking $h \in H(n, i-\ell, n)$ with the desired properties, as follows. Below h_{ID}: $\{0,1\}^n \to \{0,1\}^{n-\ell}$ is the map which on input $z \in \{0,1\}^n$ returns the first $n - \ell$ bits of z.
$i \leftarrow \ell - 1$
Repeat
  $i \leftarrow i + 1$
  Choose h at random from $H(n, i-\ell, n)$
Until $(x, h, 1^{i-\ell}) \notin S_3$ or $i = n-1$
If $(x, h, 1^m) \notin S_3$ **then output** (i, h) **else output** (n, h_{ID})

Claim 1. $\ell \leq i \leq n$ and $h \in H(n, i-\ell, n)$ where (i, h) is the output of the above.

Proof. Clearly $h \in H(n, i-\ell, n)$ for $i \leq n-1$. However, it is also true for $i = n$ because $h = h_{\text{ID}}$ is the truncation of the polynomial consisting of just a linear term, and hence is in $H(n, n-\ell, n)$. □

Claim 2. If (i, h) is the output of the above then h makes small cells in R_x.

Proof. First suppose $i \leq n-1$. In this case, the procedure has tested that $(x, h, 1^{i-\ell}) \notin S_3$. By definition of S_3 this means that $|R_{x,h,\alpha}| < 2n^2$ for all $\alpha \in \{0,1\}^{i-\ell}$.

Now suppose $i = n$. In this case, $h = h_{\mathrm{ID}}$. Clearly $|h^{-1}(\alpha)| = 2^\ell$ for any $\alpha \in \{0,1\}^{n-\ell}$. But $R_{x,h,\alpha} \subseteq h^{-1}(\alpha)$ so $|R_{x,h,\alpha}| \leq 2^\ell = n^2 < 2n^2$. □

Claim 3. $i \geq \log_2 |R_x|$ where (i, h) is the output of the above.

Proof. $|R_x| = \sum_{\alpha \in \{0,1\}^{i-\ell}} |R_{x,h,\alpha}| < 2^{i-\ell} \cdot 2n^2 = 2^{i+1}$. Taking logs, we have $\log_2 |R_x| < i+1$. Under the assumption that $\log_2 |R_x|$ is an integer, the claim follows. □

Claim 4. Let (i, h) be the output of the above. With probability at least 0.9 it will be the case that h makes non-trivial cells in R_x.

Proof. We know from Claim 3 that $i \geq \log_2 |R_x|$, so the first value with which the procedure could halt is $i = \log_2 |R_x|$. We show that in fact when the value of i after the first step in the **Repeat** loop is $\log_2 |R_x|$ then, with probability 0.9 over the choice of h from $H(n, i-\ell, n)$ that is made in the loop, it will be the case that $(x, h, 1^{i-\ell}) \notin S_3$ and h makes non-trivial cells in R_x. This means that with probability 0.9 the procedure halts with $i = \log_2 |R_x|$ and the corresponding h makes non-trivial cells in R_x, and this implies the claim.

So let $i = \log_2 |R_x|$. We claim that

$$\Pr\left[\, \exists \alpha \in \{0,1\}^{i-\ell} \text{ such that } |R_{x,h,\alpha}| < n^2/2 \text{ or } |R_{x,h,\alpha}| > 3n^2/2 \;:\; h \xleftarrow{R} H(n, i-\ell, n) \,\right] \;<\; 0.1 \,.$$

This is true by Lemma 3.4. To check this, notice that the premises of Lemma 3.4 are true here. We have $|R_x| > 2n^2$ by assumption in the lemma statement. We have $i = \log_2 |R_x|$ and $\ell = 2\log_2 n$ by definition. And we may assume wlog that n is even and at least 4.

So with probability at least 0.9 the h chosen in the loop at this point will satisfy

$$\forall \alpha \in \{0,1\}^{i-\ell} \;:\; \frac{n^2}{2} \;\leq\; |R_{x,h,\alpha}| \;\leq\; \frac{3n^2}{2} \,.$$

So this h makes small cells in R_x and also makes non-trivial cells in R_x. The former implies that $(x, h, 1^{i-\ell}) \notin S_3$ so the procedure halts. □

Putting together the claims gives the desired conclusions for the lemma. ▌

STEP 3. The final lemma shows how to implement Step 3 of G. It is analgous to Lemma 3.5.

Lemma 3.7 *There is a polynomial time oracle algorithm M_3 and sets $S_4, S_5 \in \mathcal{NP}$ such that $M_3^{S_4,S_5}(x, h, \alpha, 1^m)$ outputs 0 if $|R_{x,h,\alpha}| \geq 2|x|^2$, and outputs the set $R_{x,h,\alpha}$ otherwise, for any $x \in L_R$, $h \in H(|x|, m, |x|)$, $\alpha \in \{0,1\}^m$, and $m \in \mathsf{N}$.*

Proof: The proof is analogous to that of Lemma 3.5. The corresponding sets are

$$\begin{aligned} S_4 &= \{\, (x', h', \alpha', 1^k) \mid \exists\, y_1, \ldots, y_k \text{ such that } y_1, \ldots, y_k \text{ are distinct and } \forall i \in [k] \,:\, R(x', y_i) = 1 \\ &\qquad \text{and } h' \in H(|x|, m, |x|) \text{ and } \forall i \in [k] \,:\, h'(y_i) = \alpha' \,\} \\ S_5 &= \{\, (x', h', \alpha', 1^{s'}, 1^{i'}, 1^{j'}) \mid \exists\, y_1, \ldots, y_{s'} \text{ such that } y_1 \prec \cdots \prec y_{s'} \text{ and } \forall i \in [s'] \,:\, R(x', y_i) = 1 \\ &\qquad \text{and } h' \in H(|x|, m, |x|) \text{ and } \forall i \in [s'] \,:\, h'(y_i) = \alpha' \text{ and } 1 \leq j' \leq |y_{i'}| \\ &\qquad \text{and } 1 \leq i' \leq s' \text{ and the } j'\text{-th bit of } y_{i'} \text{ is } 0 \,\} \,. \end{aligned}$$

Here $\prec$ denotes some ordering (eg. lexicographic) on $\{0,1\}^*$. These sets are certainly in $\mathcal{NP}$ given that R is an $\mathcal{NP}$-relation, and testing membership or evaluation of functions in the hash family can be done in polynomial time. As in Lemma 3.5, M_3 begins by making a sequence of queries to S_4 at the end of which it outputs 0 if $|R_{x,h,\alpha}| \geq 2n^2$ and otherwise outputs the size s of $R_{x,h,\alpha}$. Then it computes the members $y_1, \ldots, y_s$ of $R_{x,h,\alpha}$ bit by bit, using queries to S_5. ▌

3.4 Proof of Theorem 3.1

From the above we already know that G can be implemented in probabilistic, polynomial time given an $\mathcal{NP}$ oracle. We now claim that G is a uniform generator for R with failure probability at most $c \stackrel{\text{def}}{=} 0.8$. This will prove Theorem 3.1.

In case $|R_x| \leq 2n^2$ the algorithm always halts in Step 1, outputting a uniformly chosen element of R_x. Thus, we focus on the case $|R_x| > 2n^2$.

We note that by choice of h in Step 2 we have $|R_{x,h,\alpha}| \leq 2n^2$ for every $\alpha \in \{0,1\}^{i-\ell}$, and also, with probability at least 0.9, $|R_{x,h,\alpha}| \geq n^2/2$ for every $\alpha \in \{0,1\}^{i-\ell}$. Thus, an element in R_x is output in Step 3 with probability at least $0.9 \cdot \frac{n^2/2}{2n^2} = 0.225$. We allow an additional 0.025 failure proability to cover any failure in the coin tossing, as discussed in Section 2.5, so that the probability of a non-$\perp$ output is at least 0.2, meaning the failure probability is at most 0.8.

Now, we need to establish (still for the case $|R_x| > 2n^2$ since the other is done) that in case of non-failure, the output is uniformly distributed in R_x. To establish this, we compute the probability that *any fixed* $y \in R_x$ is output. Here we consider any possible (fixed) choice of h in Step 2 (not necessarily one of the 90% of choices that makes non-trivial cells in R_x). Thus, the randomization is only over the choice of α in Step 3, and the choices in Step 4. We have:

$$\begin{aligned} \Pr[\, y \text{ is output}\,] &= \Pr\left[\, h(y) = \alpha \;:\; \alpha \stackrel{R}{\leftarrow} \{0,1\}^{i-\ell} \,\right] \cdot \frac{1}{2n^2} \\ &= 2^{-(i-\ell)} \cdot \frac{1}{2n^2} \,. \end{aligned}$$

And this last number is independent of y.

4 History and Applications

This is a survey of the history and some applications of uniform or almost uniform generation. In some applications which we will note, our procedure leads to improvements.

4.1 History

Fix an $\mathcal{NP}$ language L and its defining $\mathcal{NP}$ relation R. The approximate counting problem is the following:

> **Approximate Counting of $\mathcal{NP}$-witnesses**
> GIVEN: $x \in L$ and $\epsilon > 0$
> OUTPUT: A number r such that $r/(1+\epsilon) \leq |R_x| \leq (1+\epsilon)r$.

Jerrum et. al. [15] provided a probabilistic, polynomial time reduction of the problem of uniform generation to the problem of approximate counting. Earlier, Stockmeyer [20] had presented a procedure that achieved approximate counting in $\text{poly}(|x|, \log \epsilon^{-1})$ time given a Σ_2^{P} oracle. Combined, this yielded a uniform generator for $\mathcal{NP}$-witnesses that ran in probabilistic, polynomial time with a Σ_2^{P} oracle.

As for almost uniform generation, one can relax the notion of approximate counting to one of almost approximate counting. Here the output r must satisfy the above property only with probability $1-\delta$ for some parameter $\delta < 1$. Jerrum et. al. [15] defined this notion. A polynomial time (more specifically, $\mathrm{poly}(|x|, \log \epsilon^{-1}, \log \delta^{-1})$ time) with $\mathcal{NP}$ oracle solution for almost approximate counting would yield, via their reduction, a probabilistic polynomial time with $\mathcal{NP}$-oracle solution to almost uniform generation.

Jerrum et. al. [15] note that a polynomial time with $\mathcal{NP}$ oracle solution to almost approximate counting is implicit in Sipser [19] and Stockmeyer [20], or can be derived via the probabilistic bisection technique of Valiant and Vazirani [21]. Details of the former solution were worked out and presented by Bellare and Petrank [5] in the context of applications to zero-knowledge proof systems, based on more recent versions of Sipser and Stockmeyer's hashing techniques that were developed in [14, 9, 1]. Putting this together yields the probabilistic, polynomial time with $\mathcal{NP}$ oracle solution to almost uniform generation.

As an inspection of our procedure shows, we do not use the Jerrum et. al.'s reduction to approximate counting to achieve uniform generation. Instead, we directly use hashing based techniques of works like [19, 20, 14, 9, 1]. Appropriate enhancement and application of these techniques yields the new result.

4.2 Applications

We discuss a collection of results obtained in the area of interactive proof systems over the last few years that have exploited uniform or almost uniform generation. The goal of these results has been to classify certain functions or languages, arising in this area, in terms of their time complexity.

A typical result in this class has the following form. We start with an interactive proof system (P, V) for a language L which has some extra features, pertaining perhaps to its knowledge complexity, or the complexity of the interactive proof system. We now want to find the most efficient possible decision procedure for L or some associated function. The approach is to simplify the prover P via a use of uniform generation and thereby get the desired procedure.

For the bulk of these applications, the complexity of the final procedure is the object to minimize, while small errors in the uniform generation process do not affect the result. Thus, they have for the most part exploited the fact that almost uniform generation is possible in probabilistic polynomial time with an $\mathcal{NP}$ oracle. This accounts for the fact that in these results, you will typically see conclusions about certain languages being in $\mathcal{BPP}^{\mathcal{NP}}$. (In some cases, the results are about certain functions being computable in probabilistic polynomial time with an $\mathcal{NP}$ oracle.)

The complexity of ZK provers. The prover in a statistical zero knowledge (SZK) proof for a language L is a (probabilistic) function which given the common input x and conversation so far outputs the next message to send to the verifier. The question here is: what is the computational complexity of this function? Even though SZK languages are known to be in $\Sigma_2^{\mathrm{P}} \cap \Pi^{\mathrm{P}}{}_2$ [9, 1], it is not a priori clear that the prover, as a function is even restricted to probabilistic $\mathcal{PSPACE}$.

The question was first considered by [4] who reduced the complexity of the prover to that of (almost) uniform generation in such a way that SZK versus an honest verifier was maintained. A more general reduction, provided by [5], maintained SZK against all verifiers, as the definition of SZK requires. Jerrum et. al.'s result [15] could then be applied to say that any language having a SZK proof has one in which the prover is a probabilistic, polynomial algorithm with an $\mathcal{NP}$ oracle.

However, almost uniform generation is not enough to maintain perfect zero knowledge (PZK) and in this case, the result of [5], exploiting the uniform generation procedure of [15], was that any language with a PZK proof has one in which the prover is a probabilistic, polynomial algorithm with a Σ_2^{P} oracle. Our Theorem 3.1 can be used to improve this: combining it with the reduction of [5] we get that any language with a PZK proof has one in which the prover is a probabilistic, polynomial algorithm with a $\mathcal{NP}$ oracle.

Time complexity versus knowledge complexity. Knowledge complexity, suggested by [13] and defined by [12], provides a way to measure the number of bits of knowledge that a prover reveals to a verifier about some string x, in the process of proving that x belongs to some underlying language L. Let $\mathcal{SKC}[\kappa(\cdot)]$ denotes the class of languages possessing interactive proofs of negligible error probability and statistical knowledge complexity (SKC) at most $\kappa(\cdot)$. A body of work [5, 11, 17] has sought extensions to non-zero SKC of the results of [9, 1] which showed $\mathcal{SKC}[0] \subseteq \Sigma_2^{\mathrm{P}} \cap \Pi^{\mathrm{P}}{}_2$.

The first results used almost uniform generation. Specifically, Bellare and Petrank [5] provided a decision procedure for a language based on a SKC simulator and (almost) uniform generation, which Goldreich, Ostrovsky and Petrank [11] exploited to show that $\mathcal{SKC}[\log(\cdot)] \subseteq \mathcal{BPP}^{\mathcal{NP}}$. (Later, Petrank and Tardos [17] showed that $\mathcal{SKC}[\log(\cdot)] \subseteq \Sigma_2^{\mathrm{P}} \cap \Pi^{\mathrm{P}}{}_2$; this final result did not use uniform generation.)

Shared randomness in two prover proofs. We know that the class of languages recognized by two prover statistical zero-knowledge interactive proof systems equals $\mathcal{NEXP}$ [2, 7]. Bellare, Feige and Kilian [3] showed that a certain model feature —namely the fact that the two provers are allowed to share a random string before the protocol begins— is crucial to this result. Specifically, they showed that if this string is absent then the class of languages possessing SZK two prover proofs collapses to $\mathcal{BPP}^{\mathcal{NP}}$. The decision procedure which establishes this is based on almost uniform generation.

Complexity of proofs with bounded communication. Goldreich and Håstad [10] investigate the complexity of languages as a function of the communication complexity of interactive proof systems that recognize them. One of their results is that if the total number of bits sent by the prover in the interactive proof is logarithmic then the language is in $\mathcal{BPP}^{\mathcal{NP}}$. The decision procedure that establishes this exploits almost uniform generation.

References

[1] W. Aiello and J. Håstad. Statistical Zero-Knowledge Languages can be Recognized in Two Rounds. *Journal of Computer and System Sciences*, Vol. 45, No. 3, 1991, pp. 327–345.

[2] L. Babai, L. Fortnow and C. Lund. Non-Deterministic Exponential Time has Two-Prover Interactive Protocols. *Computational Complexity*, Vol. 1, 1991, pp. 3–40. (See also addendum in Vol. 2, 1992, pp. 374.)

[3] M. Bellare, U. Feige and J. Kilian. On the Role of Shared Randomness in Two Prover Proof Systems. *Proceedings of the third Israel Symposium on Theory and Computing Systems*, IEEE, 1995.

[4] M. Bellare, S. Micali and R. Ostrovsky. The (true) complexity of statistical zero-knowledge. *Proceedings of the* 22nd *Annual Symposium on the Theory of Computing*, ACM, 1990.

[5] M. Bellare and E. Petrank. Making Zero-Knowledge Provers Efficient. *Proceedings of the* 24th *Annual Symposium on the Theory of Computing*, ACM, 1992.

[6] M. Bellare and J. Rompel. Randomness-efficient oblivious sampling. *Proceedings of the* 35th *Symposium on Foundations of Computer Science*, IEEE, 1994.

[7] M. Ben-Or, S. Goldwasser, J. Kilian and A. Wigderson. Multi-prover Interactive Proofs: How to Remove Intractability Assumptions. *Proceedings of the* 20th *Annual Symposium on the Theory of Computing*, ACM, 1988.

[8] B. Berger and J. Rompel. Simulating $(\log^c n)$-wise independence in NC. *Proceedings of the* 30th *Symposium on Foundations of Computer Science*, IEEE, 1989.

[9] L. Fortnow. The Complexity of Perfect Zero-Knowledge. In *Advances in Computing Research: a research annual*, Vol. 5 (Randomness and Computation, S. Micali, ed.), pp. 327–343, 1989.

[10] O. Goldreich and J. Håstad. On the Complexity of Interactive Proofs with Bounded Communication. To appear in *IPL*. See also ECCC Report TR96-018, 1996.

[11] O. Goldreich, R. Ostrovsky and E. Petrank. Knowledge Complexity and Computational Complexity. *Proceedings of the* 26th *Annual Symposium on the Theory of Computing*, ACM, 1994.

[12] O. Goldreich and E. Petrank. Quantifying Knowledge Complexity. *Proceedings of the* 32nd *Symposium on Foundations of Computer Science*, IEEE, 1991.

[13] S. Goldwasser, S. Micali and C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SIAM Journal on Computing*, Vol. 18, pp. 186–208, 1989.

[14] S. Goldwasser and M. Sipser. Private Coins versus Public Coins in Interactive Proof Systems. *Advances in Computing Research: a research annual*, Vol. 5 (Randomness and Computation, S. Micali, ed.), pp. 73–90, 1989.

[15] M. Jerrum, L. Valiant and V. Vazirani. Random Generation of Combinatorial Structures from a Uniform Distribution. *Theoretical Computer Science*, Vol. 43, pp. 169–188, 1986.

[16] R. Motwani, J. Naor, and M. Naor. The probabilistic method yields deterministic parallel algorithms. *Proceedings of the* 30th *Symposium on Foundations of Computer Science*, IEEE, 1989.

[17] E. Petrank and G. Tardos. On the knowledge complexity of NP. *Proceedings of the* 37th *Symposium on Foundations of Computer Science*, IEEE, 1996.

[18] J. Schmidt, A. Siegel, A. Srinivasan. Chernoff-Hoeffding bounds for Applications with Limited Independence. *Proceedings of the* 4th *Annual Symposium on Discrete Algorithms*, ACM-SIAM, 1993.

[19] M. Sipser. A Complexity Theoretic Approach to Randomness. *Proceedings of the* 15th *Annual Symposium on the Theory of Computing*, ACM, 1983.

[20] L. Stockmeyer. The Complexity of Approximate Counting. *Proceedings of the* 15th *Annual Symposium on the Theory of Computing*, ACM, 1983.

[21] L. Valiant and V. Vazirani. $\mathcal{NP}$ is as Easy as Detecting Unique Solutions. *Theoretical Computer Science*, Vol. 47, No. 1, pp. 85–93, 1986.

A Proof of Lemma 3.3

We will make use of a "t-wise independent tail inequality". This is a Chebychev type bound for the case where the random variables are not fully independent, but are t-wise independent. Such inequalities are proved by a higher moment method, and can be found in the literature. Specifically, we use the following one from [6]. (It seems that [8, 16] were the first to use such bounds in the computer science literature. An in depth investigation which provides a variety of bounds is [18].)

Lemma A.1 **[6]** *Let $t \geq 4$ be an even integer. Suppose $X_1, \ldots, X_n$ are t-wise independent random variables taking values in $[0,1]$. Let $X = X_1 + \cdots + X_n$ and $\mu = \mathbf{E}[X]$, and let $A > 0$. Then*

$$\Pr[\,|X - \mu| \geq A\,] \;\leq\; 8 \cdot \left(\frac{t\mu + t^2}{A^2}\right)^{t/2} .$$

Now let us proceed to the proof of Lemma 3.3. Fix an $\alpha \in \{0,1\}^m$ and for each $y \in R_x$ define the random variable

$$\zeta_y \;=\; \begin{cases} 1 & \text{if } h(y) = \alpha \\ 0 & \text{otherwise.} \end{cases}$$

Under a random choice of h from $H(n,m,t)$ we have

$$\mathbf{E}[\zeta_y] \;=\; \Pr[\,h(y) = \alpha\,] \;=\; 2^{-m} .$$

Let $\zeta = \sum_{y \in R_x} \zeta_y$ and $\mu = \mathbf{E}[\zeta]$. Notice that $\zeta = |R_{x,h,\alpha}|$. Linearity of expectation tells us that $\mu = |R_x|/2^m$. So

$$\Pr\left[\,|R_{x,h,\alpha}| < \frac{|R_x|}{2^{m+1}} \text{ or } |R_{x,h,\alpha}| > \frac{3|R_x|}{2^{m+1}}\,\right] \;=\; \Pr[\,|X - \mu| > \mu/2\,] .$$

Now, notice also that the random variables $\{\zeta_y\}_{y \in R_x}$ are t-wise independent. So we can apply Lemma A.1 to get

$$\begin{aligned} \Pr[\,|X - \mu| \geq \mu/2\,] &\leq 8 \cdot \left(\frac{t\mu + t^2}{(\mu/2)^2}\right)^{t/2} \\ &= 8 \cdot \left(\frac{t|R_x|2^{-m} + t^2}{|R_x|^2 2^{-2m}/4}\right)^{t/2} \\ &= 8[4\beta(1+\beta)]^{t/2} , \end{aligned}$$

where $\beta = t2^m/|R_x|$.

Finally, note this was true for any fixed $\alpha \in \{0,1\}^m$. The desired bound of the lemma is obtained by applying the union bound.