Hitting Set Generators - A Different Lens into **P** vs **BPP**

Contents

1 Introduction

In class we have seen that **RP** and **BPP** are two classes of languages that capture "efficient randomized computation" (each with one-sided and two-sided errors). We have also studied the notion of pseudorandom generators (PRG) which, informally speaking, stretches short, truly random bits (called *seeds*) into long pseudorandom strings that *look* random to any deterministic poly-time machine. It is easily seen that the existence of poly-time computable PRGs $\mathcal{G} : \{0,1\}^{O(\log(n))} \to \{0,1\}^n$ with exponential stretch implies that every **BPP** algorithm can be deterministically simulated in polynomial time.

Similarly to **BPP** , there is also an intuitive generic method for derandomizing **RP** machines using a one-sided analogue of PRGs called *Hitting Set Generators (HSGs)*. Whereas the output of PRGs should look random to deterministic poly-time machines, HSGs only require a supposedly much weaker randomness property: that any poly-time deterministic machine (that accepts many inputs) accepts *at least* one of the outputs of an HSG. It is easy to see that HSGs $\mathcal{H} : \{0,1\}^{O(\log(n))} \to \{0,1\}^n$ can be used to derandomize **RP** algorithms just like PRGs and **BPP** .

In fact, it is trivial to see that PRGs themselves establish HSGs. Yet an HSG is not always a PRG since the "hitting" property of HSGs is weaker than PRG's "fooling". At first glance, it seems quite natural to believe that the one-sided randomness of HSGs is too limited to establish a strong derandomization of **BPP** .

In 1998, [ACR98] showed a somewhat surprising result that quick HSGs can replace quick PRGs to derandomize any probabilistic two-sided error algorithms. While the fact that two-sided randomness can be replaced by one-sided randomness in efficient computation is by itself remarkable, this result - combined with [ACR97] - also provided (independently of [IW97]) the first derandomization of **BPP** that relies on the existence of a certain boolean function that is hard only in the *worst-case*, as opposed to the *average-case* hardness assumption of [NW94].

To this date, hardly any feasible approach to **P** vs **BPP** other than the use of strong pseudorandom generators are known, except for the use of hitting set generators (which actually turn out to be equivalent in terms of their complexity theoretic implications.) The aim of this project is to explain on a semi-abstract level the ideas behind [ACR98] for using HSGs to derandomize two-sided error algorithms, and briefly introduce subsequent simplifcations and improvements of it that shed new light on derandomization using HSGs.

1.1 Preliminaries: Notations and Definitions

First note the notation for *Boolean operators*, denoted by $\mathcal{A} : a(n) \to b(n)$, which is a function mapping a string $x \in \{0,1\}^{a(n)}$ to a string $\alpha(x) \in \{0,1\}^{b(n)}$. The *output set* of $\mathcal{A}$ is a set of all possible length-$b(n)$ outputs on all input seeds of length-$a(n)$.

We will first start with a formal definition of PRGs and HSGs. Let $\mathcal{F}$ be a family of n-ary boolean functions mapping $\{0,1\}^n \to \{0,1\}$.

An ϵ-PRG ($\mathcal{G} : k(n) \to n$) for $\mathcal{F}$ is a $poly(n)$-time computable Boolean operator whose output set is a (multi)set of length-n strings called *ϵ-discrepancy set* defined as below.

Definition 1.1 (ϵ-Discrepancy Set). A (multi)set $S \subseteq \{0,1\}^n$ is said to be *ϵ-discrepant* for $\mathcal{F}$ if for any $f \in \mathcal{F}$,

$$|\mathbf{Pr}_{s\in S}[f(x) = 1] - \mathbf{Pr}_{x\in\{0,1\}^n}[f(x) = 1]| \leq \epsilon,$$

where the probability is taken over a uniform sample.

(Notice the "two-sided" randomness baked into the definition through sandwiching bounds on acceptance probability.)

Compare this with ϵ-HSGs (denoted by $\mathcal{H} : k(n) \to n$) for $\mathcal{F}$, which are also polynomial time computable Boolean operator whose output set is called an *ϵ-hitting set*. (Again take note of the "one-sided" randomness in the definition.)

Definition 1.2 (ϵ-Hitting Set). A (multi)set $H \subseteq \{0,1\}^n$ is said to be an *ϵ-Hitting Set* for $\mathcal{F}$ if for every $f \in \mathcal{F}$ with $\mathbf{Pr}_{x\in\{0,1\}^n}[f(x) = 1] > \epsilon$ (which is the property we refer to as *dense*), there exists an $x \in H$ such that $f(x) = 1$, or in probabilistic terms,

$$\mathbf{Pr}_{s\in H}[f(x) = 1] > 0.$$

The function $k(n)$ in both definitions is called the 'price' of the respective primitives. We are interested in the case when $k(n) = O(\log(n))$ and when the function family F is a family of polynomial-sized circuits. We say that and HSG is 'quick' or 'efficient' if it can be computed in time polynomial in n. (Unless otherwise noted, $k(n) = O(\log(n)$ and HSGs are assumed to be quick throughout this report.)

Also for completeness, we will note (without proof) the following property of the HSG which says that the specific choices of ϵ doesn't matter (so far as poly-time computability is concerned) as long as (1) ϵ is strictly positive and at most polynomial in n, and (2) the seed length is $O(\log n)$.

Fact 1.3. *1. An $\epsilon_1(n)$-HSG is also an $\epsilon_2(n)$-HSG for $\epsilon_1(n) \leq \epsilon_2(n)$. 2. Given an ϵ-HSG $\mathcal{H} : k(n) \to n$, it is possible to construct a $p(n)^{-1}$-HSG $\mathcal{H}' : k'(n) \to n$ in time polynomial in $2^{k'(n)}$ where $k'(n) = O(k(n^2p(n)))$.*

$L(f)$ will denote the circuit complexity (size) of a finite function $f : \{0,1\}^n \to \{0,1\}$.

2 Derandomization in [ACR98]

[ACR98]'s approach to conditionally establishing $\mathbf{P} = \mathbf{BPP}$ was to use the fact (from [NW94]) that approximating the fraction of 1's in the output of a boolean circuit is **BPP** -hard. The theorem proven by [ACR98] can be simplified to the following statement.

Theorem 2.1 ([ACR98]). *If a quick HSG $\mathcal{H} : k(n) \to n$ ($k(n) = \Omega(\log n)$) exists then it is possible to construct a deterministic poly-time algorithm $\mathcal{A}$ that, for any n and circuit $C(x_1, ..., x_n)$ of size at most $q(n)$ (for any polynomial q), that outputs a value $\mathcal{A}(C)$ such that $|\mathbf{Pr}_{x\in\{0,1\}^n}[C(x)] - \mathcal{A}(C)| \leq \epsilon$*

In fact, $\mathcal{A}$ actually does more than just approximating $\mathbf{Pr}(C = 1)$ - it uses the HSG to construct a ϵ-discrepancy set for C. This indicates that the existence of a quick HSG $H : k(n) \to n$ exists (for time-bounded $k(n)$) then $\mathbf{BPTIME}(t) \subseteq \mathbf{DTIME}(2^{k(t^{O(1)})})$. (Compare this with the result from [NW94] that the existence of a quick PRG $G : k(n) \to n$) implies $\mathbf{BPTIME}(t) \subseteq \mathbf{DTIME}(2^{k(t^2)})$).

The three core themes of [ACR98] are (1) "test of discrepancy with HSGs," (2) "input compression," and (3) "hitting sets as a hard function." The first step of [ACR98]'s derandomization is to use an HSG to generate a coarse (but not-too-bad) candidate for a discrepancy set - which is initially just a table of Boolean strings. The 'table representation' of this coarse candidate set then goes through a "compression" mechanism, the output of which is another table that represents a new set of strings. Under certain assumptions, a recursive application of this compression is (quite remarkably) guaranteed to produce a table that represents a finer candidate discrepancy set with the desired discrepancy factor. (All of this procedure runs in time polynomial in n.)

Later in this section we will explore how this compression is guaranteed to make the discrepancy factor better in the end. (This will involve a nice connection of HSG's combinatorial properties to some aspects of computational hardness.)

2.1 Using HSG to Generate Coarse Discrepancy Set: From One- to Two-Sided Randomness

Given a boolean circuit $C : \{0,1\}^n \to \{0,1\}$, a polynomially sized table T of inputs to C and a vector $\vec{a} \in \{0,1\}^n$, one can consider the following function p

$$p(a, C, T) = \mathbf{Pr}_{x \in T}[C(x \oplus a) = 1]$$

that computes the fraction of points in T that C accepts when T is "shifted" by a. Now let's restrict our choice of $\vec{a}$ to an ϵ-hitting set H rather than $\{0,1\}^n$ and define two vectors $a_{\min} = \arg\min_{a \in H} p(a, C, T)$ and $a_{\max} = \arg\max_{a \in H} p(a, C, T)$ (along with $p_{\min}(H, C, T) = p(a_{\min}, C, T)$ and $p_{\max}(H, C, T) = p(a_{\max}, C, T)$). (All of these values can be computed in $2^{k(n)}$ time.)

The crucial observation in [ACR98] that connects one-sided randomness to two-sided randomness is the following lemma derived from a careful and clever use of the fact that hitting set H hits every *dense* functions with small circuit complexity:

Lemma 2.2.

$$p_{\min}(H, C, T) - \epsilon \leq \boldsymbol{Pr}_{x \in \{0,1\}^n}[C(x) = 1] \leq p_{\max}(H, C, T) + \epsilon$$

To see why the lemma holds, define the following boolean function:

$$g(a) = \begin{cases} 1 & p(a, C, T) < p_{\min}(H, C, T) \text{ or } p(a, C, T) > p_{\max}(H, C, T) \\ 0 & \text{otherwise} \end{cases} .$$

$g(a)$ is a composition of functions that depend only on the number of 1's in the input, which provably has linear circuit complexity. It follows that $\mathbf{Pr}[g(a) = 1] < \epsilon$, since otherwise H will hit g and contradict the definitions of $p_{\max}, p_{\min}$, and g - hence the bounds on $\mathbb{E}_{x \in \{0,1\}^n}[p(a, C, T)] = \mathbf{Pr}[C(a) = 1]$ in the lemma.

First notice that independent of our choice of T, if $\Delta = p_{\min} - p_{\max}$ is small ($\leq \epsilon(n) = 1/q(n)$), then $A(C) = \frac{d_{\min} + d_{\max}}{2}$ is already a good approximation to $\mathbf{Pr}_{x \in \{0,1\}^n}[C(x) = 1]$ and the set $T' = \{x \oplus a | x \in T, a \in \{a_{\max}, a_{\min}\}\}$ is already an ϵ-discrepancy set.

Indeed this is not always that case, and Δ can be arbitrarily large. In order to get a well-behaved discrepancy set, we would need to decrease Δ down to $O(\epsilon)$. The procedure is described in the next section.

2.2 Input Compression: Refining the Candidate Discrepancy Set

The core of the derandomization is the iterative procedure for decreasing Δ small using the HSG $\mathcal{H}$ and the input circuit C. Consider an input table T of size at most $h(n)$ where h is a polynomial. The procedure checks the following condition:

$$\Delta = p_{\max} - p_{\min} \leq \epsilon(n), \tag{2.1}$$

which indicates that $\frac{p_{\max}+p_{\min}}{2}$ is already a good approximation. If the condition is false, then the procedure generates a polynomial-time computable *compression* of the input table T (which depends on $\mathcal{H}$ and C) and re-checks condition 2.1 on it. It runs until (1) 2.1 is satisfied and returns $A(C)$, or (2) the total length of the compressed sequence is small ($\leq h(n)$), in which case it returns "Failure." (The procedure is described in Algorithm 1, and it provably halts in polynomial run time.)

The compression is designed in a way such that the compression factor is increasing in Δ. It utilizes the fact that when Δ is large (and so the gap between the number of satisfying strings in $T \oplus a_{\max}$ and $T \oplus a_{\min}$ is large), it is easier to *guess* a single bit of a string in T, allows us a better compression. The fact that this compression is also efficiently *decompressible* also imposes a precise relationship between the degree of compression and the circuit complexity of the procedure F for generating the table T. We will use this fact to choose F with the right circuit complexity to generate T, in order to guarantee that the compression cannot go too far and the algorithm will never return "Failure."

How Compression Works: The *Bit Guessing* Game The inner workings of the compression can be explained using what can be called a "*bit-guessing*" game. Recall the definition of $a_{\max} = \arg\max_{a \in H} p(a, C, T)$ and $a_{\min} = \arg\min_{a \in H} p(a, C, T)$. For illustration, let's assume $a_{\min}$ and $a_{\max}$ differ only at the n-th coordinate, and that $[a_{\min}]_n = 0$ and $[a_{\max}]_n = 1$ (denote the remaining $n-1$ variables as $a_1, ..., a_{n-1}$).

If we are able to correctly (and efficiently) 'guess' the n-th column of a table T with high success rate based only on the values in the remaining $n-1$ columns, we may replace the n-th column $[T]_n$ with a column that indicates guessing error with a 1. This column will contain many 0's which, using a compression scheme that is efficient for columns with large number of 0's, can be compressed with a good overall compression rate.

How can we achieve "good guessing?" The clue lies in the definition of $a_{\max}$ and $a_{\min}$. $T_{\max} = T \oplus a_{\max}$ and $T_{\min} = T \oplus a_{\min}$ are "shifts" of the input table T with maximum and minimum number of satisfying assignments for C among all possible choices of a from the hitting set H. Since Δ indicates the gap between the number of satisfying assignments in the two sets, there is a strong correspondence between $x' = (x_1 \oplus a_1, x_2 \oplus a_2, ..., x_{n-1} \oplus a_{n-1}, b)$ ($b \in \{0, 1\}$) and $C(x_1 \oplus a_1, x_2 \oplus a_2, ..., x_{n-1} \oplus a_{n-1}, b)$ for large values of Δ. (Recall that $a_{\min}$ and $a_{\max}$ differ only on the last bit.) In particular, if $b = 1$ yields the circuit output 1 whereas $b = 0$ yields 0, it is more likely that $x_n = 0$ and that $x' = x \oplus a_{\max}$, and vice versa. (If the outputs are equal, a deterministic method can be applied to guarantee correctness more than half the time.)

The compression method $COMP$ in [ACR98] is based on a (slightly non-trivial) generalization of this game to the case when $a_{\max}$ and $a_{\min}$ may differ at more than one bits. The following lemma states that, as we expect, the compression rate of $COMP$ increases with Δ:

Lemma 2.3.
$$len(COMP(T)) \leq O(n + \log len(T)) + len(T)\left(1 - O(1)\frac{\Delta^2}{n}\right).$$

With this rate of compression (and a suitable use of another concatenation operator called $CONCAT$ to optimize compression), the derandomization algorithm can be proved to halt within a polynomial number of operations.

Correctness of the Derandomization Does compression actually reduce Δ in the end? This is where a subtle complexity argument plays in. Assume that over time the algorithm fails to decrease Δ before the size of the table gets very small, which causes the algorithm to return "failure". This guarantees the existence of a Boolean circuit computing the i-th bit $T(i)$ of the input table T with size polynomial in n (or more precisely, polynomial in $\frac{h(n)}{\epsilon(n)}$). Roughly speaking, this circuit will reverse each compression step (there are $poly(n)$ many), starting from the result of the compression of the table from the step before failure - which is small and can be hardwired into the circuit. By contrapositive, if there exists no Boolean circuit of size $poly(n)$ that can compute $T(i)$, this guarantees that the algorithm will never return "failure." This is reflected in the following lemma:

Lemma 2.4. *If the algorithm returns "failure" then there exists a polynomial $p(n)$ such that $L(T(i)) \leq p\left(\frac{h(n)}{\epsilon(n)}\right)$.*

Note that $T(i)$ takes in values $i \in [len(T)]$ and the size of Y is polynomial in n, so the input length is $\log(len(T)) = O(\log n)$. Denoting F^{hard} as the function computing $T(i)$, the RHS of Lemma 2.4, implies that $L(F^{hard}) \leq 2^{cr}$ for some constant c, where r is the length of the input to F^{hard}. Since we want F^{hard} to be computed in time polynomial in n, F should run in time *exponential* in its input length.

So by constructing a family of Boolean functions $F^{hard} = \{F_r^{hard} : \{0,1\}^r \to \{0,1\}\}$ where $F^{hard} \in \mathbf{E}$ and whose circuit complexity is harder than 2^{cr} for constant c mentioned in the previous paragraph, we can guarantee that the algorithm will always halt without "Failure." Quite surprisingly, we can use the HSG to generate such a hard function.

2.3 Computational Consequences of a HSG: HSG as a Hard Function

How do we generate a hard function in **E** with an exponential circuit complexity? Note that an HSG $\mathcal{H}$ has to hit *all* dense Boolean functions with *polynomial-sized* circuits. In other words, a dense function that $\mathcal{H}$ is guaranteed to miss is bound to have super-polynomial circuits.

Using this intuition, consider the following function $F_r : \{0,1\}^r \to \{0,1\}$ ($n = \lceil 2^{cr} \rceil$) which captures essentially the "complement" of the HSG $\mathcal{H} : k(n) \to n$:

$$F_r(a) = \begin{cases} 1 & \text{if } \forall b \in \{0,1\}^{k(n)}, a \neq [\mathcal{H}(b)]_{1:r} \\ 0 & \text{otherwise} \end{cases} .$$

$\mathcal{H}$ is computable in $poly(n)$-time, so F_r is computable in time exponential in its input length. Also note that since the output set H of $\mathcal{H}$ is sparse, the output set of F_r (which is conceptually the complement of H^C) is dense. Yet if $L(F_r) \leq n = \lceil 2^{cr} \rceil$, H must hit $F_r([x]_{1:r})$ (a function on n bits), contradicting the definition of F_r. Thus, $L(F_r) > n = \lceil 2^{cr} \rceil$ as desired.

3 Simplifications and Improvements of [ACR98]

Notice that the proof in [ACR98] relies on the HSG in two quite intricate ways. The first use of the HSG is literally **as a hitting set generator** for dense sets of strings decidable by small circuits. The second, more intricate use of it is **as a hard function** for generating the discrepancy sets. This is because the existence of a generator results in a computational consequence, namely that some function in EXP has large circuit complexity. In this section we briefly mention some notable subsequent simplifications and improvements of the results in [ACR98] that makes incrementally less sophisticated use of HSGs to derandomize **BPP** compared the original paper. (It should be noted, however, that [ACR98] does prove a stronger result.)

3.1 Using HSG Only as a Hitting Set: A Two-Level Recursive Use of HSG [ACRT99]

In a follow-up work, the authors of [ACR98] provided a simpler proof of [ACR98] (due to Luca Trevisan) which, rather than using the HSG as both the hitting set generator and a hard function, uses the HSG only as a hitting set at the cost of extra running time.

In particular, the authors define the discrepancy testing function $disc(f, T, H, \epsilon) = \mathbf{1}[p_{\max}(H, f, T) - p_{\min}(H, f, T) \leq \epsilon]$ based on Lemma 2.2 and observe that this test is in fact both efficiently computable (using a polynomial-sized circuit) and dense (i.e., accepts more than 3/4 of all length-n strings). Based on this observation, the HSG is used twice to generate (1) a hitting set H for f used as the input to $disc$ and (2) a hitting set $\tau = \{T_1, T_2, ..., T_{2^{k(mn)}}\}$ for the function $disc(f, \cdot, H, \epsilon) : \{0,1\}^{mn} \to \{0,1\}$ of size-m tables of length-n strings. Due to the hitting property, some $T_i \in \tau$ is guaranteed to pass $disc$ and is thus a discrepancy set for f. By testing $2^{k(mn)}$ number of T_i's each in time polynomial in mn, the overall process takes polynomial number of steps in n (for $k(n) = \log(n)$).

This proof is also parallelizable since the discrepancy test can be performed in parallel (and yields a parallel version of **P** = **BPP** , namely **NC** = **BPNC**). However, the size of the generated hitting set is larger, and due to the blow up in the size of the hitting set, the running time is $O(poly(t_{\mathcal{H}}(t_{\mathcal{H}}(poly(n)))))$ as opposed to the $O(poly(t_{\mathcal{H}}(poly(n))))$ running time of [ACR98]. This is because of a two-level iterated use of the HSG.

3.2 Explaining the Two-Level Recursion: "BPP $\subseteq$ RP $^{\text{PromiseRP}}$" [BF99]

[BF99] provides another much simpler version of the proof that the existence of efficient HSGs implies **BPP** $=$ **P**. This is based on the observation that the proof of **BPP** $\subseteq \Sigma_2 \cap \Pi_2$ (which is the exact same proof we studied in class) actually yields more: namely that **BPP** $\subseteq$ **RP** $^{\textbf{PromiseRP}}$. (**PromisePR** is required instead of **RP** for technical reasons.)

PromiseRP is the promise version of **RP** , where we restrict ourselves to inputs that are guaranteed to be rejected always or accepted with probability at least 1/2 for some probabilistic Turing machine. A little bit of thinking tells us that an efficient HSG "derandomizes" both **RP** and **PromiseRP** - at least for inputs with promises -, thus resulting in a full drandomization of **RP** $^{\textbf{PromiseRP}}$.

Recall that $\Sigma_2 =$ **NP**$^{\textbf{NP}}$ and that **RP** $\subseteq$ **NP**, so to put **BPP** in **RP** $^{\textbf{PromiseRP}}$, we need a 'for most' quantifier in place of the 'exists' quantifier for witnesses used for putting **BPP** in Σ_2. The main intuition behind this change in quantifiers is that our *shift-and-cover* proof from class already shows that the "witnesses", don't simply *exist*, but are in fact *abundant*. More formally, denoting S as the set of random coin tosses that lead to acceptance of $x \in L$, the proof already asserts (by a union bound argument) that the union over polynomially-many shifts of S covers $\{0,1\}^r$ *with high probability*:

$$\textbf{Pr}_{w_{1,\dots,p(n)} \in_{u,a,r} \{0,1\}^r} \left[\bigcup_{i=1}^{p(n)} (w_i \oplus S) \neq \{0,1\}^r \right] \leq 2^{-n}.$$

In fact, **BPP** $\subseteq$ **RP** $^{\textbf{PromiseRP}}$ yields an intuitive complexity theoretic understanding for the iterated appearance of $t_{\mathcal{H}}$ in the running time of [ACRT99] in Section 3.1 - we can use a hitting set to derandomize the **PromiseRP** oracle, which leaves us with a **RTIME**$(t_{\mathcal{H}}(poly(n)))$ machine. This probabilistic machine can then be derandomized with a hitting set for $O(t_{\mathcal{H}}(poly(n)))$ sized circuits.

3.3 Removing the Two-Level Recursion: Doubling the Dimension [GW99]

[GW99] yields yet another much simpler and elegant proof (so simple that it fits in 3 pages) of **P** $=$ **BPP** using HSGs that avoids the two-level recursive application of HSG due to the "two-quantifier" representation in [BF99]. This is done by "increasing the dimension" of the hitting set to two, so that the recursion is eliminated and the hitting set has only a one-level use.

More specifically, assume that we are given a **BPP** algorithm A for $L \in$ **BPP** , which takes inputs of length n and uses $2l(n) = poly(n)$ many random bits and errs with probability $\leq 2^{-(l+1)}$. Let the size of our hitting set H (for circuits of length-l inputs) be N_l.

The meat of the idea is the following: to decide $x \in L$, create a matrix $M \in \{0,1\}^{N_l \times N_l}$ where $M_{i,j} = A(x, e_i e_j)$ and run the following test:

$$test(x) = \begin{cases} 1 & \text{if } \forall c_{1,\dots,l} \in [N] \exists r \in [N] s.t., \wedge_{i=1}^{l} (M_{r,c_i} = 1) \\ 0 & \text{if } \forall r_{1,\dots,l} \in [N] \exists c \in [N] s.t., \wedge_{i=1}^{l} (M_{r_i,c} = 0) \end{cases}$$

In words, it decides $x \in L$ if for every choice of l columns there exists a row with all 1's in those columns, and decides $x \notin L$ if for every choice of l rows there is a column with all 0's in those rows. (It is not only combinatorially provable that both of these conditions cannot hold simultaneously, but this condition is also efficiently checkable for any Boolean matrix.)

Why does this work? In other words, why does $x \in L$ imply the existence of such a row? The idea is to change perspectives and consider the circuit whose input is the *random coin toss* - and moreoever, to *split* the random string. Consider a ($poly(n)$-sized) circuit $C_x(r)$ that takes in $2l$ strings and outputs $A(x,r)$, and let $r_1, r_2 \in \{0,1\}^l$ denote the first and last half of r. Recall that by definition,

$$\textbf{Pr}_{r \in \{0,1\}^{2l}} [C_x(r) \neq \mathbf{1}_{x \in L}] \leq 2^{-(l+1)}.$$

This exponentially small error probability guarantees that for at least half of r_1's, the decision of the circuit never errs for any value of r_2. If we choose l values of r_2 as the $l(n)$ strings in the hitting set H, then we can create yet another $poly(n)$-sized circuit $C_{x,r_{2,1,\dots,l}}$ that takes in r_1 as input and computes $C_{x,r_{2,1,\dots,l}}(r_1) = \wedge_{i=1}^{l} C_x(r_1 r_2)$. Since H hits every $poly(n)$-sized circuits, H also hits $C_{x,r_{2,1,\dots,l}}$, from which the argument can be derived.

4 Conclusion

We have studied the concept of efficient *Hitting Set Generators*, which are one-sided random analogues of Pseudo-random Generators that hit every dense sets with small circuit complexity. Despite the one-sided randomness inherent in HSGs, we explored how HSGs can replace PRGs to derandomize two-sided error **BPP** algorithms. Beginning from the original constructions of [ACR98] which used HSGs both as a generator for hitting sets and as a hard function, we have explored subsequent works of [ACRT99], [BF99], and [GW99] which incrementally made less sophisticated used of HSGs to derandomize **BPP** , sometimes at the cost of additional run-time.

As a last note on another striking fact about HSGs, despite their seemingly weak randomness property compared to PRGs, it can be proven that the necessary and sufficient conditions under which HSGs and PRGs exist turn out to be the same. [IW97] and [ISW99] show that the following three statements

1. Efficient PRGs exist
2. Efficient HSGs exist
3. There is a $L \in \mathbf{E} = \mathbf{DTIME}(2^{O(n)})$ such that L requires circuits of size $w^{\epsilon n}$ for some $\epsilon > 0$ and all but finitely many n

are, in fact, equivalent.

References

[ACR97] Alexander E Andreev, Andrea EF Clementi, and José DP Rolim. Worst-case hardness suffices for derandomization: A new method for hardness-randomness trade-offs. In *International Colloquium on Automata, Languages, and Programming*, pages 177–187. Springer, 1997.

[ACR98] Alexander E Andreev, Andrea EF Clementi, and Jose DP Rolim. A new general derandomization method. *Journal of the ACM (JACM)*, 45(1):179–213, 1998.

[ACRT99] Alexander E Andreev, Andrea EF Clementi, José DP Rolim, and Luca Trevisan. Weak random sources, hitting sets, and bpp simulations. *SIAM Journal on Computing*, 28(6):2103–2116, 1999.

[BF99] Harry Buhrman and Lance Fortnow. One-sided versus two-sided error in probabilistic computation. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 100–109. Springer, 1999.

[GW99] Oded Goldreich and Avi Wigderson. Improved derandomization of bpp using a hitting set generator. In *Randomization, Approximation, and Combinatorial Optimization. Algorithms and Techniques*, pages 131–137. Springer, 1999.

[ISW99] Russell Impagliazzo, Ronen Shaltiel, and Avi Wigderson. Near-optimal conversion of hardness into pseudo-randomness. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*, pages 181–190. IEEE, 1999.

[IW97] Russell Impagliazzo and Avi Wigderson. P= bpp if e requires exponential circuits: derandomizing the xor lemma. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 220–229. ACM, 1997.

[NW94] Noam Nisan and Avi Wigderson. Hardness vs randomness. *Journal of computer and System Sciences*, 49(2):149–167, 1994.

A [ACR98]'s Algorithm

```
Algorithm 1: Core of the Derandomization Algorithm in [ACR98]
Input  : A boolean circuit $C : \{0,1\}^n \to \{0,1\}$ such that $L(C) \leq q(n)$, $q(n)$ polynomial
Output : A value $A(C)$ such that $|\mathbf{Pr}[C(x_1, ..., x_n) = 1] - A(C)| \leq \frac{1}{q(n)}$
Construct Boolean sequence $Y = Y_1 Y_2 ... Y_r$ s.t. $len(Y_i) = h(n)$ for $i = 1, ..., r-1$ and $len(Y_r) \leq h(n)$ ;
Construct the Boolean encoding $W^1 = W^1_1 W^1_2 ... W^1_{r(W^1)=r}$ ;
while True do
    for $i = 1, ..., r$ do
        Compute $p^i_{\max}$ and $p^i_{\min}$
    end
    if $\exists k$ such that $\Delta^k = p^k_{\max} - p^k_{\min} \leq \epsilon(n)$ then
        return $\frac{p^k_{\max} + p^k_{\min}}{2}$;
    else
        if $r(W^t) \leq h(n)$ then
            return Failure;
        else
            if $\exists i$ s.t. $len(W^t) > h(n)$ then
                $W^{t+1} \leftarrow COMP(W^t)$;
            else
                $W^{t+1} \leftarrow CONCAT(W^t)$;
            end
        end
    end
end
```