Electronic Colloquium on Computational Complexity - Reports Series 1995 - available via:

ECCC
TR95-061

FTP: ftp.eccc.uni-trier.de:/pub/eccc/
WWW: http://www.eccc.uni-trier.de/eccc/
Email: ftpmail@ftp.eccc.uni-trier.de with subject 'help eccc'

Hitting sets derandomize BPP

Alexander E. Andreev
Department of Mathematics
University of Moscow
RU

Andrea E. F. Clementi
Dipatimento di Scienze dell'Informazione
University of Rome
I

José D. P. Rolim*
Centre Universitaire d'Informatique
University of Geneva
CH

Abstract

We show that hitting sets can derandomize *any* BPP-algorithm. This gives a positive answer to a fundamental open question in probabilistic algorithms. More precisely, we present a polynomial time deterministic algorithm which uses any given hitting set to approximate the fractions of 1's in the output of any boolean circuit of polynomial size. This new algorithm implies that if a quick hitting set generator with logarithmic price exists then $BPP = P$. Furthermore, we generalize this result by showing that the existence of a quick hitting set generator with price k implies that $BPTIME(t) \subseteq DTIME(2^{O(k(t^{O(1)}))})$. The existence of quick hitting set generators is thus a new weaker sufficient condition to obtain $BPP = P$; this can be considered as another strong indication that the gap between probabilistic and deterministic computational power is not large.

*Contact author: Centre Universitaire d'Informatique, University of Geneva, 24 rue General Dufour, CH 1204 Geneva.
E-mail: rolim@cui.unige.ch

1 Introduction

- *Motivations and previous results.* This paper addresses the issue of the derandomization of probabilistic algorithms, i.e., the design of general methods that permit an efficient deterministic simulation of algorithms which make use of random bits. *Pseudo-Random Generators* (PSRG's) ([3, 11]) constitute the best general method to this aim. A PSRG is a function $G = \{G_n : \{0,1\}^{k(n)} \rightarrow \{0,1\}^n, n > 0\}$, denoted by $G : k(n) \rightarrow n$ that "stretches" $k(n)$ truly random bits into n pseudo-random bits; more formally G is a PSRG if for any sufficiently large n and for any boolean circuit $C : \{0,1\}^n \rightarrow \{0,1\}$ whose size is at most n we have: $|\mathbf{Pr}(C(\vec{y}) = 1) - \mathbf{Pr}(C(G_n(\vec{x})) = 1)| \leq 1/n$ (where $\vec{y}$ is chosen uniformly at random in $\{0,1\}^n$, and $\vec{x}$ in $\{0,1\}^{k(n)}$). The intuition behind this is that the output of a PSRG looks "random" to any small circuit. Nisan and Wigderson [8] showed a method to construct *quick* PSRG's (i.e. PSRG's that are computable in polynomial-time in the length of their output[1]) which is based on a particular hardness assumption (i.e. the existence of boolean functions in EXP having exponential *hardness* [2, 7, 8]). In particular, they proved that, using $k(n) = O(\log n)$ truly random bits, these quick PSRG's can efficiently derandomize any two-sided error, polynomial-time algorithm (i.e. any BPP-algorithm).

In this paper we show a different approach to derandomize algorithms: we give a positive answer to the question whether *Hitting Set Generators* ([9, 6, 1]) can achieve equivalent general performances to those obtained by PSRG's. A *Hitting Set Generator* (HSG) is a function $H = \{H_n : \{0,1\}^{k(n)} \rightarrow \{0,1\}^n, n > 0\}$ ($H : k(n) \rightarrow n$) such that, for any sufficiently large n, and for any n-input boolean circuit C with size at most n and such that $\mathbf{Pr}(C(\vec{y}) = 1) \geq 1/n$, it is required to provide *just* one "example" $\vec{y}$ for which $C(\vec{y}) = 1$, i.e., there exists at least one $\vec{x} \in \{0,1\}^{k(n)}$ such that $C(H_n(\vec{x})) = 1$.

Observe first that any PSRG is also a HSG but the *converse* is not necessarily true [1, 6]. Informally speaking, a PSRG provides a precise approximation of the value $\mathbf{Pr}(C(\vec{y}) = 1)$, i.e., the fraction of 1's in the output of C, for any "small" circuit C. Thus, if C has a large fraction of 1's in its output then the PSRG must generate an input space for which this fraction has about the same large size. On the other hand, HSG's are not required to have this property: a HSG provides, for any "small" circuit C having a "sufficiently large" number of $1's$ in its output, only a witness of the fact that C is not a *null* function. Another point of view to distinguish between PSRG's and HSG's is that Hitting Sets (i.e. the output of HSG's) have a *monotone* property not verified by the output of PSGR's: if $\mathbf{Im}(H) = \{\mathbf{Im}(H_n), n > 0\}$ is a Hitting Set then any other set collection $\mathbf{Im}(H') = \{\mathbf{Im}(H'_n), n > 0\}$, such that for any $n > 0$ $\mathbf{Im}(H_n) \subseteq \mathbf{Im}(H'_n)$, is still a Hitting Set. In general, this monotone property makes the construction of efficient HSG's easier than the construction of PSRG's [1, 6].

The design of HSG's which use "few" random bits has been a central topic in complexity theory over the last ten years, since these generators are often used to reduce the required number of random bits in many known randomized algorithms [5, 9, 4]. Moreover, in [6] and afterwards in [1], some interesting deterministic algorithms have been introduced to construct a Hitting Set for a restricted class of boolean functions. However, the main question left open by these previous works is whether quick HSG's (i.e. HSG's computable in polynomial time in the length of their output, as required for PSRG's) can replace quick PSRG's in order to efficiently derandomize *any* BPP-algorithm. In this paper, we give a positive answer to this question.

Our Results. The main technical result of this paper can be stated in the following way:

Theorem 1.1 *Let $q(n)$ be any positive polynomial function. If a quick HSG $H : k(n) \rightarrow n$ (with $k(n) = O(\log n)$) exists then it is possible to construct a deterministic algorithm A that, for any n and*

[1] notice that if $k(n) = O(\log n)$ then this condition implies that the PSRG belongs to EXP

for any circuit $C(x_1, \ldots, x_n)$ of size at most $q(n)$, computes in polynomial time in n a value $A(C)$ such that

$$|\mathbf{Pr}\,(C = 1) - A(C)| \;\leq\; \frac{1}{q(n)}\ .$$

Since approximating the fraction of 1's in the output of a linear-size boolean circuit is BPP-hard (a proof of this can be found in [8]), Theorem 1.1 directly implies the following:

Corollary 1.1 *Let $k(n) = O(\log n)$. If there exists a quick HSG $H : k(n) \to n$ then $BPP = P$.*

It is also possible to generalize the above result by considering the "price" $k(n)$ (with $k(n) = \Omega(\log n)$) of the HSG as a parameter:

Corollary 1.2 *If a quick HSG $H : k(n) \to n$ exists, then for any time-bound $t(n)$, we have*

$$BPTIME(t) \subseteq DTIME(2^{O(k(t^{O(1)}))}),$$

where $BPTIME(t)$ is the class of languages accepted by probabilistic, two-sided error Turing Machines running in time t.

Notice that this result is comparable to the one in [7, 8] stating that the existence of a quick PSRG $G : k(n) \to n$ implies $BPTIME(t) \subseteq DTIME(2^{O(k(t^2))})$.

Such results globally states that quick HSG's can be used as a new general method to derandomize probabilistic algorithm. Moreover, from the previous discussion on the differences between PSRG's and HSG's, our results can be considered as a new, stronger indication on the fact that the computational power of probabilistic machines is not much larger than that of deterministic machines [2, 7, 8].

The paper is organized as follows. In Section 2 we describe the approximation algorithm of Theorem 1.1 and we analyze its complexity and its correctness; then in Section 3 we state the consequences of this algorithm in the theory of computational complexity (i.e. Corollaries 1.1 and 1.2). Due to the lack of space, the proofs of the lemmas of Section 2 are given in the Appendix.

2 The approximation algorithm

2.1 Overall description of the algorithm

The main technical contribution of this paper is a deterministic polynomial-time algorithm that uses a quick HSG $H : k(n) = O(\log n) \to n$ in order to approximate the value $Pr(C = 1)$ for any boolean circuit $C : \{0,1\}^n \to \{0,1\}$ $(n > 0)$ having polynomial size. The approximation algorithm considers a sufficiently large *Table* T (but still polynomial in n, i.e., $|T| = h(n)$ for some polynomial $h(n)$) of inputs for C, and computes two parameters d_{min} and d_{max} where d_{min} (d_{max}) is the minimum (maximum) fraction of 1's that C generates on the set of inputs of the form $\vec{y} \oplus \vec{\alpha}$ where $\vec{y} \in T$ and $\vec{\alpha} \in \mathbf{Im}\,(H_n)$ (more precisely, the minimum (maximum) is computed with respect to $\vec{\alpha}$). We then prove the following inequalities

$$d_{min} - \epsilon(n) \;\leq\; \mathbf{Pr}\,(C = 1) \;\leq\; d_{max} + \epsilon(n)\ ,$$

where $\epsilon(n)$ is a "small" positive function which depends on H, and that we will prove to be smaller than the inverse of any polynomial function in n. From the above inequalities, it should be clear that the algorithm provides a "good" approximation if and only if the difference $D = d_{max} - d_{min}$ is "small"

(more precisely, we require this value not to be greater than $\epsilon(n)$). However, this condition is not generally satisfied and consequently a further iterative procedure must be performed. This procedure consider a polynomially bounded sequence of Tables $Y = \{T_k : k = 1, \ldots, q(n)\}$, each of them having size $h(n)$. For each T_k, it computes the parameters d^k_{min} and d^k_{max}, and checks whether the following condition is true

$$\text{“ there is at least one } k \text{ for which } D^k = d^k_{max} - d^k_{min} \leq \epsilon(n) \text{ ”} \quad (1).$$

If this Condition is verified for some k then the procedure terminates and returns the value $(d^k_{max} + d^k_{min})/2$ which is a "good" approximation of $\Pr(C = 1)$. If Condition (1) is false then the procedure performs a *compression* phase on Y whose goal is to reduce the values D^k's. Indeed, when Condition (1) is false, the non-negligible values D^k's provide a key-information about the "behavior" of C on Y that the procedure uses to codify Y (using a convenient binary coding system) into a new binary sequence having a "large" expected number of 0's. The procedure can then efficiently compress this new binary sequence in polynomial time. The obtained string will be the new sequence of Tables on which the procedure will re-check Condition (1). The algorithm applies the compression phase until either Condition (1) is satisfied or the total length of the compressed sequence of Tables is smaller than $h(n)$. In the latter case, a "failure" answer will be returned.

A suitable construction of the input Tables, based on the HSG, will guarantee the efficiency (i.e. a polynomial bound on the maximum number of compression phases of the algorithm) and the correctness (i.e. the fact that the values D^k's actually decreases and thus the procedure will never return the "failure" answer) of the algorithm.

From the above discussion, we observe that the existence of a quick HSG's is crucial for three aspects of the approximation algorithm: *i)* to compute the parameters d_{min} and d_{max}; *ii)* to guarantee the efficiency of the algorithm; *iii)* to guarantee the correctness of the algorithm. While the first two aspects are based on some combinatorial properties of HSG's, the third aspect is instead based on an interesting connection between the existence of quick HSG's and the existence of "hard" boolean functions in EXP: given any quick HSG $H : k(n) = O(\log n) \rightarrow n$, it is possible to construct a boolean function $F = \{F_n : \{0,1\}^n \rightarrow \{0,1\}, n > 0\}$ which belongs to EXP and which has, for almost all n, exponential circuit size complexity. Notice that this concept of "hard" boolean function is weaker than that introduced and adopted in [8]: informally speaking, a boolean function has exponential *hardness*, according to [8], if, besides having exponential circuit size complexity, it cannot be *approximated* by circuits of subexponential size[2]. The "hard" function F is used by the algorithm to construct the input boolean sequence of Tables Y. The correctness of the algorithm is then a consequence of the fact that if Y could be "reduced" (i.e. compressed) to a boolean string having length smaller than $h(n)$ then F would have subexponential circuit size complexity.

2.2 A first approximation of probability

The length of a string $x \in \{0,1\}^*$ is denoted as $l(x)$. Given any boolean circuit $C : \{0,1\}^n \rightarrow \{0,1\}$, we denote its circuit size as $L(C)$ (notice that any 2-input boolean function is here considered as one gate). The same notation is used for the circuit size complexity of any finite boolean function.

Let $\epsilon(n)$ be a polynomial-time computable function such that, for any $n \geq 1$, $0 < \epsilon(n) < 1$. Then the operator $H = \{H_n : \{0,1\}^{k(n)} \rightarrow \{0,1\}^n, n > 0\}$ is a *quick* $\epsilon(n)$*-Hitting Set Generator* (in short, $\epsilon(n)$-HSG) if H is computable in polynomial time in n (notice that, if $k(n) = O(\log n)$ then

[2] for "approximating a boolean function" we mean to agree on a large fraction of its inputs of size n, for almost every n - a formal definition can be found in [8]

H is computable in exponential time with respect to the length of its input), and for any boolean circuit $C : \{0,1\}^n \to \{0,1\}$, such that $L(C) \leq n$ and $\mathbf{Pr}(C = 1) \geq \epsilon(n)$, there exists (at least one) $\vec{a} \in \{0,1\}^{k(n)}$ such that $C(H_n(\vec{a})) = 1$. In which follows, we consider only *quick* HSG's, and thus we omit the term *quick*. It is not hard to prove the following properties of HSG's that we will strongly use throughout the paper.

Lemma 2.1 *If $H : k(n) \to n$ is an $\epsilon_1(n)$-HSG and, for any $n > 0$, $\epsilon_1(n) \leq \epsilon_2(n)$ then $H : k(n) \to n$ is an $\epsilon_2(n)$-HSG. Moreover, if for some constant ϵ, $0 < \epsilon < 1$, there exists an ϵ-HSG $H : k(n) \to n$ then, for any positive polynomial $p(n)$, we can construct (using H) a $p(n)^{-1}$-HSG $H' : (k(n)+O(\log n)) \to n$, in polynomial time in n.*

As described in Section 2.1, our goal is to derive a deterministic, polynomial-time algorithm that, given a circuit $C : \{0,1\}^n \to \{0,1\}$ of polynomial size, uses an ϵ-HSG $H : k(n) \to n$ (with $k(n) = O(\log n)$) and a polynomially bounded *Table* of inputs for C to approximate $\mathbf{Pr}(C = 1)$. In particular, given any ϵ-HSG H (with $0 < \epsilon < 1$), the algorithm returns a value $A(C)$ such that $|\mathbf{Pr}(C = 1) - A(C)| \leq 3/2\epsilon$.

Let $T = a_1 a_2 ... a_{l(T)}$ be a boolean sequence. The results shown in this section hold for any sequence T. The correct choice of T as the collection of input Tables for the approximation algorithm will be shown in Section 2.4. We can define the string $T(n,i) \in \{0,1\}^n$ (i.e. the input for $C : \{0,1\}^n \to \{0,1\}$) as follows

$$T(n,i) = \begin{cases} a_{(i-1)n+1} a_{(i-1)n+2} ... a_{(i-1)n+n} & \textbf{if } i * n \leq l(T) \\ a_{(i-1)n+1} ... a_{l(T)} 0 ... 0 & \textbf{if } i * n > l(T) \end{cases}$$

where $i = 1, 2, ..., m = \lceil l(T)/n \rceil$. T represents our *input Table* for C. Our next goal is to define the parameters d_{min} and d_{max} which give a first approximation of $\mathbf{Pr}(C = 1)$. Given $\vec{\alpha} \in \{0,1\}^n$, consider the function

$$Med(C, T, \vec{\alpha}) = \frac{1}{m} \sum_{i=1}^{m} C(T(n,i) \oplus \vec{\alpha}) .$$

and its "complexity" $l(C,m) = L(C) * m + c_{sym} * m$ where c_{sym} is a positive constant which will be defined in the proof of the next Lemma (see the Appendix). Let $H : k(n) \to n$ be an ϵ-HSG, we denote the prefix of length j of H_n as $H_{n,j}$ i.e. $H_{n,j} = (H_n^1, H_n^2, .., H_n^j)$. We can now define the following two parameters

$$d_{min}(C, T, H) = \min_{\gamma \in \{0,1\}^{k(l(C,m))}} Med(C, T, H_{l(C,m),n}(\gamma)) ,$$

$$d_{max}(C, T, H) = \max_{\gamma \in \{0,1\}^{k(l(f,m))}} Med(C, T, H_{l(C,m),n}(\gamma)).$$

The importance of these two parameters is given by the following inequalities (for the proof, see Lemma .1 in the Appendix).

Lemma 2.2 *If $H : k(n) \to n$ (with $k(n) = O(\log n)$) is an ϵ-HSG then*

$$d_{min}(C, T, H) - \epsilon \leq \mathbf{Pr}(C(x_1, x_2, ..., x_n) = 1) \leq d_{max}(C, T, H) + \epsilon .$$

2.3 Compression

The quality of the approximation of $\mathbf{Pr}(C = 1)$ given by Lemma 2.2 depends on the value $D = d_{max}(C, T, H) - d_{min}(C, T, H)$. However, D can be arbitrarily large and thus a further procedure must be applied in order to reduce this value when it is not sufficiently small. The procedure is based on a suitable compression of a set of input Tables. In this section, we first describe the coding technique which permits to efficiently compress a single input Table. Then we generalize this technique in order to compress a polynomial sequence of input Tables.

2.3.1 Coding and decoding the Table

Let $d_1 = d_{min}(C, T, H) = Med(C, T, \vec{\alpha}_1)$ and $d_2 = d_{max}(C, T, H) = Med(C, T, \vec{\alpha}_2)$ where $\vec{\alpha}_1 = H_{l(C,m),n}(\gamma_1)$ and $\vec{\alpha}_2 = H_{l(C,m),n}(\gamma_2)$. The j-th component of vector $\vec{a}$ will be denoted as $[\vec{a}]^j$. Since we are considering the case in which $D > 0$, without loss of generality we can assume that for index s we have $[\vec{\alpha}_1]^s \neq [\vec{\alpha}_2]^s$. Our next goal consists to show that we can codify the input Table as a boolean sequence in which the s-th component of each input string for C is always 0. Consider the operator $T^{\#} : \{1, \ldots, m\} \rightarrow \{0,1\}^n$ defined as follows

$$T^{\#}(i) = T(n, i) \oplus ([T(n, i)]^s \cdot (\vec{\alpha}_1 \oplus \vec{\alpha}_2))$$

where the operation "$\oplus$" between two boolean vectors is performed component by component and the operation "$\cdot$" is the standard scalar product. The s-th component of $T^{\#}(i)$ satisfies the following equations:

$$[T^{\#}(i)]^s = [T(n, i)]^s \oplus ([T(n, i)]^s \cdot ([\vec{\alpha}_1]^s \oplus [\vec{\alpha}_2]^s)) = [T(n, i)]^s \oplus [T(n, i)]^s \cdot 1 \ = 0 \ . \qquad (1)$$

Observe also that the set $\{T^{\#}(i) \oplus \vec{\alpha}_1, T^{\#}(i) \oplus \vec{\alpha}_2\}$ is equal to the set $\{T(n, i) \oplus \vec{\alpha}_1 \ , \ T(n, i) \oplus \vec{\alpha}_2\}$. Let $N(\sigma, \phi_1, \phi_2)$ be the number of indexes i for which $[T(n, i)]^s = \sigma$, $C(T(n, i) \oplus \vec{\alpha}_1) = \phi_1$ and $C(T(n, i) \oplus \vec{\alpha}_2) = \phi_2$. We can now introduce the function which approximates the s-th component of $T(n, i)$. Consider the 2-input boolean function $Q(z_1, z_2)$ defined as follows: $Q(x, y) = x$ if $x \neq y$; $Q(0, 0) = 1$ iff $N(1, 0, 0) \geq N(0, 0, 0)$; finally, $Q(1, 1) = 1$ iff $N(1, 1, 1) \geq N(0, 1, 1)$. Then the approximation function is

$$Z(i) \ = \ Q(C(T^{\#}(i) \oplus \vec{\alpha}_1), C(T^{\#}(i) \oplus \vec{\alpha}_2)) \ , \ i = 1, \ldots, m \ .$$

Our next goal is to estimate the number of errors generated by $Z(i)$. Let $ND(\sigma, \phi_1, \phi_2)$ be the number of indexes i such that the following conditions are satisfied:

i) $[T(n, i)]^s \oplus Z(i) = 1$ (i.e. there is an error); *ii)* $[T(n, i)]^s = \sigma$;
iii) $C(T(n, i) \oplus \vec{\alpha}_1) = \phi_1$; *iv)* $C(T(n, i) \oplus \vec{\alpha}_2) = \phi_2$.

The following Lemma (proved in the Appendix - see Lemma .2) gives an upper bound on the number of approximation errors.

Lemma 2.3

$$\sum_{(\sigma,\phi_1,\phi_2) \in \{0,1\}^3} ND(\sigma, \phi_1, \phi_2) \leq m \left(\frac{1}{2} - \frac{d_2 - d_1}{2} \right) \ .$$

In which follows, we show how to represent (i.e. codify) the input Table T using the approximation function $Z(i)$. From Lemma 2.3, we will prove that this representation will have a "large" number of 0's in its last component. The function $U(i) = [T(n,i)]^s \oplus Z(i)$, $(i = 1, 2, ..., m)$ singles out the positions in the input table in which there is an error. We thus have that

$$[T(n,i)]^s = U(i) \oplus Z(i) = U(i) \oplus Q(C(T^{\#}(i) \oplus \alpha_1), C(T^{\#}(i) \oplus \alpha_2)).$$

The new representation of the input Table is then the following string:

$$(s, \alpha_1, \alpha_2, Q, [T^{\#}]^1, ..., [T^{\#}]^{s-1}, [T^{\#}]^{s+1}, ..., [T^{\#}]^n, U) \ . \tag{2}$$

From the string in Eq. 2, we can efficiently (i.e. in polynomial time) reconstruct the s-th bit in the following way. Since

$$T^{\#}(i) = ([T^{\#}(i)]^1, \ldots, [T^{\#}(i)]^{s-1}, 0, [T^{\#}(i)]^{s+1}, \ldots, [T^{\#}(i)]^n),$$

it follows that $[T(n,i)]^s = U(i) \oplus Q(C(T^{\#}(i) \oplus \alpha_1), C(T^{\#}(i) \oplus \alpha_2))$, and consequently

$$T(n,i) \ = \ T^{\#}(i) \oplus [T(n,i)]^s \wedge (\vec{\alpha}_1 \oplus \vec{\alpha}_2) \ .$$

Observe that if we adopt the representation of T shown in Eq. 2, the string size does not decrease. However, the crucial fact for the compression phase is that, if $D = d_2 - d_1$ is not small, U contains a large number of 0's. This is an immediate consequences of Lemma 2.3: indeed, if we denote the number of 1's in a boolean string U as $|U|$, Lemma 2.3 immediately implies that:

$$|U| \ \leq \ m\left(\frac{1}{2} - \frac{d_2 - d_1}{2}\right) \ . \tag{3}$$

2.3.2 Compression of strings with a large number of 0's

Given any boolean string $V = V^*w$, let $l(V) = t$ be its length, $|V| = k$ be the number of 1's, and w be its last bit. We also define the following "counting" function: $NUM(V) = 0$ if $t = k$; moreover, $NUM(V) = NUM(V^*)$ if $w = 0$, and

$$NUM(V) = \left(\begin{array}{c} t-1 \\ k \end{array} \right) + NUM(V^*) \ \text{ otherwise}$$

.

The "compressed" version of V is then the string $(l(V), |V|, NUM(V))$. Notice that we can efficiently reconstruct V from this string. The simple procedure is the following.

- Consider $V = V^*w$, if $NUM(V) = 0$, then $V = 1^{l(V)}$.

- If $\left(\begin{array}{c} l(V)-1 \\ |V| \end{array} \right) \leq NUM(V)$, then $w = 1$ and we set: $NUM(V^*) = NUM(V) - \left(\begin{array}{c} l(V)-1 \\ |V| \end{array} \right)$, $|V^*| = |V| - 1$, and $l(V^*) = l(V) - 1$.

- If $\left(\begin{array}{c} l(V)-1 \\ |V| \end{array} \right) > NUM(V)$, then $w = 0$ and we set: $NUM(V^*) = NUM(V)$, $|V^*| = |V|$, and $l(V^*) = l(V) - 1$.

Clearly, the algorithm will halt when $l(V^*) = 0$. It is not hard to see that the number of steps of the above procedure is a linear function in the length of the input string. This binary representation

is very useful when the input string contains a large number of 0's. We will show this fact in the case of the input Table T considered in the previous section. We modify the binary representation of T defined in Eq. 2 in the following way:

$$(s, \alpha_1, \alpha_2, Q, [T^{\#}]^1, \ldots, [T^{\#}]^{s-1}, [T^{\#}]^{s+1}, \ldots, [T^{\#}]^n, l(U), |U|, NUM(U)), \tag{4}$$

where s, α_1, and α_2 have been defined in Section 2.3.1; the term $[T^{\#}]^j$ denotes the boolean sequence consisting of all j-th components of the operator $[T^{\#}]$; the function Q is represented as a string of length 4. In order to efficiently code and decode the above sequence, we make use of the following coding operators

$$\lambda(a_1 a_2 \ldots a_r) = 00 a_1 (\neg a_1) a_2 (\neg a_2) \ldots a_r (\neg a_r) 11,$$

and $\Lambda(V) = \lambda(l(V))V$, where $V \in \{0,1\}^*$ (notice that for $\Lambda(n)$, with $n \geq 0$, we mean the output of Λ on the standard binary representation of n). We also assume that $a_1 \neq 0$ if $n \neq 0$. Given an input Table T, we construct the function $T^{\#}$, Q and U (more precisely the output of these functions) as shown in Section 2.3.1. Then, using the representation (4), we define

$$comp(T) = \Lambda(l(T))\Lambda(s)\Lambda(\alpha_1)\Lambda(\alpha_2)\Lambda(Q)\Lambda([T^{\#}]^1)\ldots$$

$$\ldots\Lambda([T^{\#}]^{s-1})\Lambda([T^{\#}]^{s+1})\ldots\Lambda([T^{\#}]^n)\Lambda(l(U))\Lambda(|U|)\Lambda(NUM(U)). \tag{5}$$

It is easy to show that the coding and decoding operators can be performed in polynomial time and we can thus efficiently reconstruct the binary representation in (2) and string T (we denote the decoding operator as $comp^{-1}$). Using Lemma 2.3, it is possible to give the degree of compression achieved by the operator *comp* (the proof is given in the Appendix - Lemma .3).

Lemma 2.4 *There exists a positive constant c_1 such that*

$$l(comp(T)) \;\leq\; O(n + \log l(T)) + l(T)\left(1 - c_1 \frac{(d_2 - d_1)^2}{n}\right).$$

Notice that the degree of compression is an increasing function of $D = d_2 - d_1$.

2.3.3 Compressing more input Tables

As previously mentioned, our goal is to reduce the value $D = d_2 - d_1$. The key idea is to consider a sufficiently large sequence of input Tables and apply to them the compression operator shown in the previous section. Let Y be the boolean sequence which represents the input Tables (Y corresponds to the concatenation T of the input Tables). We consider the partition $Y = Y_1 Y_2 .. Y_r$ such that $l(Y_i) = h(n)$ (for $i = 1, .., r-1$) and $l(Y_r) \leq h(n)$, where $h(n)$ is a suitable polynomial which will be defined later. We will repeatedly transform Y using two different operators; but the obtained sequence, at each step, can always be represented as a string $W = W_1 W_2 ... W_{r(W)}$ where

$$W_i \;=\; \Lambda(N_i(W))\Lambda(l_i(W))\Lambda(q_i(W))W_i^* \;;$$

here N_i and l_i are respectively the starting point and the length of the part of Y for which W_i is the coding version, and q_i denotes the type of transformation which has generated W_i. This coding structure will be strongly used to recover the bit in a generic position in Y (see the proof of Lemma 2.6). We consider two different transformations of coding sequences.

1) a *compression* action $V = COMP(W)$, where the output sequence V is obtained as follows:

$$V = V_1 \dots V_{r(V)},\ V_i \ = \ \Lambda(N_i(W))\Lambda(l_i(W))\Lambda(q_i = 1)\Lambda(comp(W_i)),\ i = 1, \dots, r(W)\ .$$

Notice that in this case we have $r(V) = r(W)$.

2) a *concatenation* action $V = CONCAT(W)$, where $V \ = \ V_1 V_2 \dots V_{r(V)}$ is obtained as follows:

$$V_i \ = \ \Lambda(N_{2i-1}(W))\Lambda(l_{2i-1}(W) + l_{2i}(W))\Lambda(q_i = 2)\Lambda(W_{2i-1})\Lambda(W_{2i}),\ \ i = 1, \dots, \frac{r(W)}{2}.$$

Notice that in this case $r(V) \leq (r(W)/2) + 1$, and if $r(W)$ is not even then $V_{r(V)} = W_{r(V)}$.
We start the transformation process with the following sequence $W^1 \ = \ W_1^1 \dots W_{r(W^1)=r}^1$, where

$$W_i^1 \ = \ \Lambda((i-1)h(n) + 1)\Lambda(l(Y_i))\Lambda(q_i = 0)\Lambda(Y_i)\ , \qquad i = 1, 2, \dots, r\ . \tag{6}$$

The algorithm works as follows (an overall scheme of the algorithm is shown in Figure 1). Consider the ϵ-HSG H given in input and assume that the coding sequence W^t has been already generated from the input Tables. Then the algorithm checks the following condition:

$$\exists i \ : \ d_{max}(C, W_i^t, H) - d_{min}(C, W_i^t, H) \leq \epsilon\ . \tag{7}$$

If Condition 7 is verified for some index i then the algorithm returns the value

$$\frac{d_{max}(C, W_i^t, H) + d_{min}(C, W_i^t, H)}{2}\ .$$

which is a good approximation of $\mathbf{Pr}\,(C = 1)$ (see Lemma 2.2). If Condition (7) is false, the algorithm checks Condition

$$r(W^t) \ \leq \ h(n)\ . \tag{8}$$

If Condition 8 is true the algorithm returns a "failure" answer. When Condition (8) is false we apply the above described transformations to generate the coding sequence W^{t+1}. The type of the transformation to be applied is given by the following rule.

$$\text{If } \exists i : l(W_i^t) > h(n), \text{ then } W^{t+1} \ = \ COMP(W^t), \text{ otherwise } W^{t+1} \ = \ CONCAT(W^t).$$

2.4 Complexity and correctness of the algorithm

The following lemma, proved in the Appendix (see Lemma .4), is a (non trivial) consequence of Lemma 2.4 and the transformation procedure described in the previous section.

Lemma 2.5 *Let $H : k(n) \to n$ be an $\epsilon(n)$-HSG where $k(n) = O(\log n)$ and $\epsilon(n)^{-1}$ is a positive polynomial function. Choose $h(n)$ (i.e. the function of the algorithm in Condition 8) as a polynomial function such that $h(n) \geq n^3\epsilon(n)^{-2}$. Furthermore, let $h_1(n)$ be an arbitrary positive polynomial function. Then, for any n and for any circuit $C : \{0,1\}^n \to \{0,1\}$ such that $L(C) \leq h(n)$, and for any boolean sequence Y such that $l(Y) \leq h_1(n)$, every transformation step can be performed in polynomial time in n, and the maximum number $t(n)$ of transformations performed by the algorithm satisfies the following property:*

$$t(n) = O\left(\frac{1}{\epsilon(n)^2} n \log n\right)\ .$$

The following two lemmas, proved in the Appendix (see respectively Lemma .5 and Lemma .6), are used to prove the correctness of the algorithm, that is, by appropriately choosing the input sequence of Tables Y, the output will never be the "failure" answer. To this aim, beside being a simple boolean string, the sequence Y will also be considered as a finite boolean function $Y : \{0,1\}^{\lceil \log l(Y) \rceil} \rightarrow \{0,1\}$. We can thus consider its circuit complexity $L(Y(i))$.

Lemma 2.6 *With the same hypothesis and definitions of Lemma 2.5, if the algorithm returns a "failure" answer then there exists a polynomial $p(n)$ such that*

$$L(Y(i)) \ \leq \ p\left(\frac{h(n)}{\epsilon(n)}\right) \ .$$

Lemma 2.7 *If there exists an $\frac{1}{2}$-HSG $H : k(n) \rightarrow n$ with $k(n) = O(\log n)$, then we can construct (in polynomial time in n) a function $F = \{F_r : \{0,1\}^r \rightarrow \{0,1\}, r > 0\}$, which belongs to EXP, and for almost all $r > 0$ $L(F_r) \geq 2^{cr}$, for some positive constant c.*

We can now prove the final Theorem.

Theorem 2.1 *Let $q(n)$ be any positive polynomial function. If for some constant $0 < \delta < 1$ there exists a δ-HSG $H : k(n) \rightarrow n$ with $k(n) = O(\log n)$, then there exists a deterministic polynomial-time algorithm A which, for any n and for any circuit $C(x_1, x_2, ..., x_n)$ of size at most $q(n)$, computes a value $A(C)$ such that*

$$|\mathbf{Pr}\,(C(x_1, x_2, ..., x_n) = 1) - A(C)| \ \leq \ \frac{1}{q(n)} \ .$$

Sketch of the proof. Let $\epsilon(n) = \frac{1}{2}q(n)^{-1}$. By Lemma 2.1 there exists an $\epsilon(n)$-HSG $H : k(n) \rightarrow n$ $k(n) = O(\log n)$ (still denoted as H). Let $C : \{0,1\}^n \rightarrow \{0,1\}$ be some circuit of size at most $q(n)$. We apply the approximation algorithm to C. In order to avoid the "failure" answer, we have to appropriately choose the input sequence Y which represents the input Tables the algorithm will work on.

Let $h(n) = n^3\epsilon(n)^{-2} + q(n)$. From Lemma 2.6, if the algorithm returns the "failure" answer then the circuit complexity of the boolean sequence Y (here Y is considered as a finite boolean function, see the previous section) satisfies the following inequality $L(Y) \leq p(h(n))$, where p is some fixed polynomial.

However, from Lemma 2.7 we can construct a boolean function $F = \{F_r : \{0,1\}^r \rightarrow \{0,1\}, r > 0\}$, which belongs to EXP, and for almost all $r > 0$ $L(F_r) \geq 2^{cr}$, for some positive constant c. It follows that, for any $n > 0$, we can choose (in polynomial-time in n) an integer k such that

$$2^{ck-1} \ \leq \ p(h(n)) \ < \ 2^{ck} \ .$$

Since $k = O(\log n)$ we can thus construct the sequence of all values of function F_k in the standard order (i.e. $F(0,\ldots,0)\ F(0,\ldots,1)\ \ldots F(1,\ldots,1)$) in polynomial-time in n. We use this sequence as the sequence of input Tables Y in the approximation algorithm. Then, from Lemma 2.6, the algorithm cannot generate the "failure" answer since $L(Y) \geq 2^{ck} > p(h(n))$. It follows that the algorithm halts only when Condition 7 is satisfied and thus returns the value $A(C) = (d_1 + d_2)/2$ where $d_1 \leq d_2 \leq d_1 + \epsilon(n)$. Thus from Lemma 2.2 we have that

$$|A(C) - \mathbf{Pr}\,(C = 1)| \ \leq \ \frac{3}{2}\epsilon(n) \ \leq \ q(n)^{-1} \ .$$

From Lemma 2.5 and the above discussion, it is easy to see that the approximation algorithm runs in polynomial time in n.

□

begin

input: a positive polynomial $q(n)$, $n > 0$, and a boolean circuit $C : \{0,1\}^n \rightarrow \{0,1\}$ such that $L(C) \leq q(n)$.
output: a value $A(C)$ such that

$$|\mathbf{Pr}\,(C(x_1, x_2, ..., x_n) = 1) - A(C)| \;\leq\; \frac{1}{q(n)}\;.$$

Using the $\epsilon(n)$-HSG $H : k(n) \rightarrow n$ $k(n) = O(\log n)$ construct the boolean function $F = \{F_r : \{0,1\}^n \rightarrow \{0,1\}\}$, $(r \geq 0)$ defined in Lemma 2.7;

Choose k such that $2^{ck-1} \;\leq\; p(h(n)) \;<\; 2^{ck}$.

Choose the boolean sequence Y (i.e. the Input Tables) as the concatenation of all the output values of F_k in the standard order;

Consider the partition $Y = Y_1 Y_2 \ldots Y_r$ such that $l(Y_i) = h(n)$ (for $i = 1, \ldots, r-1$) and $l(Y_r) \leq h(n)$, where $h(n) = n^3 \epsilon(n)^{-2} + q(n)$;

Construct the starting configuration $W^1 \;=\; W_1^1 \ldots W^1_{r(W^1)=r}$ as described in Eq. 6;

flag := false; $t := 1$;

Repeat

for any $i = 1, \ldots, r$ **do** compute $d_{max}(C, W_i^t, H)$ and $d_{min}(C, W_i^t, H)$;
If $\exists i \;:\; d_{max}(C, W_i^t, H) - d_{min}(C, W_i^t, H) \leq \epsilon(n)$
then flag := true and return

$$\frac{d_{max}(C, W_i^t, H) + d_{min}(C, W_i^t, H)}{2};$$

else

if $r(W^t) \;\leq\; h(n)$
then flag := true and return the *failure* answer;
else
if $\exists i$ such that $l(W_i^t) > h(n)$
then $W^{t+1} := \; COMP(W^t)$;
else $W^{t+1} : = \; CONCAT(W^t)$;

$t := t + 1$;

Until flag

end.

Figure 1. The approximation algorithm

3 Complexity results

The deterministic, polynomial time algorithm shown in the previous section is able to use a HSG to solve the problem of approximating the fraction of 1's in the output of a linear-size boolean circuit. It is not hard to see that this problem is BPP-hard since, informally speaking, the algorithm can be slightly modified in order to estimate the acceptance probability of a generic BPP Turing machine on any input (a proof of this fact is implicitly given in the construction of PSRG's introduced in [8]). Thus Theorem 1.1 directly implies the following:

Corollary 3.1 *Let $k(n) = O(\log n)$. If there exists a quick HSG $H : k(n) \to n$ then $BPP = P$.*

It is also possible to generalize the above result by considering the "price" $k(n)$ (with $k(n) = \Omega(\log n)$) of the HSG as a parameter:

Corollary 3.2 *If a quick HSG $H : k(n) \to n$ exists, then for any time-bound $t(n)$, we have*

$$BPTIME(t) \subseteq DTIME(2^{O(k(t^{O(1)}))}),$$

where $BPTIME(t)$ is the class of languages accepted by probabilistic Turing Machines running in time t and with two-sided error.

The proof of the above corollary can be derived by an analogous method to that used in Section 2. The only relevant differences are the parts in which the hypothesis $k(n) = O(\log n)$ is explicitly used. There are two main issues in which this hypothesis is required and thus they must be changed according to the fact that $k(n)$ now is any computable function in $\Omega(\log n)$. The first issue is the time required to compute the parameters d_{min} and d_{max} which is now bounded by $O(2^{r(n)}p(n))$, for some polynomial p. The second issue is the "hardness" result of Lemma 2.7 that must be modified as follows. If there exists an ϵ-HSG $H : \{0,1\}^{k(n)} \to n$ (with $k(n) = \Omega(\log n)$) then there exists a a sequence of boolean functions $F_1, \ldots, F_t, \ldots$ with $L(F_t) = \Theta(t)$, and such that, for any $t > 0$, it is possible to construct F_t in time $O(l(t^{O(1)}))$.

Acknowledgemnts. We would like to thank Michael Saks, Michael Sipser, and Alexander Razborov for their patience, helpful pointers, and very interesting discussions.

References

[1] Andreev A. (1995), "The complexity of nondeterministic functions", Information and Computation, to appear.

[2] Andreev A., Clementi A., and Rolim J. (1996), "Optimal Bounds on the Approximation of Boolean Functions, with Consequences on the Concept of Hardness", XIII Annual Symposium on Theoretical Aspects of Computer Science (STACS'96), Lecture Notes in Computer Science, to appear. Also in TR95-041, ECCC 1995 Reports.

[3] Blum M., and Micali S. (1984), "How to generate cryptographically strong sequences of pseudorandom bits", SIAM J. of Computing, 13(4), 850-864.

[4] Chor B., and O. Goldreich (1989), "On the Power of Two-Point Based Sampling", J. of Complexity, 5, 96-106.

[5] Karp R., Pippenger N., and Sipser M. (1982) "Time-Randomness, Tradeoff", presented at AMS Conference on Probabilistic Computational Complexity.

[6] Linial N., Luby M., Saks M., and Zuckerman D. (1993). "Efficient construction of a small hitting set for combinatorial rectangles in high dimension", in Proc. 25th ACM STOC, 258-267.

[7] Nisan N. (1990), *Using Hard Problems to Create Pseudorandom Generators*, ACM Distinguished Dissertation, MIT Press.

[8] Nisan N., and Wigderson A. (1994), "Hardness vs Randomness", J. Comput. System Sci. 49, 149-167 (also presented at the 29th IEEE FOCS, 1988).

[9] Sipser M. (1986), "Expanders, Randomness or Time vs Space", in Proc. of 1st Conference on Structures in Complexity Theory", LNCS 223, 325-329.

[10] Wegener, I. (1987), "The complexity of finite boolean functions", Wiley-Teubner Series in Computer Science.

[11] Yao A. (1982), "Theory and applications of trapdoor functions", in 23th IEEE FOCS, 80-91.

APPENDIX: The proofs

Lemma .1 *If $H : k(n) \rightarrow n$ (with $k(n) = O(\log n)$) is an ϵ-HSG then*

$$d_{min}(C,T,H) - \epsilon \ \leq \ \mathbf{Pr}\,(C(x_1, x_2, ..., x_n) = 1) \ \leq \ d_{max}(C,T,H) + \epsilon \ .$$

Sketch of the proof. Consider the boolean function

$$g(\vec{\alpha}) \ = \ \begin{cases} 1 & \textbf{if } Med(f,T,\vec{\alpha}) < d_{min}(C,T,H) \\ & \textbf{or } Med(f,T,\vec{\alpha}) > d_{max}(C,T,H) \\ 0 & \textbf{otherwise} \end{cases} \ , \ \vec{\alpha} \in \{0,1\}^n.$$

For symmetrical functions we mean boolean functions that depend only on the number of 1's in their input string, i.e., $f(x_1, \ldots, x_n) = f(\sum x_i)$. It is easy to show that g can be represented as a composition of some symmetrical function of m variables and m functions of the following forms $f(a_1 \oplus x_1, \ldots, a_n \oplus x_n)$. Since symmetrical functions have linear circuit complexity [10], there exists a constant c_{sym} such that $L(f_{sym}((x_1, \ldots, x_n)) \leq c_{sym} n$, for any symmetrical function f_{sym} (notice that this is also the correct value of c_{sym} in the definition of $l(C,m)$). It follows that $L(g) = O(m * L(C) + c_{sym} * m) = O(l(C,m))$ and we can thus compute g in polynomial time.

If $\mathbf{Pr}\,(g(\vec{\alpha}) = 1) \geq \epsilon$ then by definition of HSG's, there exists $\gamma \in \{0,1\}^{k(l(f,m))}$ such that $g(H_{l(C,m),n}(\gamma)) = 1$. But this is a contradiction with the definitions of d_{min}, d_{max} and function g. Consequently, we have $\mathbf{Pr}\,(g(\vec{\alpha}) = 1) < \epsilon$. It is possible to prove that the expected value of $Med(C,T,\vec{\alpha})$ (here we consider Med as a random function of $\vec{\alpha}$) satisfies the following inequalities

$$d_{min}(C,T,H) - \epsilon \ \leq \ \mathbf{E}\,(Med(C,T,\vec{\alpha})) \ \leq \ d_{max}(C,T,H) + \epsilon \ .$$

Moreover, by definition of $Med(C,T,\vec{\alpha})$ we can prove that

$$\mathbf{E}\,(Med(C,T,\vec{\alpha})) = \frac{1}{m}\sum_{i=1}^{m} \mathbf{E}\,(C(T(n,i) \oplus \vec{\alpha})) = \frac{1}{m}\sum_{i=1}^{m} \mathbf{Pr}\,(C(\vec{\alpha}) = 1) = \mathbf{Pr}\,(C(\vec{\alpha}) = 1) \ .$$

Consequently, we have that $d_{min}(C,T,H) - \epsilon \leq \mathbf{Pr}\,(C(\vec{\alpha}) = 1) \leq \ d_{max}(C,T,H) + \epsilon$. □

Lemma .2

$$\sum_{(\sigma,\phi_1,\phi_2) \in \{0,1\}^3} ND(\sigma,\phi_1,\phi_2) \leq m\left(\frac{1}{2} - \frac{d_2 - d_1}{2}\right) \ .$$

Sketch of the proof. Let $N(\sigma,\phi_1,\phi_2)$ be the number of indexes i for which $[T(n,i)]^s = \sigma$, $C(T(n,i) \oplus \vec{\alpha}_1) = \phi_1$ and $C(T(n,i) \oplus \vec{\alpha}_2) = \phi_2$. Observe first that $\mathbf{Pr}\,(C(T + \alpha_1)) = d_1 m = \sum N(x,1,y)$ and $\mathbf{Pr}\,(C(T + \alpha_2)) = d_2 m = \sum N(x,y,1)$ where x and y vary on the set $\{0,1\}$. Consequently, we can write the difference between the above probabilities as $d_2 m - d_1 m = N(0,0,1) + N(1,0,1) - N(0,1,0) - N(1,1,0)$.

If $\phi_1 = \phi_2$ we have

$$ND(1,\phi_1,\phi_2) + ND(0,\phi_1,\phi_2) \ = \ N(\neg Q(\phi_1,\phi_2),\phi_1,\phi_2) \ \leq \frac{1}{2}(N(0,\phi_1,\phi_2) + N(1,\phi_1,\phi_2)).$$

Consider the case $\phi_1 = \neg\phi_2$ and $\sigma = 0$; then we have $T^{\#}(i) \ = \ T(n,i)$; consequently

$$Z(i) \ = \ Q(f(T(n,i) \oplus \vec{\alpha}_1), f(T(n,i) \oplus \vec{\alpha}_2)) = Q(\phi_1,\phi_2) \ = \phi_1,$$

and thus we obtain $ND(0,0,1) \ = \ 0$, and $ND(0,1,0) = N(0,1,0)$.

If $\phi_1 = \neg\phi_2$ and $\sigma = 1$ then $T^{\#}(i) \ = \ T(n,i) \oplus \alpha_1 \oplus \alpha_2$, and consequently

$$Z(i) \ = \ Q(f(T(n,i) \oplus \vec{\alpha}_2), f(T(n,i) \oplus \vec{\alpha}_1)) \ = \ Q(\phi_2,\phi_1) \ = \ \phi_2 \ ,$$

From the above equation, we have: $ND(1,0,1) = 0$ and $ND(1,1,0) = N(1,1,0)$. By adding each component, we obtain:

$$ND(0,0,1) + ND(0,1,0) + ND(1,0,1) + ND(1,1,0) = N(0,1,0) + N(1,1,0) \; =$$

$$= \; \frac{N(0,1,0) + N(1,1,0) + N(0,0,1) + N(1,0,1)}{2} - m\frac{d_2 - d_1}{2} \; .$$

Finally, we have that

$$\sum_{(\sigma,\phi_1,\phi_2)\in\{0,1\}^3} ND(\sigma,\phi_1,\phi_2) \leq \frac{1}{2} \sum_{(\sigma,\phi_1,\phi_2)\in\{0,1\}^3} N(\sigma,\phi_1,\phi_2) - m\frac{d_2 - d_1}{2} = m\left(\frac{1}{2} - \frac{d_2 - d_1}{2}\right) \; .$$

□

Lemma .3

$$l(comp(T)) \; \leq \; O(n + \log l(T)) + l(T)\left(1 - c_1\frac{(d_2 - d_1)^2}{n}\right) .$$

Sketch of the proof. By definition, we can easily prove the following bounds:

$$\begin{gathered} l(\Lambda(l(T))) \; = \; O(\log l(T)), \;\; l(\Lambda(s)) \; = \; O(\log l(T)), \\ l(\Lambda(\alpha_1)) \; = \; O(n), \, l(\Lambda(\alpha_2)) \; = \; O(n), \\ l(\Lambda(Q)) \; = \; O(1), \text{ and } l(\Lambda([T^{\#}]^i)) \leq (l(T)/n) + 1 \text{ for } i = 1, \ldots, s-1, s+1, \ldots, n, \\ l(\Lambda(l(U))) \; = \; O(\log l(T)) \; , \; l(\Lambda(|U|)) \; = \; O(\log l(T)). \end{gathered}$$

Concerning the function NUM, we have that: $\Lambda(NUM(U)) \; = \; O(\log l(T)) + \log(NUM(U) + 1) + 1$. Consequently we have

$$l(comp(T)) \; \leq \; O(n + \log l(T)) + (n-1)\frac{l(T)}{n} + \log(1 + NUM(U)) \; .$$

By Lemma .2 we have $l(U) \leq (l(T)/n) + 1$, and $|U| \leq l(U)(1/2 - (d_2 - d_1)/2)$. It follows that

$$NUM(U) \leq \left(\begin{array}{c} l(U) \\ |U| \end{array} \right) \leq 2^{l(U)} 2^{-c_1(d_2-d_1)^2 l(U)}.$$

Consequently,

$$l(comp(T)) \; = \; O(n + \log l(T)) + l(T)\left(1 - c_1\frac{(d_2 - d_1)^2}{n}\right) \; .$$

□

Lemma .4 *Let $H : k(n) \rightarrow n$ be an $\epsilon(n)$-HSG where where $k(n) = O(\log n)$ and $\epsilon(n)^{-1}$ is a positive polynomial function. Choose $h(n)$ (i.e. the function of the algorithm in Condition 8) as a polynomial function such that $h(n) \geq n^3\epsilon(n)^{-2}$. Furthermore, let $h_1(n)$ be an arbitrary positive polynomial function. Then, for any circuit $C : \{0,1\}^n \rightarrow \{0,1\}$ such that $L(C) \leq h(n)$, and for any boolean sequence Y such that $l(Y) \leq h_1(n)$, every transformation can be performed in polynomial time in n, and the maximum number $t(n)$ of transformations performed by the algorithm satisfies the following property:*

$$t(n) = O\left(\frac{1}{\epsilon(n)^2} n \log n\right) \; .$$

Sketch of the proof. We have already observed that the compression and the concatenation actions can be performed in polynomial time. Moreover, since $L(C) \leq h(n)$ and $L(Y) \leq h_1(n)$, then from Lemma .1 we can compute the values d_1 and d_2 in polynomial time. We now show a polynomial upper bound on the number of transformations. Consider the following function

$$M(t) \;=\; \left(r(W^t)\right)^2 \left(\frac{h(n)}{2} + \max_{1\leq i\leq r(W^t)} l(W_i^t)\right),$$

where t is an arbitrary step of algorithm. Two possible cases may arise: 1) $W^{t+1} = COMP(W^t)$ or 2) $W^{t+1} = CONCAT(W^t)$.

- In Case 1, we have that

$$r(W^{t+1}) = r(W^t), \tag{9}$$

and for any $i = 1, 2, \ldots, r(W^t)$ we have $W_i^{t+1} = \Lambda(N_i(W^t))\Lambda(l_i(W^t))\Lambda(1)\Lambda(comp(W_i^t))$.

It follows that $l(W_i^{t+1}) \;\leq\; O(\log h_1(n)) + O(1) + l(comp(W_i^t))$. From Lemma .3, we obtain

$$l(W_i^{t+1}) = O(\log n) + l(comp(W_i^t)) \leq$$

$$O(\log n) + O(n + \log l(W_i^t)) + l(W_i^t)\left(1 - c_1\frac{\epsilon(n)^2}{n}\right) \leq O(n) + l(W_i^t)\left(1 - c_1\frac{\epsilon(n)^2}{n}\right) .$$

Since a $COMP$ action has been performed, the following condition hold: $\max_{1\leq i\leq r(W^t)} l(W_i^t) \;\geq\; h(n)$; consequently, for some positive constant c_2, we have

$$\frac{h(n)}{2} + \max_{1\leq i\leq r(W^{t+1})} l(W_i^{t+1}) \leq \left(1 - c_2\frac{\epsilon(n)^2}{n}\right)\left(\frac{h(n)}{2} + \max_{1\leq i\leq r(W^t)} l(W_i^t)\right) , \tag{10}$$

Eq. (9) and Eq. (10) imply

$$M(t+1) \;\leq\; M(t)\left(1 - c_2\frac{\epsilon(n)^2}{n}\right) .$$

- In Case 2, we have that

$$r(W^{t+1}) \;=\; \lceil\frac{r(W^t)}{2}\rceil , \tag{11}$$

Moreover, for $i = 1, \ldots, \lfloor\frac{r(W)}{2}\rfloor$, we have

$$W_i^{t+1} \;=\; \Lambda(N_{2i-1}(W^t))\Lambda(l_{2i-1}(W^t) + l_{2i}(W^t))\Lambda(2)\Lambda(W_{2i-1}^t)\Lambda(W_{2i}^t) ,$$

and $l(W_i^{t+1}) \;\leq\; O(\log n) + O(1) + l(W_{2i-1}^t) + l(W_{2i}^t)$. Consequently

$$\max_{1\leq i\leq r(W^{t+1})} l(W_i^{t+1}) \leq\; O(\log n) + 2\max_{1\leq i\leq r(W^t)} l(W_i^t) ,$$

and

$$\frac{h(n)}{2} + \max_{1\leq i\leq r(W^{t+1})} l(W_i^{t+1}) \leq\; \frac{h(n)}{2} + O(\log n) + 2\max_{1\leq i\leq r(W^t)} l(W_i^t) \;\leq$$

$$\leq\; 2\left(\frac{h(n)}{2} + \max_{1\leq i\leq r(W^t)} l(W_i^t)\right) . \tag{12}$$

From Eq.s (11) and (12) it is not hard to prove that (for sufficiently large n) $M(t+1) \leq M(t)(1 - c_2\frac{\epsilon(n)^2}{n})$. Consequently, in both cases we have that

$$M(t+1) \;\leq\; M(t)\left(1 - c_2\frac{\epsilon(n)^2}{n}\right) . \tag{13}$$

Let t_0 be the last step of the algorithm, then $M(t_0-1) \geq h(n)^2$ and observe also that for Step 1 we have

$$M(1) \leq \left(\frac{h_1(n)}{h(n)}+1\right)^2 \left(\frac{3}{2}h(n)+O(1)\right) .$$

From Eq. (13), we finally have $t_0 \leq O\left(\frac{n}{\epsilon(n)^2}\log n\right)$.

□

Lemma .5 *With the same hypothesis and definitions of Lemma 2.5, if the algorithm returns a "failure" answer then there exists a polynomial $p(n)$ such that*

$$L(Y(i)) \leq p\left(\frac{h(n)}{\epsilon(n)}\right) .$$

Sketch of the proof. If the algorithm returns the "failure" answer then, from Lemma 2.5, for some step t_0 we have $r(W^{t_0}) \leq h(n)$, and $t_0 = O((n/\epsilon^2)\log n)$. Suppose we want to compute the value $Y(j)$. We first find the sequence $W_i^{t_0}$ which contains the information about $Y(j)$; since sequence $W_i^{t_0}$ has prefixes of the form $\Lambda(N)\Lambda(l)$ we need only to check $N \leq j \leq N+l-1$. This step is polynomial in $h(n)$. Suppose we are in step t and let w be the coding sequence from W^t such that $N(w) \leq j \leq N(w)+l(w)-1$. We have three possible cases

i) $w = \Lambda(N(w))\Lambda(l(w))\Lambda(0)\Lambda(v)$,
ii) $w = \Lambda(N(w))\Lambda(l(w))\Lambda(1)\Lambda(comp(v))$,
iii) $w = \Lambda(N(w))\Lambda(l(w))\Lambda(2)\Lambda(v_1)\Lambda(v_2)$.

i) The sequence w belongs to W^1 (i.e. the starting configuration) and by definition the value $Y(j)$ is the $(j-N(w))$-th bit of sequence v. *ii)* We consider the sequence w^* from W^{t-1}. We can construct it by applying $w^* = comp^{-1}(comp(v))$. *iii)* We check whether $j \leq N(v_1)+l(v_1)-1$. If this is true we choose $w^* = v_1$, otherwise we choose $w^* = v_2$.

In all cases, the complexity is polynomial in $h(n)$. Furthermore, by Lemma .4, the number t_0 of total steps (i.e. transformations) is bounded by $O(\frac{n}{\epsilon(n)^2}\log n)$. Consequently we can construct $Y(j)$ in polynomial time. This fact easily implies that the circuit complexity $L(Y(i))$ is also polynomial in $\frac{h(n)}{\epsilon(n)}$.

□

Lemma .6 *If there exists an $\frac{1}{2}$-HSG $H : k(n) \to n$ with $k(n) = O(\log n)$, then we can construct (in polynomial time in n) a function $F = \{F_r : \{0,1\}^r \to \{0,1\}, r > 0\}$, which belongs to EXP, and for almost all $r > 0$ $L(F_r) \geq 2^{cr}$, for some positive constant c.*

Sketch of the proof. Without loss of generality, we can assume that $k(n) \leq \frac{1}{2c}\log n$ for some positive constant c and $n \geq 2$. consider $n = \lceil 2^{cr} \rceil$ and consider the HSG H at n:

$$H_n(y_1, \ldots, y_{k(n)}) = (H_n^1, \ldots, H_n^n)(y_1, \ldots, y_{k(n)}),$$

where $H_n^i(y_1, \ldots, y_{k(n)})$ is a boolean function. Define $H_{n,r} = (H_n^1, \ldots, H_n^r)$ with $r \leq n$. For any $\vec{a} \in \{0,1\}^r$, consider

$$F_r(\vec{a}) = \begin{cases} 1 & \textbf{if } \forall \vec{b} \in \{0,1\}^{k(n)} : \vec{a} \neq H_{n,r}(\vec{b}) \\ 0 & \textbf{otherwise} \end{cases}$$

Since H is computable in polynomial time in n, F is computable in EXP-time with respect to the length of its input. Furthermore we have

$$\mathbf{Pr}(F_r = 1) \geq 2^{-r}(2^r - 2^{k(n)}) \geq 2^{-r}(2^r - 2^{r/2}) \geq 1 - 2^{-r/2} \geq \frac{1}{2} .$$

If $L(F_r) \leq n$ (for $r \geq 2$) then, by definition of HSG, there exists $\vec{b} \in \{0,1\}^{k(n)}$ such that $F_r(H_{n,r}(\vec{b})) = 1$; but this is a contradiction with the definition of F_r. Consequently $L(F_r) > n = \lceil 2^{cr} \rceil$.

□