Hardness vs. Randomness

March 15, 2019

1 Introduction

This project will investigate the power of randomness in computation, with a focus on pseudo-random generators. We've seen in class that randomness increases the efficiency of many algorithms in terms of space and time, and can be considered a type of computational resource in itself. In two papers from 1997 and 2001, Russell Impagliazzo and Avi Wigderson dispelled "the intuition that randomness is a resource basically incomparable to time" [IW91] by showing that randomness can essentially be exchanged for time complexity. Particularly, that one of the following must be the case: either all BPP algorithms can be effectively simulated deterministically, or all EXP problems can be efficiently computed with probabilistic algorithms.

Today, it is widely believed that the former is the case — that P = BPP because of our ability to create pseudorandom generators from hard functions. We believe that there exist functions that are only computable by exponential-size circuits, and Impagliazzo and Wigderson showed that P = BPP is a consequence of this. In this project, we will follow the development of pseudorandom generators, starting from an early result in cryptography which constructed a PRG from a hard function. Then we will examine the relationship between the following 3 claims [TV07]:

- there are problems with high worst-case circuit complexity,
- there are problems with high average-case circuit complexity,
- there are good pseudorandom generators that can simulate BPP in subexponential time.

In addition, we will also explore some of the limitations of pseudorandom generators and talk about some open problems relating to randomness. If P=BPP, should that mean that problems in BPP are computable by deterministic polynomial-time algorithms *without* simulating randomness? Is there one single pseudorandom generator that can substitute for randomness in all BPP algorithms? We will discuss these further-reaching questions at the end.

2 Unpredictability

The idea of comparing hardness to randomness first came from cryptography. One-way functions — functions that are easy to compute but hard to invert — form the backbone of many cryptographic tools, such as ciphers and hash functions. One-way functions and permutations, unlike the more general NP-hard problems, are hard in the *average* case rather than just the worst case. A 1987 paper by Andrew Yao first proved that any cryptographically-secure one-way permutation can be built upon to create a pseudorandom generator that can efficiently simulate any probabilistic algorithm. This result is known as the XOR lemma. To state the XOR lemma, we will need to define a few terms:

For a function $f : \{0,1\}^m \to \{0,1\}^\ell$ for some integers m and ℓ, $S(f)$ is the worst-case circuit complexity of f, the minimum number of gates for a circuit that matches f on every input. We define the success $SUC_s^\pi(f)$ for a distribution and a circuit-size as follows:

Let π be an arbitrary distribution on $\{0,1\}^m$, and s be an integer. $SUC_s^\pi(f)$ is the maximum, over all Boolean circuits C of size at most s, of $Pr_\pi[C(x) = f(x)]$. Intuitively, $SUC_s^\pi(f)$ is the "closeness" of the closest approximation of f by a circuit of size s. In the case of functions where the range is $\{0,1\}$, we also define the advantage $ADV_s^\pi(f) = 2SUC_s^\pi(f) - 1$. Intuitively, this measures how much more accurate the circuit is at guessing $f(x)$ than a fair coin.

We use $SUC_s(f)$, without mentioning π, when π is assumed to be the uniform distribution.

We will also use the following notation: Define $g^{\oplus(k)} : (\{0,1\}^n)^k \to \{0,1\} := g(x_1) \oplus g(x_2) \oplus ...g(x_k)$. These are k random inputs to g. Yao's lemma states the following:

Choose ϵ and δ in $(0,1)$, $\epsilon < \delta$, and choose $k = O(-\log \epsilon/\delta)$. Let s be a circuit size and g be a function with $SUC_s(g) \leq 1 - \delta$. Let $s' = s(\epsilon\delta)^{O(1)}$. Then $ADV_{s'}(g^{\oplus(k)}) \leq \epsilon$.

Impagliazzo and Wigderson provide a useful intuition for this lemma. We can think of the value of $g(x)$, for a random x, as being completely predictable with probability $1 - 2\delta$ and being a random coinflip otherwise. So we have some small circuit C (of size s) that computes the "complete prediction", that is right for certain with probability $1 - 2\delta$, right by chance with probability δ, and wrong by chance with probability δ. If you use C to guess the XOR of $g(x_i)$ for k independent x_i, your guess is better than random only if all k trials are "completely predictable". If any one trial is a coinflip, then C has no advantage in predicting the XOR, and may as well be a coinflip itself! We chose C to be the circuit with the highest success probability of all circuits of size $\leq s$; thus, any circuit smaller than s has no more than $(1 - 2\delta)^k$ advantage in predicting the XOR.

Yao's lemma provides a naive pseudorandom generator from an arbitrary one-way permutation. It just needs g such that $g(x)$ is computable by a small circuit, but a small circuit approximating $g^{-1}(x)$ must fail with some constant probability δ. The XOR construction

inflates this failure probability. This technique, similar in principle to BPP error reduction, is known as "hardness amplification."

3 Pseudorandom generators from average-case hardness

The trouble with Yao's lemma is that the pseudorandom generator requires k independent inputs to the one-way permutation, and therefore kn random bits. In 1997, Impagliazzo and Wigderson built upon Yao's lemma, removing the need for the k inputs to be independent. They generalize the XOR construction $g^{\oplus k} := g(x_1) \oplus g(x_2) \oplus ...g(x_k)$ to the direct product construction $g^k := \{g(x_1), g(x_2), ..., g(x_k)\}$. The lemma still applies. Using this construction, they introduce the *direct-product generator*:

"Let G be a function from $\{0,1\}^m$ to $(\{0,1\}^n)^k$. G is an $(s, s', \epsilon, \delta)$ direct product generator, if for every boolean function g with $SUC_s(g) \leq 1-\delta$, we have $SUC_{s'}(g^k \circ G) \leq \epsilon$" [IW97].

Essentially, a direct-product generator G takes m bits of input and uses them to generate k n-bit inputs. These inputs are not independent, but still have the property that the direct product of their images, with respect to any hard function g, are still ϵ-indistinguishable to small circuits. For the purpose of applying the direct-product construction, the inputs behave as though they are independent.

We also need an extension of the definition to circuit families, where the parameters are functions of n. A circuit family G is a direct product generator if for every n, $G_n : \{0,1\}^{m(n)} \rightarrow (\{0,1\}^n)^{k(n)}$ is a $(s(n), s'(n), \delta(n), \epsilon(n))$ direct-product generator. The main portion of the paper serves to prove the existence of efficient direct-product generators that can fool circuits of arbitrary size:

"For any $0 < \gamma < 1$, there is a $0 < \gamma' < 1$ and a $c > 0$ so that there is a polynomial time $G : \{0,1\}^{cn} \rightarrow (\{0,1\}^n)^n$ which is a $(2^{\gamma n}, 2^{\gamma' n}, 2^{-\gamma' n}, 1/3)$ direct product generator" [IW97].

This is a direct-product generator that inflates cn bits of randomness to n "random" inputs of size n. This is a big improvement over the original XOR construction! The proof is very long, but it lends itself to both a method for constructing direct-product generators (which we will discuss in the next section) and the well-known "hardness vs. randomness" tradeoff:

If there is a Boolean function $f \in E$ with $S(f_n) = 2^{\Omega(n)}$, then BPP = P.

This is why it is so widely believed that P = BPP. If there are hard circuits, then they can be used to construct generators that produce strings that are indistinguishable from randomness. This resolves one of the relationships between the 3 claims in the introduction. We can now see that if there are problems with high average-case circuit complexity, then

there are good pseudorandom generators.

4 Examples of pseudorandom generators

As mentioned in the previous section, the proof of existence of direct-product generators is long, but it provides an interesting way tot construct direct-product generators from simple generators with weaker properties. We will outline their example in order to look at some well-known PRGs and the resulting direct-product generator.

4.1 Expander walks

This is a pseudorandom generator that takes $n + 4k$ random bits, and outputs kn pseudorandom bits for some chosen k:

"Fix an explicit expander graph G on vertex set $\{0,1\}^n$, of constant degree (say 16) and small second eigenvalue (say < 8). Let the generator $EW : \{0,1\}^n \times [16]^{k-1} \to (\{0,1\}^n)^k$ be defined by $EW(v, \overline{d}) = (v_1, v_2, ..., v_k)$ where $v_1 = v$ and $v_i + 1$ is the d_ith neighbour of v_i in G."[IW97]

This algorithm is quite simple — you choose a random starting vertex and a random series of edges to follow, and you output the path of k vertices. Each vertex has degree ≤ 16, so each edge number can be expressed in only 4 bits. We can see how this generator converts small random inputs, the edge numbers, into large random outputs, the vertex numbers.

4.2 Nearly-disjoint subsets

This is a pseudorandom generator that takes an m-bit random string r and outputs kn pseudorandom bits for some chosen k.

"Let $\Sigma = \{S_1, ..., S_k\}$ be a family of subsets of $[m]$ of size n. We say Σ is γ-disjoint if $|S_i \cap S_j| \leq \gamma n$ for each $i \neq j$. Let $r \in \{0,1\}^m$ and $S = \{s_1 < s_2 < ...s_n\} \subseteq [m]$. Then let the restriction of r to S, $r|_S$, be the n-bit string $r|_S = r_{s_1} r_{s_2} ... r_{s_n}$. Then, for a γ-disjoint Σ, $ND^\Sigma : \{0,1\}^m \to (\{0,1\}^n)^k$ is defined by: $ND^\Sigma(r) = r|_{S_1}, r|_{S_2}, ..., r|_{S_k}$" [IW97].

Impagliazzo and Wigderson prove that for $k = O(n)$ and $m = 2n/\gamma$, the k subsets of $[m]$ can be probabilistically generated in polynomial time from $O(m)$ random bits. With high probability, they will be γ-disjoint. Given these sets, this algorithm is also quite simple — the sets are indices in r. For each element in each set, you record the bit in r at the position indicated by that element. The output is the concatenation of these kn bits.

4.3 Direct-product (XOR) generator

Impagliazzo and Wigderson prove that you can build a direct-product generator by taking the componentwise XOR of the outputs from the above algorithms. What this tells us is that these strong generators — strong enough to fool the direct-product analogue of Yao's XOR lemma — are not hard to construct.

5 An even stronger implication

We now know that the existence of a hard circuit implies P = BPP. But in 2001, Impagliazzo and Wigderson proved an even stronger result that characterizes the relationship between hardness and randomness: "Either there is a non-trivial deterministic simulation of BPP, or BPP = EXP" [IW91]. We will flesh this out in a little more detail in order to grasp how surprising it is. We begin with the relevant definitions:

For a complexity class C, $i.o. - C$ is the class of functions that agree with a function in C for all inputs of length n for infinitely many n.

Let T and ϵ be functions of n. $HeurTIME_{\epsilon(n)}(T(n))$ is the class of pairs (f, μ) of functions $f : 0,1^* \to \{0,1\}$ and probability ensembles μ so that there is an algorithm $A(x)$ running in deterministic time $T(|x|)$ so that $\forall n$, for $x \in_{\mu_n} \{0,1\}^n$, $\Pr[A(x) \neq f(x)] < \epsilon(n)$.

If $(f, \mu) \in HeurTIME_{\epsilon(n)}(T(n))$ for some polynomially-sampleable μ, then we may just say that $f \in HeurTIME_{\epsilon(n)}(T(n))$.

Their notation is a little strange for the sake of being maximally general, so we'll clear it up. A "probability ensemble" is basically just an ordered tuple of random variables; they use a probability ensemble instead of a single distribution here because the distribution that x comes from will depend on n. In the uniform ensemble, for instance, each μ_n is the uniform distribution over strings of length n. A simple, but not perfectly general way of understanding $HeurTIME_{\epsilon(n)}(T(n))$ is just with respect to the uniform ensemble: a function f is in $HeurTIME_{\epsilon(n)}(T(n))$ if there is a $T(|x|)$-time algorithm A such that for all n, $A(x) \neq f(x)$ with probability $\leq \epsilon(n)$.

Their surprising result is that exactly one of the following holds:

1. BPP = EXP
2. BPP $\subseteq i.o. - HeurTIME_{1/n^c}(2^{n^\delta})$ for every $\delta > 0$ and $c > 0$.

So, in a world where BPP $\neq$ EXP, we can make this claim. Take any function $f \in$ BPP. We want to compute f deterministically with error $1/n^c$ in time 2^{n^δ}, for whatever c and δ we want. Then we can find an algorithm that does this, for infinitely many input lengths n. This is a considerably weaker statement than P = BPP; it doesn't claim that A will work for every input length, the time complexity is subexponential but not polynomial, and it allows A to fail with small probability. But this weaker claim is a result of a much weaker

assumption — only that BPP $\neq$ EXP. We've now resolved another relationship between the three claims in the introduction — if there are problems with high *worst-case* circuit complexity, then there are good pseudorandom generators.

6 Limitations and further questions

As it turns out, the study of randomness over the last few decades has revealed that the three claims are in fact equivalent. There are problems of high average-case complexity iff there are problems of high worst-case complexity iff there are good pseudorandom generators for fooling small circuits. A somewhat vague takeaway from all of this is that "randomness", as a computational resource, is some form of time complexity. Maybe it's just a different way of understanding the same concept — random values, with reference to some algorithm A, are just values that A can't predict due to time complexity restrictions. Or it's the other way around, and true randomness is extremely powerful and nondeterministic computations are just our way of understanding randomized computation in a limit as the advantage approaches 1. We lean toward the former, hence the fact that P = BPP is widely believed and BPP = EXP and BPP = NP are not.

Even under the assumption that P = BPP, there are still some unanswered questions about the problems we intuitively consider to be in BPP $\setminus$ P. For instance, should there be ways to solve BPP problems deterministically in polynomial time *without* simulating randomness? In the proof of BPP $\subseteq$ P/poly, we saw the use of "advice strings" that work for all input of length n, that can be hardcoded into circuits. Perhaps there is a similar principle for uniform computation — that deciders for BPP languages correspond to "advice functions" that generate good advice strings for inputs of arbitrary length. Certainly there are such functions, as the set of pairs $(n, advice_n)$ for each natural number n is a function. The question is whether these functions are computable, particularly in polynomial time. If it could be proven that such functions exist for all languages in BPP, that would prove P = BPP without needing to prove the existence of hard circuits to use as generators. Whether or not you believe such functions exist will depend on whether you believe "unknown-ness," indistinguishability with respect to the algorithm in question, is a fundamental computational resource.

References

[1] Vadhan, Salil.: Pseudorandomness survey/monograph. https://people.seas.harvard.edu/ salil/pseudorandomness/

[2] Trevisan, L. and Vadhan, S. comput. complex. (2007) 16: 331. https://doi.org/10.1007/s00037-007-0233-x

[3] Impagliazzo, Russell and Avi Wigderson. P = BPP if E Requires Exponential Circuits: Derandomizing the XOR Lemma. STOC (1997).

[4] R. Impagliazzo and A. Wigderson, Randomness vs time: Derandomizationunder a uniform assumption,Journal of Computer and System Sciences,vol. 63, no. 4, pp. 672-688, 2001