CS498A Report

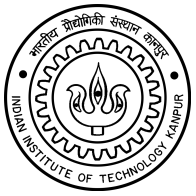
Analyzing the techniques used in some of the fundamental results in the areas of circuit lowerbounds and learning algorithms

Bachelor of Technology
in
Computer Science and Engineering

Submitted by

13652 Shaswat Chaubey

Under the guidance of
Dr. Nitin Saxena



Department of Computer Science and Engineering
Indian Institute of Technology Kanpur

Winter Semester 2016

Contents

Chapter 1

Introduction

In this report, we study three major results in the areas of natural proofs, circuit lowerbounds and psuedorandomness, as well as expanding on many of the smaller results used within. We aim to elucidate most of the lower level details of the proofs that are often skipped over, and clearly outline the higher level ideas.

1.1 Background and Recent Research

The work in circuit lowerbounds came to a halt in the late 1980s to early 1990s. Progress was made after many years due to Ryan Williams groundbreaking result showing that $NEXP \not\subset ACC0$ [2]. There has also been a very recent result that derives randomized learning algorithms from natural proofs. We look at these interesting results and many of their working parts.

1.2 Motivation

The motivation is to understand and analyze the core of why these results work, that is the fundamental ideas as well as the reasons behind certain approaches followed by the authors of the results. We also try to find other interesting problems that are closely related to these.

Chapter 2

Natural Proofs

2.1 What is a natural proof?

Natural proofs refer to a paradigm of proving lowerbounds in complexity theory, which are explicit in nature i.e. we show a property that is present in a large fraction of all circuits, but not in some circuit class of a given size, which is used to prove a lowerbound on the size of the circuits of that class to compute a circuit which has this property. Hence, at the core of a natural proof lies a natural property. Let us define a natural property in a more formal way:

Definition 2.1.1. A combinatorial property (envisioned as a set of functions, $\mathbf{C_n}$) is a Γ natural property with density δ if it satisfies the following three conditions:

- **Largeness** - $\frac{|\mathbf{C_n}|}{|\mathbf{F_n}|} \geq \delta$, where F_n is the set of all functions on inputs of size n, that is $|\mathbf{F_n}| = 2^{2^n}$.
- **Constructivity** - Membership queries to $\mathbf{C_n}$ is computable in Γ.
- **Usefulness** - For any sequence of functions f_n, if f_n has circuits of size $s(n)$ in Λ, then $f_n \notin \mathbf{C_n}$ almost everywhere.

We can give a very simple example of the following by defining the property as follows :

$f_n \in \mathbf{C_n}$, if f_n cannot be computed by a symmetric gate, that is a function which depends on the number of 1's in the input.

We see that it satisfies the following properties :

- **Largeness** - Consider $\overline{\mathbf{C_n}}$. These are the functions that can be computed by a symmetric gate. It can take only $n+1$ distinct values, depending on the number of 1's. Hence, there are only 2^{n+1} such circuits. Hence, $\frac{|\mathbf{C_n}|}{|\mathbf{F_n}|} \approx 1$

- **Constructivity** - We can manually generate all the functions that are computed by a symmetric gate, match them with the given truth table, and reject if they are the same for any of the 2^{n+1} functions. Otherwise, we accept. Since the size of the input is 2^n, this is a polynomial time algorithm with respect to it.

- **Usefulness** - By definition of the property, it is useful against the class of symmetric gates with size 1.

2.2 Limitations of a natural proof

We believe that 2^{n^ϵ}-hard psuedorandom generators exist. We show that if that assumption is true, we **cannot** use natural proofs to show the following theorem, a stronger analogue of $P \neq NP$. This uses the idea that a natural proof contains an algorithm that can be used to solve a "hard" problem. Hence, they are, in some sense, self-defeating.

Theorem 1. *If there is a P/poly-natural proof against P/poly, $H(G_k) \leq 2^{k^{o(1)}}$, for every psuedorandom function G_k such that $G_k : \{0,1\}^k \to \{0,1\}^{2k}$ in P/poly.*

Proof. Let $\mathbf{C_n}$ be the property that is P/*poly*-natural against P/*poly*. Let $G_k : \{0,1\}^k \to \{0,1\}^{2k}$ be a psuedorandom generator in P and $n = \lceil k^\epsilon \rceil$. We can split this in two parts G_0 and G_1 which output the first and second halves of the psuedorandom output respectively. The idea is to define a set of functions $f(x) \| x \in \{0,1\}^k$ in $\mathbf{F_n}$, which do not satisfy the natural property using G_0 and G_1 and then, use that fact to show that the generator does not remain psuedorandom. Define $G_y : \{0,1\}^k \to \{0,1\}^k$, where $G_y = G_{y_n} \circ \ldots \circ G_{y_1}$. We define $f(x)(y)$ as the first bit of $G_y(x)$. We see that $f(x)$ is computable by a polynomial size circuit with depth n. By definition, $f(x) \notin \mathbf{C_n}$, for any x. This means that $Pr_{x \in \{0,1\}^k}[\mathbf{C_n}(f(x)) = 1] = 0$. By largeness, $Pr_{f \in \mathbf{F_n}}[\mathbf{C_n}(f(x)) = 1] \geq 2^{-O(n)}$. We use a hybrid argument to show that this difference in probability can be used to break the generator.

We define a balanced binary tree of depth $n+1$. It has 2^n leaves, where the i^{th} leaf represents the binary representation of i as an n-bit string

$(0 \leq i \leq 2^n - 1)$. The value stored at the leaves represents the truth table of some function. We order the internal nodes with the root being numbered $2^n - 1$, it's left and right children $2^n - 3$ and $2^n - 2$, and so on. Each internal node has a random string, deonted by x_i. We define T_i as the version of the tree which has it's first i internal nodes *activated*. Note that a node can only be activated if all it's children are activated. For a leaf node, we consider it's value to be a random bit if it's parent is not activated, otherwise, if the leaf node is indexed by y, let $v_i(y)$ be the node that is it's deepest activated ancestor. We define $G_{i,y} = G_{y_n} \circ \ldots \circ G_{y_{height(v_i(y))}}$ and $f_{i,n}(y)$ as the first bit of $G_{i,y}(x_{v_i})$. This is the complete definition of the class of hybrid functions, which are indexed from 0 ($f_{0,n}$ is distributed as any random function) and $2^n - 1$ ($f_{2^n-1,n}$ is distributed as $f(x)$, where x is randomly distributed). Using the probability difference between these two extremes, we get that

$$\exists i, |\mathbf{Pr}[\mathbf{C_n}(f_{i+1,n})] - \mathbf{Pr}[\mathbf{C_n}(f_{i,n})]| \geq 2^{-O(n)}$$

Consider T_i and T_{i+1}. Fixing all the deepest roots in T_{i+1}, that is $v_i(y)$ for all y except the node v_{i+1}. Since the children of $v_{i+1}(y)$ are not fixed, we have a statistical test that distinguishes $G(x_{v_{i+1}})$ from the concatenation of randomly distributed strings x_1 and x_2 stored in it's children. This implies that $H(G) \leq 2^{O(n)} \leq 2^{O(k^\epsilon} \leq 2^{k^{o(1)}}$, using the fact that ϵ can be any positive number. Hence, proved. □

Note that we crucially exploited the constructivity algorithm to distinguish a random function from one that was not to show this result. This means that any proof technique that can be naturalized cannot be used to show proofs along the lines of $P \neq NP$, which is a much stronger statement than the one used in the theorem above. There are many newer results which use diagonalization or combinatorial arguments to show lower bounds and other results. Since those usually do not satisfy constructivity, they are not hampered by this "barrier". We see an example of such a result in the one of the following sections and see how it works. Natural proofs have also been used to derive efficient randomized learning algorithms for certain classes. We expand upon that in the last section. For other examples and results on natural proofs, please see [1].

Chapter 3

Fast algorithm for ACC0-SAT

3.1 Beigel and Zeta Transforms

The crucial component for proving that $NEXP \not\subset ACC0$, is a fast satisfiability algorithm of $ACC0 - SAT$, which is formally defined as follows :

Definition 3.1.1. Given an $ACC0$ circuit, does it output 1 on any input?

The trivial time taken is 2^n. We use two transforms to reduce this to 2^{n-n^ϵ}, for any circuit of subexponential size.

Beigel's transform is an algorithm that takes an $ACC0$ circuit, and outputs a SYM^+ circuit, which is made up of AND gates of fan-in at most $O(log^{O(1)}n)$ and the outputs of all these gates are fed into a symmetric gate at the top. The intuition behind the transform can be explained by the fact that we can replace MOD, AND and NOT gates by probabilistic polynomials of sample space $2^{O(log^3 n)}$. We then make a new circuit which takes majority vote over this circuit. Then, we use a polynomial that preserves a boolean value over powers of the modulus, which leads us tor reduce the depth, in essence pushing some computation "into" the symmetric gate allows us to reduce the depth to one, making the overall depth 2 in the end. The algorithm takes time $O(size * log(log(n)))$. For details, please see [3].

The first level represents a polynomial of degree at most $O(log^{O(1)}n)$ and the norm, which is the sum of the absolute values of the coeffiecients, to be at most $2^{O(log^{O(1)}n)}$, and the symmetric gate is a function on the value taken by this polynomial on any input to 0 or 1. This was a very fundamental result discovered in the late 1980's. The idea is to compute this transform, evaluate the symmetric function on all possible functions and then efficiently

computing the integers that are valid inputs to the symmetric gate. The only step we do not know how to do efficiently is the final step. To do this, we use a standard dynamic programming technique to compute the zeta transform of a function. The idea is that if we could figure out the number of AND gates which output 1 on any input quickly, we could calculate the input to the symmetric gate (which is just the sum of the values on the wires), which gives us the output. Hence, we note the value of the symmetric gate on all it's $2^{log^{O(1)}n}$ possible sums. To calculate the sum generated by any input, we will have to calculate $g(z)$, the sum on any binary string of length n, for all such strings. We can consider z as a set, which contains i if z_i is 1. From this point onwards in this section, we think of strings in this dual fashion. Let $f(z)$ define the number of AND gates that have exactly all the 1's in z as input. Then, $g(z)$ is simply the sum of $f(z)$ over the number of all subsets of z.

$$g(z) = \sum_{u \subseteq z} f(u)$$

Zeta transform states that we can do this in $n2^n$ time, by defining f as g_0, and defining the following recurrence:

$$g_i(z) = \begin{cases} g_{i-1}(z), & \text{if } z_i \neq 1 \\ g_{i-1}(z) + g_{i-1}(z/i), & \text{otherwise} \end{cases}$$

Observe that if the input is z, then, looking at in as a subset gives us the idea of seeing z as $\{x_1, ..., x_j\}$. Let it be partitioned into two sets A and B, such that every $x_k \leq i$ if $x_k \in A$. Then,

$$g_i(z) = \sum_{u \subseteq A} f(u \cup B)$$

From this, it follows that $g_n = g$ and we can solve the satisfiability problem on n inputs in $n2^n$ time.

3.2 The Algorithm

Consider an $ACC0$ circuit of subexponential size, 2^{n^ϵ}. Let us take l of it's inputs, and hardcode them to cover all possibilities. The new circuit, let us call it C', size is $2^{n^\epsilon + l}$, and this process takes $2^{n^\epsilon + l}$ time. Now, we convert

C' into a $SYM+$ circuit, which takes $(n^\epsilon + l) * 2^{n^\epsilon + l}$ time. This circuit takes $n-l$ inputs, and hence, can be solved using the above evaluation algorithm in $(n-l)2^{n-l}$ time. Hence, the total time taken is $(n^\epsilon + l + 1) * 2^{n^\epsilon + l} + (n-l)2^{n-l}$. We choose l to be n^ϵ, making the time of this algorithm be 2^{n-n^ϵ}, which is strictly greater than $\frac{2^n}{n^c}$, for any c. We use this algorithm as a blackbox to show that $NEXP \subseteq ACC0$ leads to a contradiction of the non-deterministic space time hierarchy.

Chapter 4

Williams' Proof

4.1 Overview

The idea behind the proof is to construct a "printer" circuit, that encodes a hard, exponential length SAT problem. We can construct an $ACC0$ circuit equivalent to this circuit, and solve it in sub-exponential time, using the power afforded to us by our assumption that $NEXP \subseteq ACC0$. This contradicts the non-deterministic space time hierarchy.

4.2 Constructing the "printer" circuit

We focus only on problems in $NTIME[2^n]$ in the proof. For any problem in $NEXP$, it can be reduced to $NTIME[2^n]$ through the universal problem of the class, $SUCCINCTBOUNDEDHALTING$, which asks the logarithm of the time taken and the description of the machine as input and simulates the machine for that many steps. It outputs yes if the machine halts and says yes on any computation path in the given amount of time. This is clearly in $NTIME[2^n]$. Hence, we have to generate a circuit that encodes an exponential length SAT problem in a circuit.

We can think of a a non-deterministic circuit that takes OR over a deterministic verifier which takes two inputs, the actual input x and the certificate I. A Cook reduction gives us a SAT problem for every circuit. What we need here is a circuit that encodes an exponential length SAT problem. We do this for the deterministic verifier, and fix the first the input x. To be able to print an exponential length SAT problem from a polynomial circuit, we need to automate the circuit in some sense. We need much more

information about the lower-level details of the circuit, such as where the head will be and so on. To do this in an easy, we turn to oblivious Turing machines, where position of head is independent of the input, but is instead a function of time. Since we have fixed the language, we have can emulate taking a step by reading the state and the value of the head, then making the transition. We give a brief description of all the processes. For even more detailed descriptions, please refer [7].

The idea behind the following emulation algorithm is the same as that of the Universal Turing machine, "If the head can't go to the tape, the tape will come to the head". Trivial simulation on an oblivious Turing machine takes $O(T(n)^2)$ time. We use the equivalence of two-tape and one-tape machines to describe a two-tape oblivious machine that simulates another machine in $T(n)log(T(n))$ time. We define the phases, parametrized on a number j. Before the start of a call to phase j, the head is always at the center position, and there are two markers at 2^{j-1}, and -2^{j-1}, defining that these are the maximal points till which we the head of the machine we are simulating may be. We can make a note of that by doubling the size of the symbols of the Turing Machine, the newer symbol corresponding to an older symbol denotes that the head of the machine being simulated is at that position. We maintain the invariant that at the end of the j^{th} phase, there would be two markers at 2^j, and -2^j. Whenever the call to phase with parameter j starts, we firstly call Compress with parameter $j-1$. Compress figures out on which side of the tape the simulated head is, and cyclically moves the 2^j length strip by 2^{j-2} such that the simulated head is moved towards the center. At this point, to remember which direction was chosen, we use the second tape as a stack. To maintain obliviousity, we move in both directions, but actually write on the TM to perfrom a cyclic shift only in the required direction. At the end of a phase, we run $EXTEND$ to pop the stacks and return the configurations at their correct postions and extend the positions of the end markers as required. Correctness is ensured because taking the steps on the simulated TM is trivial and we do not do anymore than that. We formally describe the algorithm $PHASE$ on input j formally as follows :

- If $j = 0$, simulate a step and return.
- $COMPRESS(j-1)$
- $PHASE(j-1)$
- $COMPRESS(j-1)$
- $PHASE(j-1)$

- $EXTEND(j)$

The time complexity is the number of time we simulate a step, plus the number of time taken to perform the cyclic shifts. The latter can be upper bounded by $O(2^j)$, as compress takes 2^j time in performing the cyclic shifts. We can write the recurrence of the number of steps simulated in $PHASE(j)$ and the time taken to evaluate $PHASE(j)$ as follows :

$$TIME[j] = 2TIME[j-1] + O(2^j)$$

$$STEPS[j] = 2STEPS[j-1]$$

We can see that to make $T(n)$ steps, we need $T(n)log(T(n))$ time. We can easily see that all of this can be encoded into a circuit as the size of each phase can be very strictly defined, in a fixed order. On an input, which can be thought of as the indexing of a bit of a large 2^{size} string representing the SAT instance. Then, we simply figure out which step are we taking, and output the appropriate bit from that. This can be constructed in polynomial time from x. Let's call this "printer" circuit C_x. The encoded satisfiability instance is satisfiable if and only if there exists some certificate for an accepting computation path. The recurrence of depth of this circuit is as follows :

$$DEPTH[j] = 2DEPTH[j-1] + O(1)$$

Hence, we see that the depth is $O(n)$ and it is not an $ACC0$ circuit.

4.3 Generating an equivalent ACC0 circuit

Since the "printer" circuit is not constant-depth, we give a $NEXP$ algorithm to generate a new circuit C'_x that is equivalent to the former and is in $ACC0$. We define the steps of the algorithm as follows :

- Guess a circuit D that encodes the structure of the circuit C_x and takes $log(kn^d + k)$ inputs, where $kn^d + k$ represents the number of gates in C_x. Verify that this circuit computes the type of the gate (can be encoded as a constant length string), the two gates input to it (we can assume fan-in to be at most two) by producing C_x in polynomial time. If verification fails, reject.

- Guess a circuit E which encodes the evaluation of C_x on input i. It takes two inputs i, the input to C_x and the gate index j. E is verified by using the fact that D is correct. We use D to find the number of gates that are input to j, we calculate the value of all these gates at i, and verify that they satisfy the corresponding relation, for example, if the output of the gate is NOT and $j1$, we verify that $E(i,j) \neq E(i,j1)$ and similarly construct the verifier circuit. We want to check this for all inputs, and this can be done running the $ACC0 - SAT$ algorithm on the negation of this verifier circuit. If the verifier finds E to be correct on all the inputs, then the $ACC0 - SAT$ algorithm will return unsatisfiable. We return $E(.,j\star)$ as C'_x, where $j\star$ denotes the output gate.

The algorithm takes $O(\frac{2^n}{n^c})$ time overall. The final ingredient we need to show the result is that if $NEXP$ lies in $P/poly$, then every language in $NEXP$ has witness circuits of polynomial size ([8]). Here, witness circuits refer to circuits encoding compact assignments to exponential number of variables (since the number of variables may well be exponential in number, especially in our case where the number of variables equals the size of the certificate, which is exponential) of the exponential size satisfiability problem.

4.4 The Proof

Let L be a language in $NTIME[2^n]$. Let x be any string of length n. We solve L in $O(\frac{2^n}{n^c})$ time for any x. We use the above algorithm to generate an C'_x an $ACC0$ "printer" for this problem. Since we know that there exist witness circuits to this problem, we guess a witness circuit for it, whose truth table should encode the satisfying assignment to the SAT problem encoded by C'_x. We construct an $ACC0 - SAT$ instance similarly to the second step of the above algorithm. Let the cicuit F take b length strings as input and 2^b be the number of clauses of the SAT instance. We connect $O(n)$ copies of the circuit C'_x which prints $O(n)$ bits of output as a clause. We plug in the values of the variables. If the clause is satisfied, we ouput 0, else 1. If W exists (that is, $x \in L$), then at least one computation path will exist in which we coorectly guess it. It will satisfy all the clauses, hence, we will get 0 as the output of F on all inputs. Hence, F will be unsatisfiable. This algorithm solves the problem in $O(\frac{2^n}{n^c})$ time. This leads to a contradiction in the non-deterministic time space hierarchy. Hence, $NEXP \not\subset ACC0$.

Chapter 5

Learning Algorithms from Natural Proofs

This refers to the very recent result [4] which gives a very efficient algorithm to produces an approximate circuit which given a truth table of a function in a class Λ which contains $AC^0[p]$ and has a natural proof against it. The number of inputs for which the produced circuit generates correct values on as large a fraction as we want, however, there is an expected tradeoff between the running time of the algorithm and the accuracy of the circuit. The class on which we work on should contain $AC^0[p]$ as it is the smallest class which can construct the NW generator. This result has many working parts, such as in the areas of psuedorandomn generators, learning algorithms for linear polynomials, natural proofs, error-correcting codes etc. We try to cover as much ground as possible, trying to simplify the notation, and focus on the core ideas. All the algorithms mentioned below are polynomial in the input sizes n and the inverses of the error probabilities, that is $\frac{1}{\delta}$ and $\frac{1}{\epsilon}$.

5.1 The NW Generator

We start this proof by recalling the standard properties of a Nisan Wigderson (NW) generator. It is a secure psuedorandom generator, provided that the black-box function is hard. We know that if the black-box function is not hard, then, given oracle access to it on a small number of inputs, we can construct a circuit that can predict the value of the function with probability $\frac{1}{2} + \epsilon$. This is true because if the function is not pseudorandom, we know that it is next-bit predictable. That is, there exists an algorithm P for

such that for some i (which we can guess) such that for the psuedorandom variable X, $X_i = P(X_1, ..., X_{i-1})$, with advantage ϵ. This follows from the very definitions. The key ideas behind this result is that the advantage is actually very small, and we want to amplify it to be arbitrarily large. We use an "amplified" version of the function f and certain reconstruction algorithms for reconstructing a circuit for the original value using the circuit for it. We define the Nisan-Wigderson generator's parameters, show why it is constructable in $AC^0[p]$ and outline the reconstruction algorithm for it.

Definition 5.1.1. For integer parameters n,m and $L < 2^n$, a sequence of sets $S_1, \ldots, S_L \subset [m]$, construct an NW design if for all i and $j(\neq i)$, $|S_i| = n$ and $S_i \cap S_j \leq log\ L$, then on input z, the NW generator G is defined as:

$$G(z) = f(z|S_1)...f(z|S_L)$$

where, f is a black-box function and $z|S_i$ is the string produced by only considering the values of z at the indices in S_i.

We define the above in such a way that it can be computed in $AC^0[p]$. Let F be an extension field of F_p such that $|F| = O(n)$. Let the canonical enumeration of the elements be $r_1, ..., r_|F|$ and $L = 2^l$ for some $l < n$. Consider the following polynomials over this field:

$$A_i(x) = \sum_{j=1\ to\ l} i_j x^{j-1}$$

Then, consider the following definition of the sets, $S_i = \{(r_j, A_i(r_j)|1 \leq j \leq n\}$. We see that all the properties are satisfied by this definition:

- The universe is of the size $F \times F = O(n^2)$.
- The intersection of any two sets is at most $l = log\ L$ since two distinct polynomials can agree at most at l points.
- We can also compute $z|S_i$ in $AC^0[p]$, as all the arithmetic is done in a vector space over F_p.

Since we can compute $z|S_i$, f is in a class Λ that contains $AC^0[p]$, hence, the NW generator is computable in Λ. Now, we use the fact that if f is weak, then the NW generator is not a psuedorandom generator. Hence, there exists a distinguisher circuit for it. We define the following **reconstruction** algorithm, that produces, with very high probability, a circuit C that agrees

with f with an advantage of $O(\frac{1}{L})$, using the distinguisher (this distinguisher circuit comes from the constructivity property of the natural proof, as we can envision an output of the psuedorandom generator as the truth table of a function, in the class that the natural property is useful against, that takes l bits as input) circuit :

- We pick an i randomly from $[L]$. For all $i \leq j \leq L$, fix the input to the distinguisher to be a random bit, w_j.
- For each index not in S_i, we fix it's value with a random bit.
- We build a lookup table for all the partial assignments that are generated for indices in S_j that are also in S_i and note the values taken by $f(z|S_j)$, running over all the 2^x valus of the x intersecting indices (since the intersection is guaranteed to be be small, this value is subexponential).
- The circuit made by fixing all these inputs and hardcoding the lookup table, on input x, fixes it in the place of S_i, and sends the values w_j for the first $i-1$ bits, obtained by looking up into the lookup table. If the distinguisher passes it up as random, then the circuit returns $1 - w_i$, otherwise it returns w_i.

To boost the probability, this is done multiple times, the circuits are checked by repeating this polynomial number of times, and checking the agreement of the produced circuits with f using random sampling. The best circuit is finally returned by the algorithm. This works as the random choices we make while fixing the bits tries to approximate the advantage that non-uniformity gives us.

This algorithm, gives us a circuit that has a very small advantage over simply outputting random bits. However, we would like to have a much greater advantage than this, in fact we would want no upper bound on it. To do this, we use the "amplified" version of the function f. We construct a circuit for $Amp(f)$ using the techniques above. However, we need to simulate oracle access to it for the reconstruction to work. For that, we get the constraint that $Amp(f) \in P^f$, along with the other constraint that using a circuit that approximates $Amp(f)$ with probability $1 - \delta$, we get a circuit that approximates f to any extent, the tradeoff being in the running time.

5.2 Amplifying the function

Consider the direct product version of the function f, for some constant k, and n-bit inputs $x_1, \ldots, x_k$:

$$f^k(x_1, \ldots, x_k) = f(x_1) \ldots f(x_k)$$

It has a k-bit output, which is not boolean. However, we can use a circuit for it as a middle layer in a two-step process to get us from f to $Amp(f)$.

Theorem 2. *There exists a constant c and an algorithm A such that, for constants ϵ,δ satisfying $\delta > e^{-k\epsilon/c}$, for $k \in \mathbb{N}$, if we have a circuit C' computing f^k with accuracy $1 - \delta$, A outputs a circuit with probability $\Omega(\delta)$, a circuit C that agrees with f on all but an ϵ fraction of inputs.*

We now give a description of this circuit construction algorithm. As usual, we do this polnomially many times to get a good circuit, verifying the circuit's agreement with f by cross-checking on randomly sampled inputs.

- It picks a subset of k random strings, and concatenates them to get an input for f^k.
- A random subset S of $k/2$ strings from these is chosen, and the values these strings take are fixed by taking a majority voting over the sets of values we obtain from C' for them.
- These values are hardcoded into the circuit that is being constructed.

The circuit's working can be described on an input x as follows :

- Generate a subset T of $k/2 - 1$ random strings, generate a random permutation of the set $S \cup T \cup \{x\}$ and evaluate C' on it. If the values of the strings in S agree with the values that are hardcoded in it, we output the value taken by x, otherwise we proceed.
- If the above step has been repeated $O(log(1/\epsilon)/\delta$ times, output a random bit, otherwise repeat the above step.

The idea behind the working of the above circuit is that we inject randomness in almost half of the input, believing that we know the answer

already for all but one string in the rest of the input, and hence, with high probability, one of the polynomially many random permutations generated will have an agreement on that half, and hence, with high probability, we would be able to find the value that f takes on x.

We now need to convert this non-Boolean function into a boolean function that can be used to efficiently get back this circuit for f^k. If we are working in field F, with a characteristic prime p, we map this string of k bits to an element of F. We define $(f^k)^{GL} : \{0,1\}^n k \times F^k \to F$ as follows :

$$(f^k)^{GL}(x_1, \ldots, x_k, r_1, \ldots, r_k) = \sum_{i=1}^{k} r_i (f^k(x_1, \ldots, x_k)_i) = \langle \vec{r}, f^k(x_1, \ldots, x_k) \rangle$$

Theorem 3. *There exists a probabilistic algorithm A, given $B : F^k \to F$ and $h \in F^k$ be any tuple such that*

$$Pr_{x \in F^k}[B(x) = \langle h, x \rangle] \geq \frac{1}{p} + \gamma$$

for some positive γ, then, A returns a list of size $O(\frac{1}{\gamma^2})$ such that h is on the list with probability $\frac{1}{2}$, given oracle access to B.

We use the above defined Goldrecih-Levin reconstruction, to recover $\vec{r}$ with $O(\frac{1}{2\gamma^2})$ probability. This algorithm is much more complicated than the direct product reconstruction one, hence, we simply use the above theorem, without going into the lower-level details. For more details, refer to [5].

We note that the field F is non-boolean for $p \neq 2$, hence, we have to deal with two cases.

5.2.1 When the field has characteristic 2

Since the field is Boolean, the function $(f^k)^{GL}$ is a Boolean function. We define $Amp(f)$ to be $(f^k)^{GL}$, setting k to be $\lceil \frac{3c}{\epsilon} ln \frac{1}{\gamma} \rceil$, where c is the constant in the DP reconstruction. Using the natural property against $AC^0[2]$ and the NW reconstruction algorithm, we can get a circuit H that approximates $(f^k)^{GL}$ on $1/2+\gamma$ fraction of inputs. We show that we can construct a circuit D approximating f^k using this circuit and the GL reconstruction algorithm defined above as subroutines. On input $x \in \{0,1\}^{nk}$, D does the following :

- Construct B as $H(x,r)$, for some r, which we try to recover using the GL reconstruction algorithm, using the parameter $\frac{\gamma}{2}$.
- Output a random string from the list.

Theorem 4. *The above algorithm constructs a circuit for f^k which correctly computes it for $\Omega(\gamma^3)$ fraction of inputs.*

Proof. We know that $Pr_{x,r}[H(x,r) = (f^k)^{GL}] \geq \frac{1}{2} + \gamma$. We claim that at least $frac\gamma 2$ fraction of all x agree with $(f^k)^{GL}$ on $\frac{1+\gamma}{2}$ fraction of all r. Let us assume this is wrong. Then,

$$Pr_x[Pr_r[H(x,r) = C(x,r)] \geq \frac{1+\gamma}{2}] < \frac{\gamma}{2}$$

Let A denote the event above. Consider $Pr_{x,r}[H(x,r) = (f^k)^{GL}] = Pr_x[Pr_r[H(x,r) = C(x,r)]|A] * Pr[A] + Pr_x[Pr_r[H(x,r) = C(x,r)]|\overline{A}] * Pr[\overline{A}] < \frac{\gamma}{2} + \frac{1+\gamma}{2} < \frac{1}{2} + \gamma$. But, $Pr_{x,r}[H(x,r) = (f^k)^{GL}] \geq \frac{1}{2} + \gamma$. Hence, our assumption was wrong, and $Pr_x[Pr_r[H(x,r) = C(x,r)] \geq \frac{1+\gamma}{2}] \geq \frac{\gamma}{2}$. For these strings, the probability of selecting a correct string from the list is $\Omega(\gamma^2)$. This shows that the overall probability of being correct is greater than equal to $\Omega(\gamma^3)$. □

Using DP reconstruction on the circuit for f^k, we get a circuit that approximates f for any arbitrary error value ϵ.

5.2.2 When the field has an odd prime characteristic

In this case, the field is non-Boolean, hence, we first have to reduce $(f^k)^{GL}$ to a Boolean function h first. We put h into the black-box generator to get a circuit for it, and then, transform it into a circuit for $(f^k)^{GL}$. Then, we follow steps analoguous to that of the previous case. To get a Boolean circuit, while still preserving information about $(f^k)^{GL}$ in it uses a few theorems from psuedorandomness theory and the Von Neumann trick. For the former, as we do not go into the lower-level details as far as the constants are concerned, please refresh your memory using [6] if needed.

The Von Neumann trick is a standard trick, used to convert a biased coin to an unbiased one. We define $E^{vN} : (F^2)^t \to \{0,1\}$ as follows : $E^{vN}((a_1,b_1)\ldots(a_t,b_t))$ is 0 if the smallest unequal pair has $a_i < b_i$, otherwise it is 1. We see that for every $r \in (F^2)^t$ for which it is 0, there exists a

symmetric r_1 for which it is 1, simply by swapping the coordinates of a_i and b_i for all i. The only inputs that are invariant under this are those which have all their coordinates equal. Those are p^{-t} in number. Hence, we get the following:

$$Pr_{r\in(F^2)^t}[E^{vN}(r)=1] - Pr_{r\in(F^2)^t}[E^{vN}(r)=0] = p^{-t}$$

Hence, E^{vN} is very close to a random distribution. We define $h : (F^2)^t \to \{0,1\}$ as follows:

$$h((a_1,b_1)\dots(a_t,b_t)) = E^{vN}(((f^k)^{GL}(a_1),(f^k)^{GL}(b_1))\dots)$$

We claim that from a circuit that computes h approximately on $1/2+\mu$ fraction of inputs, we can generate a circuit that generates $(f^k)^{GL}$ on $\frac{1}{p}+\gamma$ fraction of inputs, where $\gamma \geq \Omega(\frac{-\mu}{log(\mu)})$. To show this, we use results from psuedorandomness. We have a circuit that computes h with some advantage. Hence, we have a predictor for h that is better than random. We have to show that this implies a predictor for g. To show this, we instead prove the contrapositive. We say that two distributions S and T are computationally (t,s)-indistinguishable if no circuit of size lesser than s can differentiate between them with probability greater than t. Let us suppose that we cannot build a predictor for $(f^k)^{GL}$ with advantage greater than γ. Then, it's output would be indistinguishable from a random element of the field F, for any circuit of size lesser than s, with a probability difference greater than 2γ. Using a hybrid argument and then applying E^{vN} gives us that h is $(4t\gamma + p^{-t}, \Omega(s/t) - poly(t))$-indistinguishable from random functions. But, we have a predictor for h. This implies that using the appropriate reconstructions for each step, we can get a circuit for $(f^k)^{GL}$ and then proceed as in the above case. To do so, we have to ensure that the advantage μ is greater than $4t\gamma + p^{-t}$. Putting $t = O(log\frac{1}{\mu})$ gives us that $\gamma \geq O(\frac{-\mu}{log(\mu)})$.

Chapter 6

Conclusion and Future Work

We see that these proofs assimilate and build over a decade of ideas in complexity theory. Future work to push the frontier can be to show natural lower bounds for other classes like $SYM+$ and $ACC0$. We know that there are $TC0$ circuits of depth-4 which compute a psuedo-random generator based on the decisional Diffie-Helmann assumption, which we believe to be secure, hence, for classes equally powerful or stronger than it, we do not believe a natural proof exists, as it would violate the assumption. We know that no PRG exists in depth-2 $TC0$, and it is unknown whether one exists in depth-3 $TC0$. That could be another line of attack.

Acknowledgments

I thank Prof. Nitin Saxena for mentoring me, keeping me in line and making sure that I did not abuse the notation too much. He helped me learn to focus on the core ideas, and completely understanding the results, without leaving anything to chance, or as a blackbox. It was a fun learning experience. I also thank Prof. Rajat Mittal, who came to both of my talks, asked all the crucial questions and gave me advice on how to structure my thoughts.

Shaswat Chaubey

November, 2016
IIT Kanpur

References

[1] Razborov and Rudich, Natural Proofs

[2] Williams, Non-Uniform ACC Circuit Lower Bounds

[3] Begiel and Tarui, On ACC

[4] Carmosiono, Imagliazzo and Kabanets, Learning Algorithms from Natural Proofs

[5] Goldreich et. al, Learning polynomials with queries

[6] Salil Vadhan Psuedorandom Generators
people.seas.harvard.edu/~salil/pseudorandomness/prgs.pdf

[7] Pippenger et. al Relations Among Complexity Measures
www.ccs.neu.edu/home/viola/classes/papers/
PippengerF-Oblivious.pdf

[8] Impagliazzo et. al, In search of an easy witness