Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Εφαρμοσμενων Μαθηματικων και Φυσικων Επιστημων
Τομεας Μαθηματικων

Ντετερμινιστική Προσομοίωση Πιθανοτικών Κλάσεων Πολυπλοκότητας υπό Ομοιόμορφες Συνθήκες

Διπλωματικη Εργασια

του

Αντώνη Αντωνόπουλου

Επιβλέπων: Ευστάθιος Ζάχος
Καθηγητής Ε.Μ.Π.

Εργαστηριο Λογικης και Επιστημης Υπολογισμων
Αθήνα, Ιούλιος 2013

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Εφαρμοσμένων Μαθηματικών και Φυσικών Επιστημών
Τομέας Μαθηματικών
Εργαστήριο Λογικής και Επιστήμης Υπολογισμών

Ντετερμινιστική Προσομοίωση Πιθανοτικών Κλάσεων Πολυπλοκότητας υπό Ομοιόμορφες Συνθήκες

Διπλωματική Εργασία

του

ΑΝΤΩΝΗ ΑΝΤΩΝΟΠΟΥΛΟΥ

Επιβλέπων: Ευστάθιος Ζάχος
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 19η Ιουλίου 2013.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
....	....	....
Ευστάθιος Ζάχος	Άρης Παγουρτζής	Αλέξανδρος Παπαϊωάννου
Καθηγητής Ε.Μ.Π.	Επ. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2013

(Υπογραφή)

....

Αντωνης Αντωνοπουλος
Διπλωματούχος Σχολής Εφαρμοσμένων Μαθηματικών και Φυσικών Επιστημών Ε.Μ.Π.

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Εφαρμοσμένων Μαθηματικών και Φυσικών Επιστημών
Τομέας Μαθηματικών
Εργαστήριο Λογικής και Επιστήμης Υπολογισμών

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τους καθηγητές μου κ. Στάθη Ζάχο και κ. ;ρη Παγουρτζή για την επίβλεψη αυτής της διπλματικής εργασίας, και ιδιαίτερα τον κ. Ζάχο, που μου έδωσε την ευκαιρία να την εκπονήσω στο Εργαστήριο Λογικής και Επιστήμης Υπολογισμών.

Επίσης, θα ήθελα να ευχαριστήσω τον καθηγητή του Εργαστηρίου κ. Δημήτρη Φωτάκη, όπως και τους υπόλοιπους διδακτορικούς και διπλωματικούς φοιτητές, για την συμπαράσταση και την κατανόησή τους.

Περίληψη

Τα τελευταία χρόνια, οι τυχαιοκρατικοί αλγόριθμοι έχουν προσφέρει μια σημαντική τεχνική σχεδίασης αλγορίθμων στην Θεωρητική Πληροφορική, προσφέροντας αλγόριθμους με μεγάλη αποδοτικότητα και εύκολη σχεδίαση. Η κλάση **BPP** σχεδόν αντικατέστησε την **P** ως το μοντέλο του αποδοτικού υπολογισμού. Αυτή η πρόοδος, όμως, έθεσε φυσιολογικά το ερώτημα: Η τυχαιότητα είναι εγγενές χαρακτηριστικό αυτής της υπολογιστικής 'ευκολίασ', ή πρόκειται απλά για μια σχεδιαστική τεχνική που μπορεί να αφαιρεθεί μηχανιστικά επιβαρύνοντας μόνο πολυωνυμικά τον αλγόριθμο· Πρόσφατες καινοτομίες στην ερευνητική περιοχή δείχνουν ότι, κάτω από κάποιες εύλογες υποθέσεις, κάθε τυχαιοκρατικός αλγόριθμος μπορεί να προσομοιωθεί από έναν ντετερμινιστικό, χωρίς να υπάρχει παραπάνω από πολυωνυμική επιβάρυνση στον αριθμό βημάτων (χρόνο) του αλγορίθμου. Κάποιοι ερευνητές πιστεύουν ακόμα και ότι **BPP** = **P** Σε αυτό το κείμενο θα μελετηθούν τα αποτελέσματα αυτά που αφορούν Ομοιόμορφες Συνθήκες, δηλαδή υποθέσεις που αφορούν ομοιόμορφα υπολογιστικά μοντέλα. Μια παράλληλη προσέγγιση έρχεται από το πεδίο των Κυκλωμάτων Boole, που αποτελεί σύνηθες εναλλακτικό υπολογιστικό μοντέλο, παρ'οτι μη-ομοιόμορφο. Αποδεικνύεται ότι η ύπαρξη κάτω φραγμάτων για τέτοια μοντέλα, είναι αποτέλεσμα της ύπαρξης τέτοιων ντετερμινιστικών προσομοιώσεων.

Λέξεις Κλειδιά

Ντετερμινιστική Προσομοίωση, Τυχαιοκρατικοί Αλγόριθμοι, Τυχαιοκρατικές Κλάσεις Πολυπλοκότητας, Κυκλώματα, Κάτω φράγματα

Abstract

During the past years, randomization has offered a great comfort in Computer Science, by providing efficient algorithms for many computational problems. The class **BPP** has almost replaced **P**, as the class of efficiently solvable problems. This progress also raised a question: The simplification given to our computations by using a random "coin toss" is inherent or circumstantial? In other words, randomization provides a non-trivial computational boost-up, or it's just a design comfort, and we can finally remove it. Recent advances have proven that it is possible, under some reasonable assumptions, to replace a **BPP** randomized algorithm with a deterministic one (i.e., to *derandomize*), only with polynomial loss of efficiency. Today, there are many researchers who believe that finally $\mathbf{BPP} = \mathbf{P}$. The main reason for this perception to be widely believed, is that real *randomness* doesn't really exist in computers. It is under discussion if it even exists in Nature. Randomized Algorithms and "random sources" occasionally used by Computer Scientists (especially Cryptographers) are based on functions whose behavior is simply hard to predict. It is not clear that our computers have access to an "endless stream" of indepedent coin tosses. The main topic will be to investigate if we can simulate a randomized algorithm by a deterministic one, using constructions that provide bits almost indistinguishable from bits chosen at random (using he Uniform Distribution). The existence of such constructions, and the conditions necessary for their existence is a wide field of research during the last two decades. Also, a different view on the issue of derandomization comes from another area of research in Theoretical Computer Science, the Boolean Circuits, and specifically from the effort to find lower bounds for certain families of circuits. The existence of such bounds could separate known Complexity Classes, and it would imply even that $\mathbf{P} \neq \mathbf{NP}$! The "quest" for lower bounds, using Boolean Circuits as an alternative model of computation, seemed easier than using the (traditional) Turing Machines, and there were many and remarkable results. Unfortunately, all these efforts were (so far) unfruitful, but, as we will see, they are closely related with derandomization conjectures.

Keywords

Derandomization, Uniform Assumptions, Circuit Lower Bounds, Deterministic Simulation, Probabilistic Complexity Classes

Contents

Chapter 1

Introduction

1.1 What is Derandomization?

During the past years, randomization has offered a great comfort in Computer Science, by providing efficient algorithms for many computational problems. The class **BPP** has almost replaced **P**, as the class of efficiently solvable problems.

This progress also raised a question: The simplification given to our computations by using a random "coin toss" is inherent or circumstantial? In other words, randomization provides a non-trivial computational boost-up, or it's just a design comfort, and we can finally remove it.

Recent advances have proven that it is possible, under some reasonable assumptions, to replace a **BPP** randomized algorithm with a deterministic one (i.e., to *derandomize*), only with polynomial loss of efficiency. Today, there are many researchers who believe that finally **BPP** = **P**.

The main reason for this perception to be widely believed, is that real *randomness* doesn't really exist in computers. It is under discussion if it even exists in Nature. Randomized Algorithms and "random sources" occasionally used by Computer Scientists (especially Cryptographers) are based on functions whose behavior is simply hard to predict. It is not clear that our computers have access to an "endless stream" of indepedent coin tosses.

Randomized algorithms have some cases of possible powers, including:

- Randomization always help for hard problems (i.e. **BPP** = **EXP**)
- The power orf randomization is problem-specific.
- True randomness is never needed, and random coin tosses can be simulated deterministically (i.e. **BPP** = **P**).

The main topic will be to investigate if we can simulate a randomized algorithm by a deterministic one, using constructions that provide bits almost indistinguishable from bits chosen at random (using he Uniform Distribution). The existence of such constructions, and the conditions necessary for their existence is a wide field of research during the last two decades.

Also, a different view on the issue of derandomization comes from another area of research in Theoretical Computer Science, the Boolean Circuits, and specifically from the effort to find lower bounds for certain families of circuits. The existence of such bounds could separate known Complexity Classes, and it would imply even that $\mathbf{P} \neq \mathbf{NP}$! The "quest" for lower bounds, using Boolean Circuits as an alternative model of computation, seemed easier than using the (traditional) Turing Machines, and there were many and remarkable results. Unfortunately, all these efforts were (so far) unfruitful, but, as we will see, they are closely related with derandomization conjectures.

1.2 Short History of Derandomization

As we mentioned above, in Computer Science randomization doesn't really exist. Depending on the theory we use to define and measure it, the meaning of "randomness" takes different forms. Viewed by Shannon's Information Theory, randomness represents the *lack of information.* In the context of Kolmogorov's Complexity Theory, it represents the *lack of structure.* In the theory we'll use as our model for randomness, it is viewed as an *effect on an observer with certain computational abilities.* In this model, we view objects as equal if they cannot be told apart by any *efficient* procedure. That is, a Distribution that cannot be efficiently distinguished by the Uniform Distribution will be considered random.

Hardness-Randomness Tradeoffs The main idea in derandomizing techniques is the use of a*hard* computational problem to construct pseudorandom sequences, i.e. sequences of bits that look random to any efficient observer, which we will use to replace the random bits of a randomized algorithm. The algorithm will not have enough time to distinguish the pseudorandom sequence from the truly random one, and so it will behave in the same way. The above idea, an interpretation of computational hardness as randomness, is known as "Hardness-Randomness Tradeoffs", and was introduced during the 80's by Andew Yao [Yao82], and M. Blum-S.Micali [BM84], who in their works on Cryptography introduced the concept of *hardness-randomness tradeoffs*: If we had a hard-to-compute function, we could use it to compute a string that "looks" random to any observer, by constructing functions that perform this procedure, called *Pseudorandom Generators.*

The stronger the *hardness assumption* we make, the better the *deterministic simulation* we obtain! The exchange between computational hardness and randomness forms a hypothetical "curve", in which we can consider two sides: The "High-End", in which we demand a *full* derandomization of a probabilistic complexity class, and we ask for the corresponding hardness assumption, and the "Low-End", in which we aks for the *weakest* hardness assumption we can make, in order to obtain any version of a (non-trivial) deterministic simulation of a probabilistic complexity class.

A few years later, Noam Nisan and Avi Wigderson [NW94] weakened the hardness assumption, introducing new trade-offs, first time for the purposes of *derandomization*, i.e. the simulation of every randomized algorithm by a deterministic one. They showed that under an "average-case" assumption we can build a pseudorandom generator strong enough to simulate every probabilistic polynomial-time algorithm.

This work culminated in 1997, when Russell Impagliazzo and A. Wigderson finally proved in [IW97], that $\mathbf{P} = \mathbf{BPP}$ if $\mathbf{E}$ requires exponential-size circuits. In their proof they managed to show that an assumption about the worst-case complexity of a problem implies an assumption about its average-case complexity. Such a result is usually called a *hardness amplification* result, and it gave them the possibility to use the aforementioned resuts of Nisan and Wigderson. This consists a "High-End Tradeoff" between Hardness and Randomness.

Uniform Derandomization All the above results where based on a non-uniform setting, that is, the use of lower bounds of uniform classes in non-uniform models. The problem with non-uniformity is that different model of computation (circuit) is used for each input length, and there is no a priori connection between the different circuits used.

In 1998, Impagliazzo and Wigderson gave the first result on a uniform complexity assumption (namely $\mathbf{BPP} \neq \mathbf{EXP}$). In their proof the use the above results on the non-uniform setting, and many other results from Complexity Theory.

The work on "uniform" Derandomization was continued, and other classes, such as **ZPP**, **RP**, and **AM** (which can be viewed as the randomized version of **NP**) started to receive attention.

We will mainly focus to advances on *Uniform Derandomization* of **BPP**, **RP** and Arthur-Merlin games, by presenting all necessary notions and techniques. However, a short introduction to non-uniformm derandomization (using Pseudorandom Generators) is inevitable, not only for the sake of the completeness of the text and the historical antecedence of these results, but because uniform derandomization uses tools developed especially for the non-uniform setting.

Chapter 2

Basic Definitions and Results

A basic knowledge of (classical) complexity theory, such as the Turing Machine computation model, basic complexity classes and fundamental results is necessary. In the next sections, we briefly mention some basic notions, which the advanced reader can skip.

2.1 Basic Complexity Facts

Definition 2.1. *A function $f : \mathbb{N} \to \mathbb{N}$ is called time-constructible or space-constructible if there is a Turing Machine (from now TM) M that computes tha function f on input n in time $\mathcal{O}(n+f(n))$ or space $\mathcal{O}(f(n))$, respectively.*

By "time" we mean the number of steps of the TM M, and by "space" the extra cells used by M during the computation. The time bound of a TM must be superlinear, in order the TM to be able to read its input.

Definition 2.2. *For a time-constructible function $t(n)$, and a space-constructible function $s(n)$, let:*

- **DTIME**$(t(n))$ *be the set of languages decided by a polynomial-time TM in $t(n)$ time.*
- **NTIME**$(t(n))$ *be the set of languages decided by a polynomial-time* nondeterministic *TM in $t(n)$ time.*
- **DSPACE**$(s(n))$ *be the set of languages decided by a polynomial-time TM using $s(n)$ space.*
- **NSPACE**$(s(n))$ *be the set of languages decided by a polynomial-time* nondeterministic *TM using $s(n)$ space.*

where n is the length of the input string x, usually denoted as $|x|$.

These classes form hierarchies, that is, sequences of inclusions, which are proper under some conditions, as we'll see in the following theorems. The hierarchies confirm our intuition, that if we let a TM run for strictly more time or use strictly more space, it can compute strictly more languages. Using the above definitions, we can construct our basic complexity classes:

- $\mathbf{P} = \bigcup_{c \in \mathbb{N}} \mathbf{DTIME}(n^c)$
- $\mathbf{NP} = \bigcup_{c \in \mathbb{N}} \mathbf{NTIME}(n^c)$
- $\mathbf{E} = \mathbf{DTIME}\left(2^{\mathcal{O}(n)}\right)$
- $\mathbf{EXP} = \mathbf{DTIME}\left(2^{n^{\mathcal{O}(1)}}\right)$

We also define the following Complexity Classes:

- $\mathbf{EE} = \mathbf{DTIME}\left(2^{n^{\mathcal{O}(n)}}\right)$
- $\mathbf{QuasiP} = \mathbf{DTIME}(2^{poly \log n})$
- $\mathbf{SUBEXP} = \bigcap_{\epsilon > 0} \mathbf{DTIME}(2^{n^{\epsilon}})$

and the advice string computational model:

Definition 2.3. *Let $\mathcal{C}$ be a class of languages, and $\mathcal{F}$ a class of functions from nonnegative integers to strings. We define the class $\mathcal{C}/\mathcal{F}$ to consist of all languages $A = \{x : \langle x, h(|x|)\rangle \in L\}$ for some $L \in \mathcal{C}$ and $h \in \mathcal{F}$. We use $\mathcal{C}/h(n)$ for a certain function h. We call $h_{|x|} = h(|x|)$* advice string *for each input length.*

According to the above definition, we introduce the class $\mathbf{P_{/poly}}$ as the class of languages L for which there exists a language $B \in \mathbf{P}$ and a function $h \in poly(n)$, with $h : |h(n)| \leq p(n)$ for some fixed polynomial p, such that:

$$x \in L \Leftrightarrow \langle x, h(|x|)\rangle \in B$$

It is clear from the definition that we can use a different advice string for each input length[1].

One can note the similarity between this class and the classical characterization of $\mathbf{NP}$ (the existence of a succinct certificate for each "*yes*" instance[2]). The difference between the two characterizations is that the certificate (or witness) $h(|x|)$ of a string $x \in L$ must work for *all* strings of the same length. We cannot simply guess such a witness, and instead have to store it in the "program". In fact, it is known that $\mathbf{P_{/poly}} - \mathbf{NP} \neq \emptyset$, since $\mathbf{P_{/poly}}$ contains a non-recursive language (a version of the Halting Problem).

---

[1] We will give an alternative definition $\mathbf{P_{/poly}}$ on a next chapter, in the context of Boolean Circuits.

[2] It's a classical Complexity's Theorem that: "A language $L \in \mathbf{NP}$ if and only if there exists a relation R, which is polynomially decidable and polynomially balanced (i.e. $(x, y) \in R \Leftrightarrow |x| \leq |y|^k$, for some $k \in \mathbb{N}$), such that: $L = \{x \mid \exists y : (x, y) \in R\}$".

2.2 Oracle Turing Machines and Relativization

Oracle Turing Machines are an extension of regular machines, in which we add an extra property: the ability to function in an alternative computational "universe". There, we can explore new computational abilities, and find alternative relations between known complexity classes. The transition to this new "universe" is done by giving access to an extra language A, called *oracle*, which we can ask questions of the form: "Is x in A?", and take an instant answer. We give the formal definition:

Definition 2.4. *An Oracle Turing Machine is a Deterministic or Nondeterministic Turing Machine M, that has a special read-write tape and three extra states: $q_?$ (the query state), and q_{yes}, q_{no} (the answer states). We also specify a language $A \subseteq \{0,1\}^*$, that is used as the* oracle *for M.*

During the execution, M can enter the state $q_?$, and the machine enters q_{yes} if $z \in A$, or q_{no} if $z \notin A$, where z is the content of the special oracle tape. Regardless the choice of A, a membership query to A counts as one single *computational step. We denote such a machine M, using A as oracle on input x as: $M^A(x)$.*

Also, we can define Oracle Turing Machines using Boolean functions $f : \{0,1\}^* \to \{0,1\}$ instead of languages (we always use total functions *on $\{0,1\}^*$), because each such function f can be regarded as the characteristic function of the language $A = \{x \mid f(x) = 1\}$. We write M^f to denote the computation of M using f as an oracle.*

We can group time or space bounded Oracle Turing Machines, and create variations of our usual classes. For example, $\mathbf{P}^A$ is the class of all languages decided by a polynomial-time deterministic Turing Machine with oracle access to A, $\mathbf{NP}^A$ its non-deterministic counterpart, and in general:

Definition 2.5. *If $\mathcal{C}$ is a complexity class, we denote $\mathcal{C}^A$ the complexity class of all languages decided by the same machines as in $\mathcal{C}$, but now with oracle access to A.*

Also, if we define $\mathbf{P}^{\texttt{SAT}}$, where SAT is the language encoding the well-known $\mathbf{NP}$-complete problem, we can easily see that it is equal to any class $\mathbf{P}^L$, where $L \in \mathbf{NP}$, because every $L \in \mathbf{NP}$ is reduced to SAT in polynomial time. So, we can rewrite this class as $\mathbf{P}^{\mathbf{NP}}$.

In general, when one class uses as oracle another class, it's implied that the former's languages are decided by Turing Machines which use as oracle any complete problem of the latter.

A bizarre phenomenon concerning this kind of classes is that there is a language A for which $\mathbf{P}^A = \mathbf{NP}^A$, and another language B for which $\mathbf{P}^B \neq \mathbf{NP}^B$. So, there are contradicting "alternative universes", which (unfortunately) means that oracles don't have a conclusive answer for $\mathbf{P}$ vs $\mathbf{NP}$.

It is remarkable that Oracle Turing Machines have the fundamental (and crucial) properties of regulars: They can be also represented by strings, and one can simulate another with negligible loss of efficiency, regardless of what is the oracle A. So, any result using only these properties in regular TMs holds also for TMs with oracle access to every language A. In other words, it can be "transferred" from our regular computational universe to any alternative. These results are called *relativizing results.* **P** vs **NP** is a *nonrelativizing* result, due to the inconsistency we saw above.

We end this section by defining an "oracle type" reduction, called Turing reduction. Intuitively, a language A is *Turing reducible* to a language B, if there is a TM M for A, which can ask during its computation some membership questions about language B (i.e. it can use B as an oracle). We use directly Boolean functions in the formal definition, because we will find it more flexible in our future use of these reductions:

Definition 2.6 (Turing Reductions)**.** *A function f is polynomial-time Turing reducible to a function g, denoted by $f \leq_T^p g$, if there is a polynomial-time Oracle Turing Machine M^g which computes f.*
Of course, we can restrict *the length of the queries by writing $f_n \leq_T^p g_{h(n)}$ which means*[3] *that the queries to the oracle g have length* at most $h(n)$. *The same definition holds for languages (using their characteristic functions): A language A is polynomial-time Turing reducible to a language B, denoted by $A \leq_T^p B$,if $\chi_A \leq_T^p \chi_B$.*

We mention some essential properties of Turing Reductions:

- $A \leq_T^p A$ (Reflexive property)
- $\left(A \leq_T^p B\right) \wedge \left(B \leq_T^p C\right) \Rightarrow A \leq_T^p C$ (Transitive propery)
- $A \leq_m^p B \Rightarrow A \leq_T^p B$, where "$\leq_m^p$" denotes the (regular) Karp reduction, which implies that Turing reductions are stronger than Karp reductions.
- $A \leq_T^p B \Rightarrow A \leq_T^p \overline{B}$
- **P** and **PSPACE** are closed under Turing reductions.
- **NP** $=$ *co***NP** if and only if **NP** is closed under Turing reductions.

[3]For a Boolean function f, f_n denotes the restriction of f to inputs of length n.

2.3 The Polynomial-Time Hierarchy

Now that we have seen the power given by oracles, it's reasonable to question *how much* power we take by adding a certain oracle, and which is the relation between different oracle classes.

We will define an hierarchy of such classes, which capture a large variety of problems, in order to study these questions, and also relate with them the complexity classes we 've already defined.

Definition 2.7 (Polynomial Hierarchy). *We recursively define:*
$\Delta_0^p = \Sigma_0^p = \Pi_0^p = \boldsymbol{P}$, *and for any* $i \in \mathbb{N}$*:*

- $\Delta_{i+1}^p = \mathbf{P}^{\Sigma_i^p}$
- $\Sigma_{i+1}^p = \mathbf{NP}^{\Sigma_i^p}$
- $\Pi_{i+1}^p = co\mathbf{NP}^{\Sigma_i^p}$

Also we have:

$$\mathbf{PH} \equiv \bigcup_{i \geqslant 0} \Sigma_i^p$$

Intuitively, each time we "jump" on the next Δ_i^p by using the previous Σ_{i-1}^p as an oracle to a polynomial-time deterministic machine. Then, Σ_i^p is its non-deterministic analogue, and Π_i^p the complementary class of Σ_i^p.

The inclusions are shown in the following Hasse diagram, where $A \to B$ means $A \subseteq B$:

$$\begin{tikzcd}
 & \Sigma_1^p \arrow[dr] & & \Sigma_2^p \arrow[dr] & & \cdots \\
\mathbf{P} \arrow[ur] \arrow[dr] & & \Delta_2^p \arrow[ur] \arrow[dr] & & \Delta_3^p \arrow[ur] \arrow[dr] & \\
 & \Pi_1^p \arrow[ur] & & \Pi_2^p \arrow[ur] & & \cdots
\end{tikzcd}$$

All our regular classes exist in the first two levels:

- $\mathbf{P} = \Delta_1^p$, $\mathbf{NP} = \Sigma_1^p$, $co\mathbf{NP} = \Pi_1^p$
- $\mathbf{P}^{\mathbf{NP}} = \Delta_2^p$, $\mathbf{NP}^{\mathbf{NP}} = \Sigma_2^p$, $co\mathbf{NP}^{\mathbf{NP}} = \Pi_2^p$

We remind to the reader that the famous Traveling Salesman Problem (`TSP`) is in $\mathbf{FP}^{\mathbf{NP}}$ (in fact it is $\mathbf{FP}^{\mathbf{NP}}$-complete), where $\mathbf{FP}$ is the class of *functions* (instead of languages) decided by polynomial-time deterministic Turing Machines, i.e. the "function variation" class of $\mathbf{P}$, fact that emphasizes he robustness of those classes.

The strange symbols Σ, Π, Δ, are used for traditional reasons, since Polynomial Hierarchy is the "efficient" analogue of Arithmetical Hierarchy, defined by Stephen Kleene, whose main difference is that it deals with the *decidability*, and not the efficient recognition of languages.

We mention some basic properties of these classes. For every $i > 0$:

- $\Sigma_i^p \cup \Pi_i^p \subseteq \Delta_{i+1}^p \subseteq \Sigma_{i+1}^p \cap \Pi_{i+1}^p \subseteq \mathbf{PSPACE}$
- Closure Properties:
  - If $A, B \in \Sigma_i^p$, then $A \cup B \in \Sigma_i^p$, $A \cap B \in \Sigma_i^p$ and $\overline{A} \in \Pi_i^p$.
  - If $A, B \in \Pi_i^p$, then $A \cup B \in \Pi_i^p$, $A \cap B \in \Pi_i^p$ and $\overline{A} \in \Sigma_i^p$.
  - $A, B \in \Delta_i^p$, then $A \cup B \in \Delta_i^p$, $A \cap B \in \Delta_i^p$ and $\overline{A} \in \Delta_i^p$.
- Also, $\mathbf{NP}^{\Sigma_i^p \cap \Pi_i^p} = \Sigma_i^p$.

Despite its elegance, the oracle description of the Polynomial Hierarchy is not always useful and clear, due to the tricky oracle description. We'll give an aternative description of each language $L \in \mathbf{PH}$, used also in the Arthmetical Hierarchy for the first time. Firstly, we will "connect" each $L \in \Sigma_i^p$ class to the previous Π_{i-1}^p class[4]:

Theorem 2.1. *Let L be a language ,and $i \geq 1$.*
$L \in \Sigma_i^p$ iff there is a polynomially balanced relation R (that is, $(x, y) \in R \Leftrightarrow |x| \leq |y|^k$ for some k), such that the language $\{x; y \mid (x, y) \in R\}$ is in Π_{i-1}^p, and:

$$L = \{x \mid \exists y : (x, y) \in R\}$$

In other words, we can jump from Π_{i-1}^p to Σ_i^p class by adding an(other) existential quantifier in front of our predicate R. Of course, in the same way we can "jump" from a Σ_{i-1}^p to a Π_i^p class, by using the complementary universal quantifier:

Theorem 2.2. *Let L be a language , and $i \geq 1$. $L \in \Pi_i^p$ iff there is a polynomially balanced relation R, such that the language $\{x; y \mid (x, y) \in R\}$ is in Σ_{i-1}^p, and:*

$$L = \{x \mid \forall y, |y| \leq |x|^k : (x, y) \in R\}$$

[4]Recall the footnote of page 6 for a simplification of this Theorem on the 1^{st} level of the Polynomial Hierarchy.

By applying recursively the above Theorems, we can have a full description of each language in the Polynomial Hierarchy, using alternating quantifiers:

$$L \in \Sigma_i^p \Leftrightarrow L = \{x \mid \exists y_1 \forall y_2 \exists y_3 \cdots Qy_i : (x, y_1, ..., y_i) \in R\}$$

where the i^{th} quantifier Q is $\forall$, if i is even, and $\exists$, if i is odd. And also:

$$L \in \Pi_i^p \Leftrightarrow L = \{x \mid \forall y_1 \exists y_2 \forall y_3 \cdots Qy_i : (x, y_1, ..., y_i) \in R\}$$

where the i^{th} quantifier Q is $\forall$, if i is odd, and $\exists$, if i is even. In both cases R is a is a polynomially balanced and polynomially-time decicable $(i+1)$-ary relation. We can intuitively say that a language is in Σ_i^p if it can be described by i alternating quantifiers (apllied on a proper prediacate) starting with $\exists$, and in Π_i^p, if the the first is $\forall$.

There are certain results concerning the inclusions in this Hierarchy. So far, we believe that the inclusions are proper, and the Hierarchy has infinite levels (unless $\mathbf{P} = \mathbf{NP}$). The following Theorems show under which conditions (which are not likely to be valid) that is not happening:

Theorem 2.3. *If for some $i \geq 1$, $\Sigma_i^p = \Pi_i^p$, then for all $j > i$:*

$$\Sigma_j^p = \Pi_j^p = \Delta_j^p = \Sigma_i^p$$

Or, the polynomial hierarchy collapses *to the i^{th} level.*

Especially:

- If $\mathbf{P} = \mathbf{NP}$, or even $\mathbf{NP} = co\mathbf{NP}$, the Polynomial Hierarchy collapses to the first level.
- If there is a **PH**-complete problem, then the polynomial hierarchy collapses to some finite level.

It is known that every language in **PH** can be simulated by a polynomial-*space* deterministic Turing Machine, i.e. $\mathbf{PH} \subseteq \mathbf{PSPACE}$. But, it is open question whether $\mathbf{PH} = \mathbf{PSPACE}$. If it finally is, then **PH** has complete problems (**PSPACE** has enough), and so it collapses to some finite level.

From the above it is obvious that there are many complexity classes which can be fully described by the *number* and the *type* of quantifiers applied on a polynomial-time complutable and polynomially balanced predicate.

We present an alternative characterization of complexity classes using only the quantifiers needed for the quantification implied by the definition of each class. This notation provides us with a uniform description of complexity classes defined in various contexts (as we'll see in the next section), and we'll be able to obtain immediate relations and inclusions among them.

Definition 2.8. *We denote as $\mathcal{C} = (Q_1/Q_2)$, where $Q_1, Q_2 \in \{\exists, \forall\}$, the class $\mathcal{C}$ of languages L satisfying:*

- $x \in L \Rightarrow Q_1 y \; R(x, y)$
- $x \notin L \Rightarrow Q_2 y \; \neg R(x, y)$

We can easily notice that: $co\mathcal{C} = co(Q_1/Q_2) = (Q_2/Q_1)$.

So, using the classical existential and universal quantifiers we can define the basic complexity classes, by implying their definitional properties. For example, in class **P** there is a computational path which either accepts, either rejects. So, it is easy to see that $\mathbf{P} = (\forall/\forall)$. On the other hand, for languages in class **NP** there is a computational tree for each input, and we accept it if *there is* an accepting branch, or we reject it if *all* the branches reject. Hence, we have that: $\mathbf{NP} = (\exists/\forall)$. The complementary class *co***NP** can be also defined as $co\mathbf{NP} = (\forall/\exists)$.

Instead of using a single quantifier, we can use quantifier "vectors", and so we can describe every class in the Poylnomial hierarchy, according to the alternating quantifier characterization we mentioned above:

- $\Sigma_2^p = (\exists\forall/\forall\exists)$, $\Pi_2^p = (\forall\exists/\exists\forall)$, and in general:
- $\Sigma_k^p = (\exists\forall \cdots Q_m)/\forall\exists \cdots Q_n)$, where:
  - Q_m represents $\exists$, if k is odd, or $\forall$, if k is even, and
  - Q_n represents $\forall$, if k is odd, or $\exists$, if k is even.
- $\Pi_k^p = (\forall\exists \cdots Q_m/\exists\forall \cdots Q_n)$, where:
  - Q_m represents $\forall$, if k is odd, or $\exists$, if k is even.
  - Q_n represents $\exists$, if k is odd, or $\forall$, if k is even.

2.4 Randomized Complexity Classes

In this section we will develop a theory for Turing Machines using probabilistic choices. This is a computational model used very widely, as we saw in the introductory chapter, and the question of the intrinsic relation between these complexity classes and their deterministic analogues is the main purpose of this thesis.

Basic definitions and results of Randomized Computation and Complexity Theory will be mentioned, by introducing the notion of a probabilistic TM, the different types of algorithms used, and the complexity classes containing them.

Definition 2.9 (Probabilistic Turing Machines)**.** *A* Probabilistic Turing Machine *(PTM) is a Turing Machine with* two *transition functions* δ_0 *and* δ_1*. To execute a PTM M on an input x, we choose in each step with probability* 1/2 *to apply the transition function* δ_0 *or* δ_1*. This choice is indepedent of all previous choises. The machine outputs "yes" (accepts) or "no" (rejects). We denote by $M(x)$ the* random variable *corresponding to the value M writes at the end of its proccess. For a function $T : \mathbb{N} \to \mathbb{N}$, we say that M runs in $T(n)$-time if for an input x, M halts in $T(|x|)$ steps regardless of the random choices it makes.*

The ressemblance between a PTM and a nondeterministic TM is remarkable (an NDTM has also two transition functions), and confirms the universality of non-determinism (which is although a non-realistic model). The main difference between them is the perception we have about the computations graph: A NDTM is said to *accept*, if $\exists$ a branch that outputs "yes", whereas in the case of PTMs, we consider the *fraction* of accepting branches.

Definition 2.10 (**BPP** Class)**.** *For $T : \mathbb{N} \to \mathbb{N}$, and $L \subseteq \{0,1\}^n$, we say that a PTM M decides L in time $T(n)$, if $\forall x \in \{0,1\}^n$, M halts in $T(|x|)$ steps, and:*

- *If* $x \in L \Rightarrow Pr[M(x) = \text{"yes"}] \geq 2/3$
- *If* $x \notin L \Rightarrow Pr[M(x) = \text{"no"}] \geq 2/3$

We denote by $\mathbf{BPTIME}(T(n))$ *the class of languages decided by PTMs in* $\mathcal{O}(T(n))$ *time. We also define:*

$$\mathbf{BPP} = \bigcup_c \mathbf{BPTIME}(n^c)$$

The class **BPP** captures our notion of "effective" probabilistic computation, exactly as **P** in deterministic computations. Our main topic wil be to explore the *relation* between the two computational models and their complexity classes.

Also, the class **BPP** captures the (probabilistic) algorithms with (what we call) "*two-sided*" error, which means that a **BPP** algorithm is allowed to make error for both outputs, i.e. answer "no" when $x \in L$ or "yes" when $x \notin L$.

However, many algorithms have appeared the last decades which have only "*one-sided*" error, that is, they *never* answer "yes" if $x \notin L$, although they may answer "no" when $x \in L$ (and vice versa). A classical example is the "Miller-Rabin" and "Solovay-Strassen" primality tests.
So, we need to introduce the proper complexity classes for these problems:

Definition 2.11 (**RP** Class)**.** ***RTIME***$(T(n))$ *contains all languages L for which $\exists$ PTM M, running in $T(n)$ time such that:*

- *If* $x \in L \Rightarrow Pr[M(x) = "yes"] \geq 2/3$
- *If* $x \notin L \Rightarrow Pr[M(x) = "no"] = 1$

We also define:

$$\boldsymbol{RP} = \bigcup_c \boldsymbol{RTIME}(n^c)$$

Basic Properties:

- **RP** $\subseteq$ **NP** (Since every accepting path is a "certificate" for the input.)
- The class **coRP** $= \{L|\overline{L} \in \mathbf{RP}\}$ captures "one-sided" algorithms with the error in the other direction.
- **RP** $\subseteq$ **BPP** and **coRP** $\subseteq$ **BPP**.
- The choice "2/3" on the above definitions is arbitrary. By indepedent repetitions, we can increase it however we want! In fact, we can reduce it to $1 - 2^{-p(|x|)}$ (for $p(|x|) > 1$).
- **P** $\subseteq$ **BPP** $\subseteq$ $\mathbf{P_{/poly}}$ and **BPP** $\subseteq$ **EXP**.
- Also **BPP** $\subseteq \Sigma_2^p \cap \Pi_2^p$ (Sipser-Gács Theorem)
- If **P** = **NP**, then **BPP** = **P**.

We know that it's not very likely that **P** = **NP**, but the possibility that **BPP** $\neq$ **P** remains still open. However, many researchers suspect that finally **BPP** = **P**. On the next chapters, we will show that if certain plausible complexity-theoretic conjectures are true, then **BPP** = **P**, and our two models of efficient computations coincide.

Also, **BPP** has *not* complete problems (as far as we know). That is an expected property, since **BPP** is a *semantic* class, because **BPP** TMs accept or reject with a certain probability, which is a non-trivial property. We can't even "test" if a given TM has this property, due to classical undecidability results (Rice's Theorem).

If, finally, we prove that **P** = **BPP**, then **BPP** will have a complete problem (Since **P** has).

2.5 Interactive Proof Systems

The standard "certificate" **NP** scenario, where we accept a statement (for example a proof) if someone provides a succinct certificate (which exists only for true statements), can be generalized by introducing *interaction* in the basic scheme. That is, the person who verifies the proof asks the person who provides the proof a series of "queries", before he is convinced, and if he is, he provide the certificate. From now on, the first person will be called **verifier**, and the second **prover**.

If the verifier is a simple deterministic TM, then the interactive proof system is described precisely by the class **NP**. So, if we want to obtain more computational power using the interaction, we have to let the *verifier* be probabilistic, which means that the verifier's queries will be computed using a probabilistic TM.

We now give the precise definition of probabilistic proof systems, and the class contains them:

Definition 2.12. *For an integer $k \geq 1$ (that may depend on the input length), a language L is in $\mathbf{IP}[k]$ if there is a probabilistic polynomial-time T.M. V that can have a k-round interaction with a function $P : \{0,1\}^* \rightarrow \{0,1\}^*$ such that:*

- $x \in L \Rightarrow \exists P : Pr[\langle V, P\rangle(x) = 1] \geq \frac{2}{3}$ *(Completeness)*
- $x \notin L \Rightarrow \forall P : Pr[\langle V, P\rangle(x) = 1] \leq \frac{1}{3}$ *(Soundness)*

We also define:

$$\mathbf{IP} = \bigcup_{c \geq 1} \mathbf{IP}[n^c]$$

In class **IP**, the verifier's random string is *private*, since the prover does not depend on the verifier's random strings. Often these are called *private coin* interactive proofs. In a variation called Arthur-Merlin proofs (or *public coin* proofs), the verifier's questions are obtained by tossing coins and *revealing them to the prover.*

The story goes like this (from [Bab85][5]):

> "*King Arthur recognizes the supernatural intellectual abilities of Merlin, but doesn't trust him. How should Merlin convince the intelligent but impatient King that a string x belongs to a given language L? If $L \in \mathbf{NP}$, merlin will be able to present a* witness *which Arthur can check in polynomial time.*"

So, Merlin plays the role of the *prover*, and Arthur the role of the *verifier*, but in this case, Merlin has even more power than an ordinary prover, since he is able to read the whole history of the computation of Arthur on the given input, including the random coin tosses made by Arthur.

[5] László Babai used the legend of the medieval England to emphasize the analogy between a prover's infinite powers and Merlin's "magic". Merlin *cannot* predict Arthur's future random choices, and Arthur has no way of hiding from Merlin the results of his previous random choices. This will become clearer in the formal definition. The interested reader is also referred to [BM88]

Definition 2.13. *For every k, the complexity class $\mathbf{AM}[k]$ is defined as a subset to $\mathbf{IP}[k]$ obtained when we restrict the verifier's messages to be random bits, and not allowing it to use any other random bits that are not contained in these messages.*
We denote $\mathbf{AM} \equiv \mathbf{AM}[2]$.

So, $\mathbf{AM}$ is the class of languages L with an interactive proof, in which the verifier sends a random string, and the prover responding with a message, where the verifier's decision is obtained by applying a deterministic polynomial-time algorithm to the message.

Also, the class $\mathbf{MA}$ consists of all languages L, where there's an interactive proof for L in which the prover first sending a message, and then the verifier is "tossing coins" and computing its decision by doing a deterministic polynomial-time computation involving the input, the message and the random output (i.e. the "coins").

Basic properties of Interactive Proof Systems:

- The "output" $\langle V, P\rangle(x)$ is a random variable.
- It is remarkable that every language in the Polynomial Hierarchy has an interactive proof. In fact, it is known that $\mathbf{IP} = \mathbf{PSPACE}$ (proved by Adi Shamir in 1990).
- We can replace in Definition 2.12 the completeness parameter 2/3 with $1 - 2^{-n^s}$ and the soundness parameter 1/3 by 2^{-n^s}, without changing the class for any fixed constant $s > 0$.
  We can also replace the (completeness) constant 2/3 with 1, without changing the class, but replacing the soundness constant 1/3 with 0, is equivalent with a *deterministic verifier*, so class $\mathbf{IP}$ is reduced to $\mathbf{NP}$.
- Obviously, $\mathbf{MA} \subseteq \mathbf{AM}$.
- It should be clear that $\mathbf{MA}[1] = \mathbf{NP}$, $\mathbf{AM}[1] = \mathbf{BPP}$, and that $\mathbf{AM}$ could be intuitively approached as the probabilistic version of $\mathbf{NP}$ (usually denoted as $\mathbf{AM} = \mathbf{BP}\cdot \mathbf{NP}$).
- We can relate Arthur-Merlin classes with the Polynomial Hierarchy (as we did with $\mathbf{BPP}$). In fact: $\mathbf{AM} \subseteq \Pi_2^p$ and $\mathbf{MA} \subseteq \Sigma_2^p \cap \Pi_2^p$.

If we consider the complexity classes $\mathbf{AM}[k]$ (the languages that have Arthur-Merlin proof systems of a bounded number of rounds, they form an *hierarchy*:

$$\mathbf{AM}[0] \subseteq \mathbf{AM}[1] \subseteq \cdots \subseteq \mathbf{AM}[k] \subseteq \mathbf{AM}[k+1] \subseteq \cdots$$

Unlike the Polynomial Hierarchy, in which we believe the inclusions are *proper*, Arthur-Merlin Hierarchy collapses to the second level:

Theorem 2.4. *For constants $k \geq 2$, $\mathbf{AM}[k] = \mathbf{AM}[2]$.*

It is, by definition, $\mathbf{AM}[k] \subseteq \mathbf{IP}[k]$ for all k. But, S. Goldwasser and M. Sipser proved in 1987 the following counterintuitive result:

Theorem 2.5. *For every k with $k(n)$-computable in poly(n):*

$$\mathbf{IP}[k] \subseteq \mathbf{AM}[k+2]$$

Also, R. Boppana, J. Håstad and S. Zachos proved a significant collapsion theorem:

Theorem 2.6. *If $co\mathbf{NP} \subseteq \mathbf{AM}$, then the Polynomial Hierarchy collapses to $\Sigma_2^p = \Pi_2^p = \mathbf{AM}$.*

The following Hasse diagram caputures the inclusions between the most important complexity classes we've seen so far:

$$\begin{tikzcd}
 & \mathbf{NP} \arrow[r] & \mathbf{MA} \arrow[rr] \arrow[dr] & & \Sigma_2^p \\
 & & & \mathbf{AM} \arrow[dddr] & \\
\mathbf{P} \arrow[uur] \arrow[r] \arrow[ddr] & \mathbf{BPP} \arrow[uur] \arrow[ddr] & & & \\
 & & & co\mathbf{AM} \arrow[uuur] & \\
 & co\mathbf{NP} \arrow[r] & co\mathbf{MA} \arrow[ur] \arrow[rr] & & \Pi_2^p
\end{tikzcd}$$

We will now give a (private coin) Interactive Proof system for the most famous problem in $\mathbf{IP}$ that is not known to be in $\mathbf{NP}$: Graph non-isomorphism. We say that two graphs G_1 and G_2 are *isomorphic*, if there is a permutation π of the labels of the nodes of G_1, such that $\pi(G_1) = G_2$. If G_1 and G_2 are isomorphic, we write $G_1 \cong G_2$. So, we formulate the following problems:

- `GI`: Given two graphs G_1, G_2, decide if they are isomorphic.
- `GNI`: Given two graphs G_1, G_2, decide if they are *not* isomorphic.

It is obvious that `GI` $\in \mathbf{NP}$, since a succinct certificate for the isomporphism is the permutation π. So, **GNI** is in $co\mathbf{NP}$, as the complement of `GI`. We will give an interactive proof for **GNI**:

Verifier: Picks $i \in \{1, 2\}$ uniformly at random. Then, it permutes randomly the vertices of G_i to get a new graph H. Is sends H to the Prover.
Prover: Identifies which of G_1, G_2 was used to produce H. Let G_j be the graph. Sends j to V.
Verifier: Accept if $i = j$. Reject otherwise.

Now, we can confirm that it is indeed an Interactive Proof protocol:

- If $G_1 \ncong G_2$, then the powerfull prover can (nondeterministivally) guess which one of the two graphs is isomprphic to H, and so the Verifier accepts with probability 1.

- If $G_1 \cong G_2$, the prover can't distinguish the two graphs, since a random permutation of G_1 looks exactly like a random permutation of G_2. So, the best he can do is guess randomly one, and the Verifier accepts with probability (at most) 1/2, which can be reduced by additional repetitions.

This proof system relies on the Verifier's access to a *private* random source which cannot be seen by the Prover, so we confirm the crucial role the private coins play.

This protocol couldn't be an Arthur-Merlin (public coin) proof system, but we can produce an alternative protocol by restating our problem[6], which places `GNI` in the **AM** class:

Theorem 2.7. *`GNI` $\in$ **AM**.*

We discussed before why `GI` $\in$ **NP**. It is open whether `GI` is **NP**-complete, and along with `FACTORING`, is the most famous problem that is not known to be either in **P** or **NP**-complete.

If it finally is **NP**-complete, we have that `GNI` is co**NP**-complete, and Theorem 2.6 implies that the Polynomial Hierarchy collapses to the 2^{nd} level.

2.6 Counting Complexity Essentials

We will give a brief introduction to Counting Problems and Classes. In this kind of computation, we are interested not only in the existence of a solution, but in the *number* of different solutions of a certain problem. As an example, we can define the following variation of SAT:

Definition 2.14 (#SAT)**.** *Given a Boolean expression, compute the number of different truth assignments that satisfy it.*

[6]For detailed analysis of the protocol, see [AB09], pages 151-155.

A more important problem is counting the number of different perfect matchings on a bipartite graph. We know that this number is given by a characteristic of the adjacency matrix, called the *permanent*, and defined as follows:

$$perm(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} A_{i,\sigma(i)}$$

where A is a "0,1"-matrix, and S_n is the set of all permutations of n elements. Thus, we have the following formal definition:

Definition 2.15 (PERMANENT)**.** *Given the adjacency matrix A of a bipartite graph, compute the number of different perfect matchings for this graph, that is the quantity:*

$$perm(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} A_{i,\sigma(i)}$$

We can now define a class containing such "counting" problems. This class contains functions, and not languages (decision problems) as **P** and **NP**, because we are not interested only in a "yes"/"no" output, but in a certain answer (e.g. the number of satisfying truth assignments for a Boolean expression).

Definition 2.16 (#**P**)**.** *A function $f : \{0,1\} \to \mathbb{N}$ is in #**P**, if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$, and a polynomial-time Turing Machine M, such that for every $x \in \{0,1\}^*$:*

$$f(x) = \left| \left\{ y \in \{0,1\}^{p(|x|)} : M(x,y) = 1 \right\} \right|$$

Intuitively, #**P** contains problems of finding *the number of* y's satisfying a polynomial-time decidable relation $R(x,y)$, given the input x.

An important question concering #**P** is if all problems in it are efficiently solved, that is is if #**P** $\subseteq$ **FP**, or if #**P** $=$ **FP**. We do know that if #**P** $=$ **FP**, then **P** $=$ **NP**, since computing the number of certificates is harder than finding out whether a certificate exists, and if **P** $=$ **PSPACE** (which is not likely), then #**P** $=$ **FP**, since counting the number of certificates can be done in polynomial space.

Normally, the next step after introducing a complexity class, is to define reductions among its problems. The detailed definition of such reductions will not mentioned here[7], but we can intuitively say that a function f is #**P**-complete, if it is in #**P**, and a polynomial-time algorithm for f implies that # **P** $=$ **FP**. We have the following results:

Theorem 2.8. #`SAT` *is* #**P**-*complete.*

[7] The reader is referred to Papadimitriou's [Pap94] and Arora-Barak's [AB09] textbooks for formal definitions.

Theorem 2.9 (Valiant's Theorem)**.** `PERMANENT` *is* $\#\mathbf{P}$*-complete.*

Also, we can define counting classes respicting the *parity* of the number of accepting paths:

Definition 2.17. *A language L is in class $\oplus\mathbf{P}$ iff there is a polynomial-time nondeterministic Turing Machine M such that $x \in L$ iff the number of accepting paths of M on input x is* odd*.*

We conclude this section by stating a quite counter-intuitive result. Both $\#\mathbf{P}$ and $\mathbf{PH}$ are natural generalization of $\mathbf{NP}$, but the they have definitional differences (alternation of quantifiers and counting solutions), and different structure (the former is a class of functions while the latter is a class of languages), so it seemed quite implausible to correlate them somehow.

However, Seinosuke Toda proved in 1989 [Tod91] that counting is stronger than quantifiers:

Theorem 2.10 (Toda's Theorem)**.**

$$\mathbf{PH} \subseteq \mathbf{P}^{\#\mathtt{SAT}}$$

The theorem states that we can efficiently solve any problem in the Polynomial Hierarchy, given an oracle[8] to a $\#\mathbf{P}$-complete problem.

2.7 Pseudorandom Constructions

2.7.1 Pseudorandom Generators

The notion of pseudorandomness started from Cryptography, where we need to extend a random key to a much larger string, which must seem "random enough". Cryptographers' solution consists on focusing on the distribution of strings, that such a distribution has to *look like* the Uniform Distribution to *every* polynomial-time algorithm. Such a distribution is called *pseudo-random.*

We give the following definition:

Definition 2.18 (Pseudorandom Generators (PRGs))**.** *Let $G : \{0,1\}^* \to \{0,1\}^*$ be a polynomial-time computable function. Also, let $S : \mathbb{N} \to \mathbb{N}$ be a polynomial-time computable function such that $\forall n : S(n) > n$. We say that G is a* pseudorandom generator *of stretch $S(n)$, if $|G(x)| = S(|x|)$ for every $x \in \{0,1\}^*$, and for every probabilistic polynomial-time algorithm A, there exists a negligible[9] function $\epsilon : \mathbb{N} \to [0,1]$ such that:*

$$\left|Pr\left[A(G(U_n)) = 1\right] - Pr\left[A(U_{S(n)}) = 1\right]\right| < \epsilon(n)$$

[8] We remind that a Turing Machine can use a language as "oracle", that is, it has access to language's characteristic function, and each membership (to the language) question takes one computational step.

[9] A function $\epsilon : \mathbb{N} \to [0,1]$ is called negligible if $\epsilon(n) = n^{-\omega(1)}$.

for every $n \in \mathbb{N}$.

The above definition implies that it is *infeasible* for polynomial-time adversaries to distinguish between a completely random string of length $S(n)$, and a string that was generated by applying the generator G to a much shorter string of length n.

Also, this definition will not be used till the end. Insted, on a next chapter, we'll give an alternative and "weaker" definition based on a different (non-uniform) model of computation. So, in order to be clear, the Generator in Definition 2.18 will be called from now on a "*Secure* Pseudorandom Generator".

As we can see from the definition, a Pseudorandom Generator is defined by its three fundamental properties:

1. **Stretch Function**: Any pseudorandom generator stretches "short" strings of length n, called *seeds*, into longer outputs of length $S(n)$. The function $S : \mathbb{N} \to \mathbb{N}$ is called the *stretch function* of the generator.

2. **Computational Indistinguishability**: A basic property of a generator is that it "persuades" certain Turing Machines that it's output is uniformly random. In other words, any algorithm A (with certain computational abilities, probabilistic polynomial-time in the above definition), who might be thought as an "observer", cannot decide whether a string is an output of the generator, or a truly random string.

3. **Resources used**: Since a generator is a function, it has its own computational complexity, i.e. the computational resources it is allowed to use. In the above definition, we chose for the generator to work in polynomial time. As we mentioned above, in next chapters we will adapt this property.

The most remarkable result concerning Definition 2.18 is that we can connect the existence of pseudorandom generators, to the (conjectured) existence of one-way functions. In fact, we can use any one-way function to construct a generator. The following theorem states that, proved by Johan Håstad, Russell Impagliazzo, Leonid Levin and Michael Luby in 1999:

Theorem 2.11. *If one-way functions exist, then for every* $c \in \mathbb{N}$*, there exists a pseudorandom generator with stretch* $S(n) = n^c$.

Despite its theoretical value, the above Theorem can't be used until we find certain one-way functions. We will use, instead, conjectures on the *hardness* of certain functions (that is, the lower bound of resourses needed for their computation) and on the complexity classes containing them.

We end this section, by defining a variation (in fact a “relaxed” version) of a pseudorandom generator, called a *Hitting Set Generator*. This is a function G, such that for any adversary A which accepts a randomly chosen z with probability at least ε, it is required to ”provide” just one example z that A accepts. Formally:

Definition 2.19 (Hitting Set Generators (HSGs)). *A function $G : \{0,1\}^k \to \{0,1\}^m$, for $m > k$ is a Hitting Set Generator for a class $\mathcal{A}$, if for every function $A : \{0,1\}^m \to \{0,1\}$ in $\mathcal{A}$ such that $Pr_{z\in\{0,1\}^m}[A(z)) = 1] > \varepsilon$, there exists a $y \in \{0,1\}^k$ such that $A(G(y)) = 1$.*

Chapter 3

Boolean Circuits

3.1 An Introduction to Boolean Circuits

A Boolean Circuit is a natural model of *nonuniform* computation, a generalization of hardware computational methods. Its main difference from the (uniform) Turing Machine model is that while the same T.M. is used on all input sizes, a nonuniform model allows a different circuit (or a different *algorithm*) to be used for each input size. We give the formal definition of a circuit:

Definition 3.1 (Boolean circuits). *For every $n \in \mathbb{N}$ an n-input, single output Boolean Circuit C is a directed acyclic graph with n sources and* one *sink.*

- *All nonsource vertices are called* gates *and are labeled with one of $\wedge$ (and), $\vee$ (or) or $\neg$ (not).*
- *The vertices labeled with $\wedge$ and $\vee$ have* fan-in *(i.e. number or incoming edges) 2.*
- *The vertices labeled with $\neg$ have* fan-in *1.*
- *The* size *of C, denoted by $|C|$, is the number of vertices in it.*
- *For every vertex v of C, we assign a value as follows: for some input $x \in \{0,1\}^n$, if v is the i-th input vertex then $val(v) = x_i$, and otherwise $val(v)$ is defined recursively by applying v's logical operation on the values of the vertices connected to v.*
- *The* output *$C(x)$ is the value of the output vertex.*
- *The* depth *of C is the length of the longest directed path from an input node to the output node.*

The fixed size of the input limits our model to only one input size. In order to overcome this, we need to allow families (or sequences) of circuits to be used:

Definition 3.2. *Let* $T : \mathbb{N} \to \mathbb{N}$ *be a function. A* $T(n)$-size circuit family *is a sequence* $\{C_n\}_{n \in \mathbb{N}}$ *of Boolean circuits, where* C_n *has* n *inputs and a single output, and its size* $|C_n| \leq T(n)$ *for every* n.

Note that these infinite families of circuits are defined arbitrarily. There is *no* pre-defined connection between the circuits, and also we haven't any "guarantee" that we can construct them efficiently.

Like each new computational model, we can define a complexity class on it by imposing some restriction on a *complexity measure*. In the case of circuits, we can define such a measure by bounding the size of each circuit of a family that accepts a language L:

Definition 3.3. *We say that a language* L *is in* $\mathbf{SIZE}(T(n))$ *if there is a* $T(n)$-size circuit family $\{C_n\}_{n \in \mathbb{N}}$, *such that* $\forall x \in \{0,1\}^n$:

$$x \in L \Leftrightarrow C_n(x) = 1$$

And, by taking all circuit families of polynomial size, we define the class of "efficient" circuit computation:

Definition 3.4. $\mathbf{P}_{/\mathbf{poly}}$ *is the class of languages that are decidable by polynomial size circuits families. That is,*

$$\boldsymbol{P}_{/\boldsymbol{poly}} = \bigcup_c \boldsymbol{SIZE}(n^c)$$

A main concern in this point, is to connect somehow this new computational model with our existing one. Using the fact that every language in $\mathbf{P}$ can be transformed to a (polynomial-zize) circuit, as stated in the following theorem:

Theorem 3.1. *Let* `CVP` *(*`CIRCUIT VALUE` *Problem) denote the language consisting of all pairs* $\langle C, x\rangle$, *where* C *is an* n*-input and single-output circuit, and* $x \in \{0,1\}^n$ *is such that* $C(x) = 1$. `CVP` *is* $\boldsymbol{P}$*-complete.* [1].

we can reduce every language in $\mathbf{P}$ in $\mathbf{P}_{/\mathbf{poly}}$:

Theorem 3.2. $\mathbf{P} \subsetneq \mathbf{P}_{/\mathbf{poly}}$

[1]We remind that a language is $\mathbf{P}$-complete if it is in $\mathbf{P}$, and every language in $\mathbf{P}$ is logspace-reducible to it.

The inclusion is *proper*, because we know that there are *undecidable* languages with polynomial-sized circuits (which aren't obviously in **P**). The "circuit version" of `HALTING PROBLEM` is one of them.

Karp and Lipton posed the question of whether `SAT` is or not in $\mathbf{P}_{/\mathbf{poly}}$. They proved in [KL80] that if that happens, i.e. if `SAT` has polynomial-size circuits, then the Polynomial Hierarchy collapses to its second level:

Theorem 3.3 (Karp-Lipton). *If* $\mathbf{NP} \subseteq \mathbf{P}_{/\mathbf{poly}}$*, then* $\mathbf{PH} = \Sigma_2^p$.

Similarly, the following theorem shoes that $\mathbf{P}_{/\mathbf{poly}}$ seems not to contain $\mathbf{EXP}$[2]:

Theorem 3.4 (Meyer's Theorem). *If* $\mathbf{EXP} \subseteq \mathbf{P}_{/\mathbf{poly}}$*, then* $\mathbf{EXP} = \Sigma_2^p$.

Just like the Turing Machine model, there exists an Hierarchy Theorem for Boolean Circuits, proving that larger circuits can compute strictly more functions than smaller circuits (fact that assures the *robustness* of this computational model):

Theorem 3.5 (Nonuniform Hierarchy Theorem). *For every functions* $T, T' : \mathbb{N} \to \mathbb{N}$ *with* $\frac{2^n}{n} > T'(n) > 10T(n) > n$*,*

$$\boldsymbol{SIZE}(T(n)) \subsetneq \boldsymbol{SIZE}(T'(n))$$

3.1.1 Uniformly Generated Circuits

The main difference between the classes **P** and $\mathbf{P}_{/\mathbf{poly}}$ is that the latter contain languages for which there *exists* a circuit family to decide it, even if we have no way of constructing this family. That's the reason why pathological phenomena exist, such as that $\mathbf{P}_{/\mathbf{poly}}$ contains undecidable languages.

So, a first approach would be to try to restrict our study to the families that can actually be constructed (let's say by an efficient Turing Machine):

Definition 3.5 (P-Uniform Circuit Families). *A circuit family* $\{C_n\}$ *is* **P***-uniform if there is a polynomial-time Turing Machine that on input* 1^n *outputs the description of the circuit* C_n.

The problem is that if we restrict circuits to be **P**-uniform, the class $\mathbf{P}_{/\mathbf{poly}}$ (for which we know already that $\mathbf{P} \subset \mathbf{P}_{/\mathbf{poly}}$) collapses to **P**:

Theorem 3.6. *A language* L *is computable by a* **P***-uniform circuit family if and only if* $L \in \mathbf{P}$.

Even if it's known that every language has circuits of size $\mathcal{O}(2^n/n)$, it may be very difficult to construct them. If we place a uniformity condition on the circuits, that is, if we require them to be efficiently computable, then the circuit complexity of some languages might exceed 2^n.

[2] This theorem will be crucial for our results in Chapter 4!

Definition 3.6 (DC-Uniform Circuit Families)**.** *Let $\{C_n\}_{n\geq 1}$ be a circuit family. We say that it is Direct Connect Uniform (DC-Uniform) family if there is a polynomial-time algorithm that, given the pair $\langle n, i\rangle$, can compute the i^{th} bit of C_n's adjacency matrix representation. More precisely, a family $\{C_n\}_{n\in\mathbb{N}}$ is DC-Uniform if and only if the functions:*

- $SIZE(n)$*: Returns the size S of the circuit C_n.*
- $TYPE(n, i)$*: Returns the label of the i^{th} vertex of C_n. That is one of $\{\wedge, \vee, \neg, NONE\}$.*
- $EDGE(n, i, j)$*: Returns* 1 *if there is a directed edje in C_n from the i^{th} vertex to the j^{th} vertex.*

are computable in polynomial time.

A Turing Machine can now generate any required vertex of the circuit in polynomial time. This is an importantproperty , because we have a succinct representation of the circuit (in the terms of a T.M.), although it may have exponential size.

We can now give another characterization of the class **PH** (The Polynomial-Time Hierarchy):

Theorem 3.7. *A language $L \in$ **PH** iff L can be computed by a DC-Uniform circuit family $\{C_n\}_{n\in\mathbb{N}}$ that satisfies the following conditions:*

1. *that uses AND, OR, NOT gates.*
2. *that has size $2^{n^{\mathcal{O}(1)}}$ and constant depth.*
3. *its gates can have unbounded (exponential) fan-in.*
4. *its NOT gate appear only at the input level (that is, they are only applied directly to the input, and not to the result of other gates).*

Without the restriction of constant depth, the family describes *precisely* the class **EXP**!

3.1.2 Circuits computing Boolean Functions

We will also use Boolean Ciruits as a computational model for Boolean functions. Note that $\{\vee, \wedge, \neg\}$ is a *complete* set, so it can compute all Boolean functions.

Definition 3.7. *For a finite Boolean Function $f : \{0,1\}^n \to \{0,1\}$, we define the (circuit)* complexity *of f as the size of the smallest Boolean Circuit computing f (that is, $C(x) = f(x), \forall x \in \{0,1\}^n$).*

We can generalize the above definition for functions $f : \{0,1\}^* \to \{0,1\}^*$:

Definition 3.8 (Circuit Complexity)**.** *For a finite Boolean Function* $f : \{0,1\}^* \to \{0,1\}^*$*, and* $\{f_n\}$ *be such that* $f(x) = f_{|x|}(x)$ *for every* x*.*[3] *The (circuit)* complexity *of* f *is a function of* n *that represents the smallest Boolean Circuit computing* f_n *(that is,* $C_{|x|}(x) = f(x), \forall x \in \{0,1\}^*$*).*

It is clear by the definition that for each f_n we use a different circuit (having different number of inputs). The analogue of Definition 3.5, and Theorem 3.6, is that if f has a *uniform* (i.e. a polynomial-time algorithm that on input n produces a circuit computing f_n) sequence of polynomial-size circuits, then $f \in \mathbf{P}$. Also, any $f \in \mathbf{P}$ has a uniform sequence of polynomial-size circuits.

So, a super-polynomial circuit complexity for any (boolean) function in **NP**, would imply that $\mathbf{P} \neq \mathbf{NP}$.

3.1.3 Nondeterministic Circuits

We can also define variants of the Boolean Circuit model, in order to capture the notion of nondeterminism:

Definition 3.9 (Nondeterministic Circuits)**.** *A* nondeterministic *Boolean Circuit* $C(x, w)$ *is a Boolean Circuit that gets* x *as input, and a string* w *as a "witness". We say that* $C(x) = 1$ *if there exists a witness* w *such that* $C(x, w) = 1$*, and* $C(x) = 0$ *otherwise.*
Also, a co-nondeterminicstic *Boolean Circuit is defined similarly, with* $C(x) = 0$ *if there exists a witness* w *such that* $C(x, w) = 0$*, and* $C(x) = 1$ *if* $C(x, w) = 1$ *for all witnesses* w*.*

Definition 3.10 (SV Circuits)**.** *A* SV (single-valued) circuit *is a nondeterministic Boolean Circuit* $C(x, w)$ *has three possible outputs:* $1, 0$ *and "quit", such that for every input* $x \in \{0,1\}^n$ *either:*

- *for all* w*:* $C(x, w) \in \{1, quit\}$
- *for all* w*:* $C(x, w) \in \{0, quit\}$

We say that $C(x, w) = b \in \{0,1\}$*, if there exists (at least) one witness* w *such that:* $C(x, w) = b$*, and then we say that* w *is a* proof *that* $C(x) = b$*. When no such* w *exists, we say that* $C(x) = quit$*.*
Also, we say that C *is a* nondeterministic TSV (Total Single-Valued) *if* C *defines a total Boolean Function on* $\{0,1\}$ *(that is,* $\forall x \in \{0,1\}^n : C(x) \neq quit$*).*
Otherwise, we say that it is a nondeterministic PSV (Partial Single-Valued) *circuit.*

[3] By $|x|$ we denote the length of string x.

It's easy to see that if a TSV Circuit of size $\mathcal{O}(s(n))$ computes a Boolean function f, then f has also a nondeterministic *and* a co-nondeterministic circuit of size $\mathcal{O}(s(n))$.

We could intuitively compare the functions computed by TSV circuits, with the class $\mathbf{NP} \cap co\mathbf{NP}$ in the Nondeterministic T.M. model, or with the class $\mathbf{ZPP} = \mathbf{RP} \cap co\mathbf{RP}$ in the Probabilistic T.M. model.

Also, we can define *oracle* circuits, which have special gates called *oracle gates*, with arbitrary fan-in. A gate with fan-in s contributes size s to the circuit, and can be used for oracle access to a fixed language L. The output of the gate on a string x is 1 if $x \in L$, otherwise the output is 0. Nondeterministic and SV-nondeterministic oracle circuits are defined by combining the above definitions.

3.2 Circuit Lower Bounds

As we saw, the significance of proving lower bounds for this computational model is related to the famous "**P** vs **NP**" problem. In fact, if we ever prove that $\mathbf{NP} \nsubseteq \mathbf{P_{/poly}}$, then we'll have shown that $\mathbf{P} \neq \mathbf{NP}$ (since we now that $\mathbf{P} \subseteq \mathbf{NP}$ and $\mathbf{P} \subseteq \mathbf{P_{/poly}}$).

The main reason we prefer this computational model, instead of trying to prove lower bounds for Turing Machines, is that a Boolean circuit is considered a more direct or "pervasive" model, and also that we already know (since 1949) that some functions require very large circuits to compute:

Theorem 3.8 (C.E. Shannon)**.** *For every $n > 1$, $\exists f : \{0,1\}^n \to \{0,1\}$ that cannot be computed by a circuit C of size $\frac{2^n}{10n}$.*

Proof: The proof uses simple counting arguments. We know that the number of (boolean) functions from $\{0,1\}^n$ to $\{0,1\}$ is 2^{2^n}. Using the adjacency list representation, every circuit of size at most S can be represented by a string of $9S \log S$ bits. So, the number of such circuits is $2^{9S \log S}$. Let $S = 2^n/(10n)$, and see that the number of circuits of size S is at most $2^{9S \log S} < 2^{2^n}$.

Hence, the number of functions is clearly bigger than the number of circuits, so there is a function that cannot be computed by circuits of that size. $\square$

During the 1970s and 1980s, many researchers believed that circuit lower bounds are indeed the solution to the "**P** vs **NP**" problem, for reasons we mentioned above. Unfortunately, there is almost no progress on the matter: The best lower bound for an **NP** language is $5n - o(n)$, proved very recently (in 2005). On the other hand, there are better lower bounds for some special cases, i.e. some restricted classes of circuits, such as: bounded depth circuits, monotone circuits, and bounded depth circuits with "counting" gates. We will briefly discuss the first two:

3.2.1 Bounded Depth Circuits

Firstly, recall that the *depth* d of a circuit is the length of the longest directed path in it. Intuitively, the notion of depth captures the "parallel" time to decide a language, because it can be computed by enough "processors" in d stages.

We restrict here the depth d to be a constant, but we allow *unbounded fan-in* ($\wedge$-gates and $\vee$-gates taking any number of incoming edges).

Definition 3.11. *Let $PAR : \{0,1\}^n \to \{0,1\}$ be the* parity *function, which outputs the modulo 2 sum of an n-bit input. That is:*

$$PAR(x_1, ..., x_n) \equiv \sum_{i=1}^{n} x_i \pmod 2$$

We present a lower bound for PAR (proved by Furst, Saxe, Sipser in 1981):

Theorem 3.9. *For all constant d, PAR has* no *polynomial-size circuit of depth d.*

The above result (improved by Håstad and Yao) gives a relatively tight lower bound of $\exp\left(\Omega(n^{1/(d-1)})\right)$, on the size of n-input PAR circuits of depth d.

3.2.2 Monotone Circuits

We define the notion of monotone function and circuit, and we present a lower bound result for this model:

Definition 3.12. *For $x, y \in \{0,1\}^n$, we denote $x \preceq y$ if every bit that is* 1 *in x is also* 1 *in y. A function $f : \{0,1\}^n \to \{0,1\}$ is* monotone *if $f(x) \leq f(y)$ for every $x \preceq y$.*

Definition 3.13. *A Boolean Circuit is* monotone *if it contains only AND and OR gates, and no NOT gates. Such a circuit can only compute monotone functions.*

We consider the `CLIQUE` problem, which is known to be **NP**-complete. `CLIQUE` is a monotone funcion, since adding an edge to the graph cannot destroy any clique existed in it. We present a result proved during the '80s by Andreev, Alon and Boppana:

Theorem 3.10 (Monotone Circuit Lower Bound for `CLIQUE`)**.** *Denote by $CLIQUE_{k,n} : \{0,1\}^{\binom{n}{2}} \to \{0,1\}$ be the function that on input an adjacency matrix of an n-vertex graph G outputs* 1 *iff G contains an k-clique.*
There exists some constant $\epsilon > 0$ such that for every $k \leq n^{1/4}$, there is no monotone circuit of size less than $2^{\epsilon\sqrt{k}}$ that computes $CLIQUE_{k,n}$.

So, we proved a significant lower bound ($2^{\Omega(n^{1/8})}$). Similar lower bounds are known for functions in **P**.

The significance of the above theorem lies on the fact that there was some alleged connection between monotone and non-monotone circuit complexity (e.g. that they would be polynomially related). Unfortunately, Éva Tardos proved in 1988 that the gap between the two complexities is exponential.

We will use the above notions and results in the next chapter, by exploiting the "hardness" of computing certain Boolean functions, in order to define combinatorial constructions, known as Pseudorandom Generators, that produce sequences of *pseudorandom* bits. Their alleged randomness will be depended on the difficulty to compute a predefined *hard* function, in the sense that if we could "predict" (let's say by using a circuit) the next bit of such a sequence, we could use this circuit to easily compute efficiently the Boolean function, contradicting its hardness.

We end this chapter by an extra paragraph, which may be not prerequisite for the rest of our results in the technical sense (although some proofs in Chapters 6 and 7 are inspired from these notions), but it may answer to the reader's natural occuring question: *Why the circuit approach doesn't work, despite the intuition of so many researchers? Boolean Circuits seem to be a more clear and "pervasive" model than Turing Machines, but we finally face the same obstacles in the effort to prove that* $\mathbf{P} \neq \mathbf{NP}$. *Why?*

3.2.3 Epilogue: Where is the problem?

We discussed above that all research for circuit lower bounds was -finally- a dead-end. It is a fair question to ask (or even to try to prove) the cause of this difficulty.

A partial answer to that question was given by A. Razborov and S. Rudich in [RR94]. They connected circuit lower bounds with a notion called "Natural Proofs", and proved that a result for a lower bound using such techniques would imply the inversion of strong one-way functions.

Today, we have a lot of evidence that strong one-way functions *cannot* be inverted in subexponential time, so the techniques we use are *inherently* weak to prove general lower bounds for circuits.

We briefly give some definitions, and their main theorem:

Definition 3.14. *Let $\mathcal{P}$ be the predicate: "A Boolean function $f : \{0,1\}^n \to \{0,1\}$* doesn't *have n^c-sized circuits for some $c \geq 1$."*

Obviously, $\mathcal{P}(f) = 0, \forall f \in \mathbf{SIZE}(n^c)$ for a $c \geq 1$. We call this n^c-*usefulness.* Also:

Definition 3.15. *A predicate $\mathcal{P}$ is natural if:*

- *There is an algorithm* $M \in \boldsymbol{E}$ *such that for a function* $g : \{0,1\}^n \to \{0,1\}$*:* $M(g) = \mathcal{P}(g)$.
- *For a random function* g*:* $\boldsymbol{Pr}[\mathcal{P}(g) = 1] \geq \frac{1}{n}$

The main result, which expose the inherent problem of our approach, is the following:

Theorem 3.11. *If strong one-way functions exist, then there exists a constant* $c \in \mathbb{N}$ *such that there is* no n^c*-useful natural predicate* $\mathcal{P}$*.*

Chapter 4

Derandomization using Pseudorandom Generators

4.1 Pseudorandom Generators re-defined

We define again the notion of a pseudorandom generator, in a little different version, than the *secure* pseudorandom generators defined in the Introductory Chapter, for the purposes of Derandomization. The main differences in this variation are that:

1. We allow the generator to run in *exponential* time, instead of polynomial.

2. We use *nonuniform* distinguishers (circuits), instead of the classical model of probabilistic polynomial-time Turing Machines.

So, this is a *relaxation* of the original definition, which allow us to construct such generators under weaker conditions, for our derandomization purposes:

Definition 4.1 (Pseudorandom Generators (PRGs))**.** *A distribution R over $\{0,1\}^m$ is an (S, ϵ)-pseudorandom (for $S \in \mathbb{N}$, $\epsilon > 0$) if for every circuit C, of size at most S:*

$$|\boldsymbol{Pr}[C(R) = 1] - \boldsymbol{Pr}[C(U_m) = 1]\,| < \epsilon$$

where U_m denotes the uniform distribution *over* $\{0,1\}^m$

If $S : \mathbb{N} \to \mathbb{N}$, a 2^n-time computable function $G : \{0,1\}^* \to \{0,1\}^*$ is an $S(\ell)$-pseudorandom generator if $|G(z)| = S(|z|)$ for every $z \in \{0,1\}^*$ and for every $\ell \in \mathbb{N}$ the distribution $G(U_\ell)$ is $(S^3(\ell), \frac{1}{10})$-pseudorandom.

In the above definition:

- The choices of the constants 3 and $\frac{1}{10}$ are (obviously) arbitrary, and made for convenience.

- The functions $S : \mathbb{N} \to \mathbb{N}$ will be considered *time-constructible* and *non-decreasing*.

4.2 Derandomization Results

The reason we use pseudorandom generators is that they can be used to efficiently simulate every randomized algorithm in **BPTIME**. The only requirement is the existence of such constructions.

The general method we use to derandomize a **BPP** algorithm is quite simple: Firstly, we know that there is a bounded-time Probabilistic Turing Machine "representing" this algorithm in this computational model, using a string of (truly) random bits of length $\rho(n)$. The naïve approach would be to *enumerate* all possible random strings, which are $2^{\rho(n)}$. So, we have:

$$\mathbf{BPP} \subseteq \mathbf{DTIME}(2^{\rho(n)} \cdot poly(n))$$

which is -unfortunately- exponential in the general case.

But, if we replace the random string with the output of a Pseudorandom Generator, we only have to enumerate all possible strings of the *seed*, which has much smaller length $\ell(n)$[1] (assuming that the generator has stretch function $S(\ell(n)) \geq \rho(n)$, and if it is equal we take it as it is, if it's bigger we take the $\rho(n)$-bit prefix of the output). Then, we have a result of the form:

$$\mathbf{BPP} \subseteq \mathbf{DTIME}(2^{\ell(n)} \cdot poly(n))$$

and if $\ell(n) << \rho(n)$ we could even achieve a polynomial simulation of every **BPP** algorithm (e.g. if $\ell(n) = \log n$). The above ideas can be formalized in the following result:

Theorem 4.1. *Suppose that there exists an $S(\ell)$-pseudorandom generator for a time-constructible nondecreasing $S : \mathbb{N} \to \mathbb{N}$. Then, for every polynomial-time computable function $\ell : \mathbb{N} \to \mathbb{N}$, and for some constant c:*

$$\mathbf{BPTIME}(S(\ell(n)) \subseteq \mathbf{DTIME}(2^{c\ell(n)})$$

Proof: We already know that $L \in \mathbf{BPTIME}(S(\ell(n))$ if $\exists$ algorithm A which runs in time $cS(\ell(n))$ for some constant c, and satisfies the following condition[2]:

$$Pr\left[A(x,r) = L(x)\right] \geq \frac{2}{3}$$

[1] Because we use our generator along with a **BPP** algorithm, the length of its seed is function of the length of the input, exactly as the length $\rho(n)$ of the Probabilistic T.M.'s random string.

[2] We remind that $L(x)$ is the characteristic function of L.

The main idea is to *replace* the random string $r \in \{0,1\}^m$ ($m \leq S(\ell(n))$) used by the PTM which computes A with a string $G(z)$, produced by picking a random $z \in \{0,1\}^{\ell(n)}$. Then A will not "detect" the switch for the most of the time, so the probability $\frac{2}{3}$ will not drop below $\frac{2}{3} - \frac{1}{10}$, which is greater than $\frac{1}{2}$. So we don't have to simulate all r's, we only have to enumerate over all strings $G(z)$, for $z \in \{0,1\}^{\ell(n)}$, and check whether or not the *majority* make A accept.

So, let B an algorithm computed by a Determinictic Turing Machine. On input $x \in \{0,1\}^n$, B will go over all $z \in \{0,1\}^{\ell(n)}$, and will compute $A(x, G(z))$ (that is, A on input x, using $G(z)$ as random string), and output the majority answer.

We claim that for sufficiently large n, the fraction of z's for which $A(x, G(z)) = L(x)$ is at least $\frac{2}{3} - \frac{1}{10}$. (That suffices to show that $L \in$ **DTIME**$(2^{c\ell(n)})$, because we can "feed" the algorithm with the correct answer for finitely many inputs):

Suppose, for the sake of contradiction, that exists a infinite sequence of x's such that:

$$Pr\left[A(x, G(z)) = L(x)\right] < \frac{2}{3} - \frac{1}{10}$$

Then, there exists a *distinguisher* for the pseudorandom generator: we can construct a circuit computing the function $r \mapsto A(x, r)$, where x is "embedded" into the circuit (that is possible because we use nonuniformity). The circuit will have size $\mathcal{O}(S(\ell(n))^2)$, shich is surely smaller than $S(\ell(n))^3$, for sufficiently large n. We have our contradiction, so we proved that $Pr\left[A(x, G(z)) = L(x)\right] < \frac{2}{3} - \frac{1}{10}$, and our claim is valid. □

- In the above proof we see the necessity of running the PRG in exponential time. The derandomized algorithm enumerates over all possible z's of length ℓ, so it needs exponential (in ℓ) time.
- Also, allowing the generator to run in exponential time makes it more "easy" to prove the existence of such a PRG than allowing it run in polynomial-time, as in secure PRGs, used in Cryptography.

As special cases of the above theorem, we can obtain the following simulations, which make clear the importance of constructing such pseudorandom generators, in order to achieve a full (or even partial) derandomization of **BPP**:

Theorem 4.2.
- *If there exists a* $2^{\epsilon\ell}$*-pseudorandom generator for some constant* $\epsilon > 0$*, then* **BPP** $=$ **P**.
- *If there exists a* 2^{ℓ^ϵ}*-pseudorandom generator for some constant* $\epsilon > 0$*, then* **BPP** $\subseteq$ **QuasiP** $=$ **DTIME**$(2^{poly\log(n)})$.

- *If for every $c > 1$ there exists an ℓ^c-pseudorandom generator, then* $\mathbf{BPP} \subseteq \mathbf{SUBEXP} = \bigcap_{\epsilon>0} \mathbf{DTIME}(2^{n^\epsilon})$.

It is fair to question whether such generators indeed exist, and how we can prove this existence. We can connect the existence of some pseudorandom generator to the *hardness* of a function, that is, how "difficult" is to compute it.

We can measure the hardness using known complexity measures, such the number of steps of a Turing Machine, or the minimum size of a circuit which computes it.

4.3 Pseudorandomness using Hardness of Functions

We introduce the notion of average-case and worst-case harness of functix‘ons. This will be a useful tool for "measuring" the size of the minimum Boolean Circuit computing a function:

Definition 4.2 (Average-case and Worst-case hardness)**.** *For $f : \{0,1\}^n \to \{0,1\}$, and $\rho \in [0,1]$ we define the ρ-*average-case hardness *of f, denoted $H^\rho_{avg}(f)$, to be the largest S that for every circuit C of size at most S:*

$$Pr_{x\in\{0,1\}^n}\left[C(x) = f(x)\right] < \rho$$

We define the worst-case hardness *of f, denoted $H_{wrs}(f)$ to equal $H^1_{avg}(f)$, and the* average-case hardness *of f, denoted $H_{avg}(f)$ to equal:*

$$\max\{S | H^{1/2+1/S}_{avg}(f) \geq S\}$$

That is, $H_{avg}(f)$ is the largest number S such that:

$$Pr_{x\in\{0,1\}^n}\left[C(x) = f(x)\right] < \frac{1}{2} + \frac{1}{S}$$

for every Boolean Circuit C on n inputs with size at most S.

The following result will be (just) mentioned, a very important "Hardness Amplification" theorem, which we will use later:

Theorem 4.3. *Let $f \in \mathbf{E}$* [3] *be such that $H_{wrs}(f)(n) \geq S(n)$ for some time-constructible nondecreasing $S : \mathbb{N} \to \mathbb{N}$. Then, there exists a function $g \in \mathbf{E}$ and a constant $c > 0$ such that:*

$$H_{avg}(g)(n) \geq S(n/c)^{1/c}$$

for every sufficiently large n.

[3] Recall that $\mathbf{E} = \mathbf{DTIME}\left(2^{\mathcal{O}(n)}\right)$.

We will now use the assumption of average-case hardness of certain functions, to construct pseudorandom generators. By using (quantitatively) stronger assumptions, we construct stronger generators. The strongest assumption will yield a $2^{\Omega(\ell)}$-pseudorandom generator, implying that **BPP** = **P**.

The main theorem is the following, finally proved by Chris Umans in 2003, after enormous efforts by many researchers. It is based on a pseudorandom generator constructed by R.Shaltiel and Umans in [SU05]. This construction will not be mentioned here, instead we will introduce in the next section the first pseudorandom generator from average-case hardness, presented by N.Nisan and A. Wigderson in 1988, which is strong enough to imply a version of Theorem 4.2 for "average" hardness of functions.

Theorem 4.4 (PRGs from average-case hardness)**.** *Let $S : \mathbb{N} \to \mathbb{N}$ be time-constructible and non-decreasing. If there exists $f \in \boldsymbol{E}$ such that $\forall n : H_{avg}(f)(n) \geq S(n)$, then there exists an $S(\delta\ell)^{\delta}$-peudorandom generator for some constant $\delta > 0$.*

If we combine the above Theorem 4.4 with Theorem 4.3, we can obtain the following theorem, which strengthen the possibilities that derandomization of probabilistic algorithms is possible.

Theorem 4.5 (Derandomizing under worst-case assumptions)**.** *Let $S : \mathbb{N} \to \mathbb{N}$ be time-constructible and nondecreasing. If there exists $f \in \boldsymbol{E}$ such that $\forall n : H_{wrs}(f)(n) \geq S(n)$, then there exists a $S(\delta\ell)^{\delta}$-peudorandom generator for some constant $\delta > 0$.*
In particular, the following hold:

1. *If there exists $f \in \boldsymbol{E}$ and $\epsilon > 0$ such that $H_{wrs}(f)(n) \geq 2^{\epsilon n}$, then* ***BPP** = **P**.*

2. *If there exists $f \in \boldsymbol{E}$ and $\epsilon > 0$ such that $H_{wrs}(f)(n) \geq 2^{n^{\epsilon}}$, then* ***BPP** $\subseteq$ **QuasiP**.*

3. *If there exists $f \in \boldsymbol{E}$ such that $H_{wrs}(f)(n) \geq n^{\omega(1)}$, then* ***BPP** $\subseteq$ **SUBEXP**.*

We can replace **E** with **EXP** in (2) and (3) of Theorem 4.5, which is very important, because **EXP** contains many classes we believe to have hard problems, such as **NP**, **PSPACE**, and even $\oplus$**P**.

4.4 The Nisan-Wigderson construction

We will focus on the construction of the most important and useful for our future results pseudorandom generator, the Nisan-Wigderson (NW) generator, introduced in [NW94].

This generator streches a short seed into a long string that looks random to any algorithm from a complexity class $\mathcal{C}$, using a function that is hard for $\mathcal{C}$.

The simple approach would be to take as many indepedent parts of the seed as we can, "feed" to the hard function, and concatenate them to take the output. Any distinguisher that could tell the difference between the output and the Uniform Distribution, could be used to construct a circuit that computes the function, constradicting its hardness assumption.

But such a generator hasn't the possibility to strech the seed very much (more than a multiple of the seed). In order to have non-trivial derandomization results, our generator's output must be *exponentially* larger than the input.

So, instead of taking indepedent parts of the seed as arguments for the hard function, we can take them *partially depedent*, but we still have to control and bound the "amount" of depedence, so we will distribute our seed into sets which have the same cardinality, and the intersection of every pair is bounded. Such combinatorial structures are known as Designs:

Definition 4.3 (Combinatorial Designs)**.** *A family $\mathcal{S} = \{S_1, \dots, S_m\}$, where each $S_i \subset \{1, \dots, \ell\}$ is an (ℓ, n, k)-design, if:*

1. $|S_j| = n$*, for every* j
2. $|S_i \cap S_j| \leq k$*, for all* $i \neq j$

These designs can be efficiently constructible:

Lemma 4.6. *For every integer n and fraction $\gamma > 0$, there is a $(\ell, n, \log m)$-design $\{S_1, \dots, S_m\}$ over $\{1, \dots, \ell\}$, where $\ell = \mathcal{O}(n/\gamma)$ and $m = 2^{\gamma n}$. Such a design can be constructed in $\mathcal{O}(2^\ell \ell m^2)$ steps.*

Now, we are ready to formally define our generator function:

Definition 4.4 (Nisan-Wigderson Generator)**.** *Let $\mathcal{S} = \{S_1, \dots, S_m\}$ a (ℓ, n, d)-design and $f : \{0,1\}^n \to \{0,1\}$. The NW-generator is the function $NW^f_{\mathcal{S}} : \{0,1\}^\ell \to \{0,1\}^m$ that maps every $z \in \{0,1\}^\ell$ to*

$$NW^f_{\mathcal{S}}(z) = f(z_{|S_1}) \circ f(z_{|S_2}) \circ \cdots \circ f(z_{|S_m})$$

where $z_{|S_i}$ denotes the restriction of z to the coordinates indexed by S_i. For example, if $z = 10101$ and $S_i = \{1, 3, 5\}$, then $z_{|S_i} = 111$.

Recall from Definition 4.1 that our generator is allowed to run in *exponential* time in the size of its input, and we use circuits as distinguishers.

We will prove that $NW^f_{\mathcal{S}}$ is a pseudoranom generator:

The main fact that implies the pseudorandomness of NW-Generator, is that a possible distinguisher can be used to build a circuit which violates the given hardness of f:

Theorem 4.7. *Let $f : \{0,1\}^n \to \{0.1\}$ be a Boolean function and $\mathcal{S} = \{S_1, \ldots, S_m\}$ be a $(\ell, n, \log m)$-design. Suppose that $D : \{0,1\}^m \to \{0,1\}$ is such that:*

$$\left|\mathbf{Pr}_{r\in\{0,1\}^m}[D(r) = 1] - \mathbf{Pr}_{z\in\{0,1\}^\ell}[D(NW^f_{\mathcal{S}}(z)) = 1]\right| > \epsilon$$

Then, there exists a circuit C of size $\mathcal{O}(m^2)$ such that:

$$\left|\boldsymbol{Pr}_{x\in\{0,1\}^n}[D(C(x)) = f(x)] - 1/2\right| \geq \frac{\epsilon}{m}$$

Proof: The main idea of the proof is that if we find a circuit D that can (as the theorem states) distinguish the output of the NW-generator from the Uniform Distribution, then we can use it to build another circuit C computing a large fraction of f' s outputs, violating its *hardness assumption*, and so we are led to a contradiction. Such a distinguisher can find a bit of the outut of the generator where the distinction is noticeable. On such a bit, D is distinguishing $f(x)$ from a random bit, and such a distinguisher can be used as a predictor for f.

We will use a technique known as the emphhybrid argument: We define $m+1$ distributions $H_0, \ldots, H_m$ as follows: we sample a string $v = NW^f_{\mathcal{S}}(z)$ for a random z, and then a string $r \in \{0,1\}^m$ according to the Uniform Distribution. Each H_i is defined by taking i bits from v and the last $m-i$ bits from r. So, H_0 is the Uniform Distribution over $\{0,1\}^m$, and H_m is distributed as $NW^f_{\mathcal{S}}(z)$. According to our hypothesis, there is a $b_0 \in \{0,1\}$ such that:

$$\left|\mathbf{Pr}_{r\in\{0,1\}^m}[D'(r) = 1] - \mathbf{Pr}_{y\in\{0,1\}^\ell}[D'(NW^f_{\mathcal{S}}(y)) = 1]\right| > \epsilon$$

where $D'(x) = b_0 \oplus D(x)$. We observe that:
$\varepsilon \leq \mathbf{Pr}[D'(NW^f_{\mathcal{S}}(y)) = 1] - \mathbf{Pr}[D'(r) = 1] = \mathbf{Pr}[D'(H_m) = 1] - \mathbf{Pr}[D'(H_0) = 1]$

$$= \sum_{i=1}^{m}(\mathbf{Pr}[D'(H_i) = 1] - \mathbf{Pr}[D'(H_{i-1}) = 1])$$

So, there exists an i such that:

$$\mathbf{Pr}[D'(H_i) = 1] - \mathbf{Pr}[D'(H_{i-1}) = 1] \geq \frac{\varepsilon}{m}$$

Now, recall that: $H_{i-1} = f(z_{|S_1}) \circ \cdots \circ f(z_{|S_{i-1}}) \circ r_i r_{i+1} \circ \cdots \circ r_m$ and
$H_i = f(z_{|S_1}) \circ \cdots \circ f(z_{|S_{i-1}}) \circ f(z_{|S_{i-1}}) \circ r_{i+1} \circ \cdots \circ r_m$.

We can assume without loss of generality that $S_i = \{1, \ldots, \ell\}$. Then we can see $z \in \{0,1\}^t$ as a pair (x, y) where $x = z_{|S_i} \in \{0,1\}^\ell$ and $y = z_{|[t]\setminus S_i} \in \{0,1\}^{t-\ell}$. For every $j < i$ and $z = (x, y)$ we define $f_j(x, y) = f(z_{|S_i})$. Observe that f_j depends on $|S_i \cap S_j| \leq \log m$ bits onf x and on

$\ell - |S_i \cap S_j| \geq \ell - \log m$ bits of y. With this notation we have:

$$\mathbf{Pr}_{x,y,r_{i+1},\ldots,r_m}[D'(f_1(x,y),\ldots,f_{i-1}(x,y),f(x),\ldots,r_m)=1]-$$
$$-\mathbf{Pr}_{x,y,r_{i+1},\ldots,r_m}[D'(f_1(x,y),\ldots,f_{i-1}(x,y),r_i,\ldots,r_m)=1] > \frac{\varepsilon}{m}$$

The above means that when D' is given a string that contains $f_j(x,y)$ for $j < i$ in the first $i-1$ entries, then D' is more likely to accept the string if contains $f(x)$ in the i-th entry than if it contains a random bit in the i-th entry. This is food anough to (almost) get a predictor for f.

Consider the following algorithm:

Algorithm A
Input: $x \in \{0,1\}^\ell$

1. Pick random $r_i, \ldots, r_m \in \{0,1\}$
2. Pick random $y \in \{0,1\}^{t-\ell}$
3. Compute $f_1(x,y)\ldots,f_{i-1}(x,y)$
4. If $D'(f_1(x,y)\ldots,f_{i-1}(x,y),r_i,\ldots,r_m)$ output r_i
5. Else output $1-r_i$

Let $A(x,y,r_1,\ldots,r_m$ the output of A on input x and random choices $y, r_1, \ldots, r_m$. We have:

$$\mathbf{Pr}_{x,y,r}[A(x,y,r)=f(x)] =$$
$$\mathbf{Pr}_{x,y,r}[A(x,y,r)=f(x)|r_i=f(x)]\cdot\mathbf{Pr}_{y,r_i}[r_i=f(x)]+$$
$$+\mathbf{Pr}_{x,y,r}[A(x,y,r)=f(x)|r_i\neq f(x)]\cdot\mathbf{Pr}_{y,r_i}[r_i\neq f(x)]=$$
$$=\frac{1}{2}\mathbf{Pr}_{x,y,r}[D'(f_1(x,y),\ldots,f_{i-1}(x,y),r_i,\ldots,r_m=1|f(x)=r_i]+$$
$$+\frac{1}{2}\mathbf{Pr}_{x,y,r}[D'(f_1(x,y),\ldots,f_{i-1}(x,y),r_i,\ldots,r_m=0|f(x)\neq r_i]=$$
$$=\frac{1}{2}+\frac{1}{2}\mathbf{Pr}_{x,y,r}[D'(f_1(x,y),\ldots,f_{i-1}(x,y),r_i,\ldots,r_m=1|f(x)=b]-$$
$$-\frac{1}{2}\mathbf{Pr}_{x,y,r}[D'(f_1(x,y),\ldots,f_{i-1}(x,y),r_i,\ldots,r_m=1|f(x)\neq b]=$$

$$= \frac{1}{2} + \mathbf{Pr}_{x,y,r}[D'(f_1(x,y),\ldots,f_{i-1}(x,y),r_i,\ldots,r_m = 1|f(x) = b] -$$
$$-\frac{1}{2}\mathbf{Pr}_{x,y,r}[D'(f_1(x,y),\ldots,f_{i-1}(x,y),r_i,\ldots,r_m = 1|f(x) = b] -$$
$$-\frac{1}{2}\mathbf{Pr}_{x,y,r}[D'(f_1(x,y),\ldots,f_{i-1}(x,y),r_i,\ldots,r_m = 1|f(x) \neq b] =$$
$$= \frac{1}{2} + \mathbf{Pr}[D'(H_i) = 1] - \mathbf{Pr}[D'(H_{i-1}) = 1] \geq \frac{1}{2} + \frac{\varepsilon}{m}$$

So A is good, and it is worthwile to see whether we can get an efficient implementation. Since we have:

$$\mathbf{Pr}_{x,y,r}[A(x,y,r) = f(x)] \geq \frac{1}{2} + \frac{\varepsilon}{m}$$

there surely exist $c_1,\ldots,c_m$ to give to $r_1,\ldots,r_m$, and a fixed value w to give to y such that:

$$\mathbf{Pr}_{x,r}[A(x,w,c_1,\ldots,c_m) = f(x)] \geq \frac{1}{2} + \frac{\varepsilon}{m}$$

Since w is fixed, in order to implement A we only have to compute $f_j(x,w)$ given x. However, for each j, $f_j(x,w)$ is a function that depends only on $\leq \log m$ bits of x, and so is computable by circuits of size $\mathcal{O}(m)$. Even composing $i-1 < m$ such circuits, we still have that the sequence $f_1(x,y),\ldots,f_{i-1}(x,y),c_1,\ldots,c_m$ can be computed, given x, by a circuit C of size $\mathcal{O}(m^2)$.

Finally, we notice that at this point $A(x,w,c)$ is doing the following: outputs the XOR between c_i and the complement of $D'(C(x))$. Since c_i is fixed, either $A(x,w,c)$ always equals $D(C(x))$, or one is the complement of the other. In either case, the conclusion follows. $\square$

Chapter 5

Uniform Derandomization of BPP

5.1 Main Theorem

In this chapter we reach the main topic of this thesis, the derandomization of probabilistic classes under uniform complexity assumptions. The first result concerns **BPP**, and how we can get a non-trivial derandomization of this class under a *uniform* hardness assumption. Specifically, the main theorem states that unless every exponential-time problem can be solved in probabilistic polynomial time, we can partially derandomize **BPP**, by simulating every $L \in \mathbf{BPP}$ by a subexponential algorithm. This is a "Low-End" result, since it provides us a non-trivial derandomization of **BPP** under a plausible hardness assumption[1].

This result comes with two "defects": Firtsly, this simulation doesn't works everywhere (or even almost everywhere), but only for infinitely many

[1] Recall that in the curve of Hardness-Randomness Trade-offs we have the:

- "**High End**:" What (usually strong) assumption must we make in order to have a *full* derandomization of a probabilistic complexity class?
- "**Low End**:" What is the *weakest* assumption we can make, and still have some version of non-trivial derandomization of a probabilistic complexity class?

input lengths (i.o. complexity[2]). Also, it may fail on a negligible fraction of inputs even of these lengths (called a "*heuristic simulation*").

This simulation will not succeed only for inputs chosen according to the Uniform Distribution, but for inputs chosen according to *every* distribution that can be sampled in polynomial time (*P-sampleable*).

The following theorem was proved by R. Impagliazzo and A. Wigderson in 1998, and it was partially based on a 1993's result of L. Babai, L. Fortnow, C. Lund and A. Wigderson. After that, new efforts improved and extended these ideas, and led to similar derandomizations of **RP** and **AM**. Ramifications on these results were presented until very recently. The main theorem states that:

Theorem 5.1. *If* $\mathbf{EXP} \neq \mathbf{BPP}$*, then, for every* $\epsilon > 0$*, every* **BPP** *algorithm can be simulated deterministically in (subexponential) time* 2^{n^ϵ} *so that, for infinitely many* n*'s, the simulation is correct on at least* $1 - \frac{1}{n}$ *fraction of all inputs of size* n*.*

We can view the above as a *gap theorem* on Derandomization: Either $\mathbf{BPP} = \mathbf{EXP}$, that is, randomness solves any hard problem (is a computational "panacea"), or every problem in **BPP** admits a non-trivial subexponential derandomization, that works on almost all instances.

5.2 Proof of Theorem 5.1

The proof will be completed in two parts: The first will use the assumption that $\mathbf{EXP} \nsubseteq \mathbf{P_{/poly}}$, it was given by Babai, Fortnow, Lund and Wigderson [BFNW93], and lied on techniques used in significant previous results on Interactive Proof Systems in [BFL91] (namely, that $\mathbf{MIP} = \mathbf{NEXP}$). The case of $\mathbf{EXP} \subseteq \mathbf{P_{/poly}}$ (which is considered high improbable by the scientific community) was proved by Impagliazzo and Wigderson [IW98], and used multiple results from Complexity Theory, as well as the Nisan-Wigderson construction. We start by showing the first part (the "easy" one):

[2]Indeed, we have two types of hardness:

- If f has circuit complexity exceeding S **infinitely often** (i.o.), we mean that *there are infinite many* n's, such no circuit of size $S(n)$ can compute f correctly on all inputs of length n.
- If f has circuit complexity exceeding S **almost everywhere** (a.e.), we mean that *for all but finitely many* n's, such no circuit of size $S(n)$ can compute f correctly on all inputs of length n.

It is not known whether an "infinitely-often" type of hardness implies a corresponding "almost-everywhere" hardness.

5.2.1 First Part of the Proof

Theorem 5.2 ([BFNW93]). *If* **EXP** $\nsubseteq$ $\mathbf{P}_{/\mathbf{poly}}$ *, then* **BPP** $\subseteq$ **SUBEXP** *for infinitely many input lengths.*

In order to compose the complete proof, we give a series of lemmata. The first is a result from [BFL91], regarding the power of **EXP** provers:

Lemma 5.3 ([BFL91]). *Every language* $L \in$ **EXP** *has a multi-prover interactive proof system, where the "honest" provers are limited to computing within deterministic exponential time.*

Lemma 5.4. *If* **EXP** $\subseteq \mathbf{P}_{/\mathbf{poly}}$*, then* **EXP** $=$ **MA**.

Proof: By Lemma 5.3, we imply that a simulation of an **EXP** language by a multi-prover interactive protocol will need only **EXP**-strong provers. Also, using the hypothesis, for these provers we can find two polynomial-sized circuits C_1 and C_2 computing them.
So, *Merlin* gives *Arthur* C_1 and C_2. Then, *Arthur* simulates the verifier V using the two circuits for the two provers. This is an **MA** protocol for any **EXP** language. □

***Proof** (of Theorem 5.2).* : Let L be an **EXP**-complete language. We encode a the set L_n as a boolean function f. Let p be a prime greater than n, and let g be the unique multilinear extension of f to $\mathbb{Z}_p^n \to \mathbb{Z}_p$. [3]

The following lemma gives us information about g:

Lemma 5.5 ([GL89]).

1. *If* **BPP** *doesn't have subexponential simulation for infinitely many input sizes, there is a family of polynomial-size circuits computing* g *for all but a* $1/3n$ *fraction of the inputs of length* n*.*

2. *If* **BPP** *is not in* **SUBEXP***, there for an infinite number of input lenghts* n*, there is a polynomial-size circuit computing* g *for all but a* $1/3n$ *fraction of the inputs of length* n*.*

Assume, now, that **BPP** does not have a i.o. simulation. Then, by Lemma 5.5 we have a family D_n of circuits (polynomial-size) computing g for all but $1/3n$ fraction of the inputs. Since g is random self-reducible, we can create the following randomized polynomial-size circuit family for g: C_n will use random inputs to generate the random self-reduction of g and use the D_n circuit for those queries. The probability of the random self-reduction queries onte of the strings that D_n fails to compute correctly is bounded by $(n+1)/3n < 2/5$ for almost every n.

[3]Recall that if we extend a boolean function to a multilinear extension g over $\mathbb{Z}_p$, we obtain a random self-reducible f-hard and $\mathbf{PSPACE}^f$-easy function g. Each function $h : \{0,1\}^s \to \mathbb{Q}$ has a *unique* multilinear extension $\tilde{h} : \mathbb{Q}^s \to \mathbb{Q}$.

We can replace (using the techiques of the proof of $\mathbf{BPP} \subset \mathbf{P}_{/\mathbf{poly}}$) the randomness by non-uniformity: we can use the known amplification techniques to reduce the error below 2^{-n}. Then, there must be a single random sequence (the advice string) that gives a correct answer for all inputs. We encode this string into the circuit. It has polynomial length, so the circuit stays polynomial-size. □

5.2.2 Second Part of the Proof

The proof can be completed in four steps:

1. Assume that `PERMANENT` is **EXP**-complete.
2. Build a pseudorandom generator (in fact a sequence of generators) with super-polynomial output size using `PERMANENT` as a hard function.
3. Run the simulation for every $L \in \mathbf{BPP}$, by replacing the random string with the possible outputs of the generator.
4. Remove the oracle from f_n, by using its properties.

Now, we can examine each step in detail:

Step 1: The Hard Function

In our proof we will make the assumption that $\mathbf{EXP} \subseteq \mathbf{P}_{/\mathbf{poly}}$, since otherwise every **BPP** algorithm can be simulated deterministically in subexponential time infinitely often. This is a known result proved by L.Babai, L.Fortnow, N.Nisan and A.Wigderson, which imply the conclusion of out theorem (something even stronger, to be accurate):

.. So, we will proceed under the -*fair*- assumption that $\mathbf{EXP} \subseteq \mathbf{P}_{/\mathbf{poly}}$. Then, Meyer's Theorem (Theorem **??**) implies that $\mathbf{EXP} = \Sigma_2^p \subseteq \mathbf{PH}$, and by Toda's Theorem (Theorem **??**) we have that:

$$\mathbf{EXP} \subseteq \mathbf{PH} \subseteq \mathbf{P}^{\texttt{PERMANENT}}$$

since `PERMANENT` problem is **#P**-complete problem, just as #`SAT` in the original theorem's formulation.

By the above inclusion, we have that every **EXP** language can be transformed in a `PERMANENT` instance in polynomial time, so `PERMANENT` is **EXP**-complete under polynomial-time reductions.

This observation is very important for the procedure, since `PERMANENT` has two nice properties:

Firstly, it is *random self-reducible*, that is, solving the problem on any input x can be reduced to solving it on a sequence of random inputs $y_1, y_2, \ldots,$ where each y_i is uniformly distributed among all inputs.

Secondly, one can solve PERMANENT in polynomial time using an oracle for the permanent of smaller matrices. This property is called *downward self-reducibility*. We give the formal definition:

Definition 5.1 (Downward Self-Reducibility)**.** *A function f is* downward self-reducible *if there is a polynomial-time Turing Machine M, such that:*

$$\forall n \forall x \in \{0,1\}^n : M^{f_{n-1}}(x) = f(x)$$

where by f_k we denote an oracle that solves f on inputs of size at most k. Using Turing Reductions (Definition 2.6), we can rewrite the above as:

$$f_n \leq_T^p f_{n-1}$$

Of course, we can use any Σ_2^p-hard function in **EXP** with the above two properties.

Step 2: The Pseudorandom Generator

Now, let f be a function with all the above properties, and f_n be the restriction of f to inputs of length n. For each input size n, we will construct a pseudorandom generator (similar to the Nisan-Widgerson generator we constructed previously), using PERMANENT function f as a hard function.

In order to construct such generators, and connect them somehow, we need to formalize the notion of "construction", "construction problems", and the reductions among them.

In general, a *construction problem* $A = \{A_n\}$ is a family of non-empty subsets $A_n \subset \{0,1\}^*$.

In these terms, we can define the *approximation* of f by circuits:

Definition 5.2. *Let $f : \{0,1\}^* \to \{0,1\}^*$ and $\epsilon : \mathbb{N} \to [0,1]$. The construction problem $C^{f,\epsilon}$ can be defined as follows: each $C_n^{f,\epsilon}$ contains all circuits C with n inputs satisfying:*

$$\boldsymbol{Pr}_{x \in \{0,1\}^n}[C(x) = f(x)] \geq \epsilon(n)$$

By writing C^f, we mean $C^{f,1}$, i.e. circuits computing f.

In the same way, we can define the notion of *distinguishers* as construction problems:

Definition 5.3. *Let $m : \mathbb{N} \to \mathbb{N}$, $\epsilon : \mathbb{N} \to [0,1]$, and $G = \{G_n : \{0,1\}^{m(n)} \to \{0,1\}^n\}$. $D^{G,\epsilon}$ can be defined as follows: each $D_n^{G,\epsilon}$ contains all circuits D with n inputs satisfying:*

$$\boldsymbol{Pr}_{y \in \{0,1\}^{m(n)}}[D(G(y)) = 1] - \boldsymbol{Pr}_{x \in \{0,1\}^n}[D(x)) = 1] \geq \epsilon(n)$$

That is, a distinguisher D is a circuit family that, for each n, cannot be fooled by the generator G, and can "distinguish" whether a string is an output of G and not chosen at random[4], with non-negligible probability.

A construction problem A can be generated *efficiently*, if there exists a probabilistic polynomial-time algorithm taking two inputs n, α, and produces a member of A_n with probability at least $1 - \alpha$, taken over the algorithm's coin tosses.

For our purposes, it is necessary to "relate" somehow a construction problem with another. So we need a kind of reduction:

Definition 5.4. *An* efficient construction *of B from A is a probabilistic polynomial-time algorithm that $\forall n \forall \alpha$, $\alpha \in A_n$, outputs a member of B_n with probability at least $1 - \alpha$. If such a construction exists, we denote it by $A \to B$. When we allow to the construction to make also queries to an oracle O, we denote it $A \to^O B$.*

The above definition implies that if $A \to B$, and A is efficiently constructible, then B is also efficiently constructible. Note that $\to$ is a *transitive* relation.

In the context of the above, we can state a more strict definition of random self-reducibility (which we introduced in Step 1):

Definition 5.5 (Random Self-Reducibility)**.** *A function $f : \{0,1\}^* \to \{0,1\}^*$ is random self-reducible if for some $c > 0$:*

$$C^{f,1-n^{-c}} \to C^f$$

Now, we can begin constructing our Generator, using a similar method as we used in Section ..., where we "built" the Nisan-Wigderson construction.

Let $d \in \mathbb{N}$ be the output of our generator G_d (that is d is a parametrization of our generators).

- *Direct product function:*
  Let $n_1 = n^{c+2}$. Also define $g : \{0,1\}^{n_1} \to \{0,1\}^{n^{c+1}}$ by $g(x_1, \ldots, x_n) = f(x_1) \ldots f(x_n)$.
- *Hard-core bit:*
  Let $n_2 = n_1 + n^{c+1}$. Any such string can be viewed as an input x to g (where $|x| = n_1$), and a string r of length n^{c+1}. Then, we define[5] $h(x,r) = \langle g(x), r \rangle$.
- *Almost disjoint sets generator:*
  Let $m = n_2^2$, $\ell = n^d$, $z \in \{0.1\}^m$ and $S = \{s_1 < s_2 < \cdots < s_{n_2}\}$ be a subset of bit positions between 1 to m.

[4] All strings chosen at random in the above definitions, are chosen *uniformly* at random.
[5] By $\langle y, r \rangle$ we denote the inner product modulo 2.

In Section ..., we have *explicitely* constructed ℓ such that $|S_i \cap S_j| \leq \log_n \ell$, for every $i \neq j$, given $S_1, \ldots, S_\ell$.
We define $G_d^f : \{0,1\}^m \to \{0,1\}^\ell$ as:

$$G_d^f(z) = h(z|_{S_1})h(z|_{S_2})\ldots h(z|_{S_\ell})$$

It is clear that $G_d^{f_n} \leq_T^p h_{n_2} \leq_T^p g_{n_1} \leq_T^p f_n$, so we have that:

$$G_d^{f_n} \leq_T^p f_n$$

Using the above, we can show that a distinguisher for $G_d^{f_n}$ can be used along with an oracle for f_n to efficiently construct a polynomial-size circuit for f_n. So, we finally want to prove that $D^{G_d,1/5} \to^{f_n} C^f$.
In order to achieve this, we will prove the three following lemmata:

Lemma 5.6. $D^{G_d,1/5} \to^{f_n} C^{h,1/2+\mathcal{O}(1/\ell)}$

Proof: Firstly, note that we can use an oracle for h_{n_2}, because, as we saw, $h_{n_2} \leq_T^p g_{n_1} \leq_T^p f_n$. As in Section ..., we can construct a circuit to predict h:

- Pick $i \in \{1, \ldots, \ell\}$ uniformly at random.
- For each $j : 1 \leq j \leq \ell$, with $j \notin S_i$, pick z^j in $\{0,1\}$ uniformly at random.
- For each $i' < i$, query h at all $2^{|S_i \cap S_j|} \leq \ell$ strings that might be $z|_{S_{i'}}$ for a consistent with the z_j's, and store the answered queries in a table T.
- Pick $b_{i'} \in \{0,1\}$ uniformly at random for $i \leq i' \leq \ell$.
- Let $D \in D_m^{G,1/5}$, and C the following circuit:
  - On input x, set $z|_{S_i} = x$, while the other bits of z are chosen radomly.
  - Set $b_{i'} = h(z|_{S_{i'}})$, for $i' \leq i$, by looking up the appropriate entry in T.
  - *If* $D(b_1, \ldots, b_\ell = 1$, output b_i, *else* output $\neg b_i$
- By random sampling, using the oracle h_{n_2}, estimate $\mathbf{Pr}\left[C(x) = f(x)\right]$. *If* $\mathbf{Pr}\left[C(x) = f(x)\right] > \frac{1}{2} + \frac{1}{20\ell}$, output C, or *else* repeat.

In previous Section, we have shown that the expected probability of success for C is at least $\frac{1}{2}+\frac{1}{10\ell}$, so the number of repetitions before outputting a "good" C is at most $\mathcal{O}(n\ell)$ with high probability. □

We also give a lemma without proof:

Lemma 5.7 ([GL89]). $C^{h,1/2+\mathcal{O}(\ell^{-1})} \rightarrow C^{g,\mathcal{O}(\ell^{-3})}$

Lemma 5.8. $C^{g,\mathcal{O}(\ell^{-3})} \rightarrow^{f_n} C^{f,1-n^{-c}}$

Proof: Let $C \in C_{n_1}^{g,\alpha}$. We can construct a circuit C' as follows:

- Let $n_3 = n_1/n$.
- Repeat from $r = 1$ to $r = n^3/\alpha$:
  - Pick $i \in \{1, \dots, n_3\}$ uniformly at random.
  - For each $j \neq i$, pick $x_j \in \{0,1\}^n$ at random, query $f(x_j)$ and record the answer.
  - Flip coins until a "head" arises or until n "tails" have been flipped.
  - Let t_1 be the number of flips.

Let C'_r be the following three-valued ciruit:

- On input x, compute t: the number of bit positions $j \neq i$ where the j'th bit of $C(x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_{n_3})$ *disagrees* with $f(x_j)$.
- *If* $t < t_1$, output the i^{th} bit of $C(x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_{n_3})$, *else* output "*reject*".

Let C' be the circuit that outputs the majority answer from those C'_r that do not reject. We can prove that, for nonnegligible α, $C' \in C^{f,1-n^{-c}}$ with high probability.
Also, if α is at least inverse polynomial, this construction takes polynomial time. □

So, finally we can prove that given an efficient Distinguisher, we can efficiently construct a circuit computing f (with access to an oracle):

Lemma 5.9. *If f is random self-reducible, then:*

$$D^{G_f,1/5} \rightarrow^{f_n} C^f$$

Proof: Using Lemmata 5.6, 5.7, 5.8, and f's Random Self-Reducibility (Definition 5.5), we respectively have:

$$D^{G_d,1/5} \rightarrow^{f_n} C^{h,1/2+\mathcal{O}(1/\ell)} \rightarrow C^{g,\mathcal{O}(\ell^{-3})} \rightarrow^{f_n} C^{f,1-n^{-c}} \rightarrow C^{f}$$

□

Step 3: The Derandomization

We will now use the generators $G_d^{f_n}$ we constructed in the previous step to *derandomize* **BPP** algorithms in deterministic subexponential time 2^{k^δ}, for a (arbitrarily) given constant $\delta > 0$. The simulation is described by the following algorithm:

1. Let $\delta > 0$ be given.
2. Let x, with $|x| = k$ be the input to the **BPP** algorithm. The algorithm uses k^{c_1} random bits (polynomial in the size of the input, as defined) and time.
3. We set $d = \frac{2cc_1}{\delta}$, and $n = k^{\delta/2c}$.
4. Compute the range of G_n, that is a set of $n^d = \cdots = k^{c_1}$ strings. The time required for this is $\mathcal{O}\left(2^{n^c}\right) = \mathcal{O}\left(2^{k^\delta}\right)$.
5. We then simulate the **BPP** algorithm on each element, and take as output the majority output.

We'll show that if that simulation fails on *all* inputs for a given δ, then we can efficiently construct a distinguisher for $G_d^{f_n}$ using an oracle for f_n.

Lemma 5.10. *If this (heuristic) algorithm fails to be in* ***SUBEXP****, then we have an efficient distinguisher, that is,* $D^{G_d^f,1/5}$ *is efficiently constructible with oracle access to* f_n.

Proof: Assume that the above (deterministic) algorithm is incorrect, with probability $1/k^d$ with respect to some P-sampleable distribution μ_k, on k-bit strings[6], for all but finitely many k's.

Then, we can set $k = n^{2c/\delta}$ (n is given), and sample $r = k^{\mathcal{O}(1)}$ instances $x_1, \ldots, x_r$ according to μ_k. The necessary time for this is polynomial (by

[6] The probability distribution μ (or even the family of probability distributions $\{\mu_n | n \in \mathbb{N}\}$) is *polynomially sampleable* (P-sampleable) if there is a polynomial p and a polynomial-time computable function M, so that if r is a $p(n)$-bit string chosen *uniformly* at random, then $M(n, r)$ is distributed according to μ (or μ_n).

definition) in k, and so it is in n. The algorithm fails with high probability for one of these instances!

We can construct in (probabilistic) polynomial time a family of circuits D_i, which view their input as a random sequence, and simulate the **BPP** algorithm on x_i, using that random sequence.

With high probability, at least one D_i is a distinguisher for G_n. To find which is a "good" distinguisher, we can use our oracle: We can test all D_i's by using that $G_d^{f_n}$ can be evaluated with oracle access to f_n.

If we reduce the error probability of the **BPP** algorithm to $\frac{1}{10}$, then our distinguisher is in $D^{G_d^f, 1/5}$.

□

Step 4: Removing the Oracle

We proved (Lemma 5.10) that if the derandomization algorithm fails, we have a probabilistic polynomial-time algorithm such that, for every n, constructs a circuit for f_n by using it as oracle.

If we can remove the oracle need in the above algorithm, we can turn it into a **BPP** algorithm computing f:

Lemma 5.11. *If f is downward self-reducible, and C^f is efficiently constructible using oracle f_n, then $f \in$ **BPP**.*

Proof: We can construct all circuits $C_1 \in C_1^f, \ldots, C_n \in C_n^f$. If we have computed C_i, we can efficiently construct C_{i+1}^f, with oracle f_{i+1} and error $e = \frac{1}{n^2}$, by simulating queries to f_{i+1} by M^{C_i} (where M is the Turing Machine described in Definition 5.1).

Also, if T_1 denotes the time taken by the construction without the oracle queries, and T_2 the time taken to simulate queries not counting the time to evaluate oracle calls by M, we have that:

$$|C_{i+1}| \leq T_1 \cdot T_2$$

which is a fixed polynomial in n, indepedent of $|C_{i+1}|$.

So, we have each $|C_i|$ bounded by a polynomial, thus the time for each stage (including oracle calls) is also bounded polynomially. The probability that $C_n \notin C_n^f$ is at most $e \cdot n = \frac{1}{n}$, so the error is bounded and we have described a **BPP** algorithm computing f. □

Finally, we proved if our heuristic derandomization algorithm fails, we have an efficient distinguisher for G_d^f (by Lemma 5.10), and so we have a **BPP** algorithm computing f (Lemma 5.11). But f denotes the `PERMANENT` function, which we assumed in *Step 1* that is **EXP**-complete. So, every language in **EXP** can be decided by this **BPP** algorithm, hence **BPP** = **EXP**, which *contradicts* the hypothesis of our theorem. That completes the proof. □

5.3 Main Corollaries and Consequences

The main corollary is the following:

Corollary 5.12. *There are functions in* $\boldsymbol{EXP} \cap \mathbf{P}_{/\mathbf{poly}}$ *that* cannot *be simulated deterministically in time* $2^{o(n)}$ *with extra* $o(n)$ *advice, so that, for infinitely many* n*'s, the simulation to be correct on at least* $\frac{2}{3}$ *fraction of all inputs of size* n*.*

Corollary 5.13. *If every* ***BPP*** *language can be simulated deterministically in time* $2^{o(n)}$ *with extra* $o(n)$ *advice, so that, for infinitely many* n*'s, the simulation to be correct on at least* $\frac{2}{3}$ *fraction of all inputs of size* n*, then* $\boldsymbol{EXP} \neq \boldsymbol{BPP}$*.*

Combining the above two Propositions, we have the concluding:

Corollary 5.14. *If* $\boldsymbol{BPP} = \boldsymbol{EXP} \cap \mathbf{P}_{/\mathbf{poly}}$ *, then* $\boldsymbol{BPP} = \boldsymbol{EXP}$*.*

We can also give an alternative formulation of Theorem 5.1:

Theorem 5.15. *If* $\boldsymbol{EXP} \neq \boldsymbol{BPP}$*, then, for every* $\epsilon > 0$*, there is a quick generator* $G : \{0,1\}^{n^\epsilon} \to \{0,1\}^n$ *that is pseudorandom with respect to any P-sapleable family of* n*-size Boolean Circuits infinitely often.*

We can give an analysis of the above proposition:
Let $B_G(n)$ be the set of all Boolean Circuits C of size n that are "bad" for the generator G, that is:

$$C \in B_G(n) \Leftrightarrow |Pr_x\left[C(x) = 1\right] - Pr_y\left[C(G(y)) = 1\right]| \geq \frac{1}{n}$$

Now let R be any probabilistic polynomial-time algorithm that, on input 1^n, outputs a Boolean Circuit of size n. Then, there are infinitely many n's such that:

$$Pr\left[R(1^n) \in B_G(n)\right] < \frac{1}{n}$$

where the probability is taken over the internal coin tosses of R.

The proof of Theorem 5.1 uses, as we saw, Meyer's Theorem ($\mathbf{EXP} \subset \mathbf{P}_{/\mathbf{poly}} \Rightarrow \mathbf{EXP} = \Sigma_2^p$), which is a *non-relativizing result.* We do not know if Theorem 5.1 relativizes, unlike the precedent (non-uniform) hardness-randomness tradeoffs (we mentioned in Chapter 3) which relativize.

5.4 Simiral Results

In [TV07], Luca Trevisan and Salil Vadhan gave generalization of Theorem 5.1, providing a continuous trade-off between hardness and randomness, which we state here without proof:

Theorem 5.16. *If* $\mathbf{EXP} \nsubseteq \bigcup_{c\in\mathbb{N}} \mathbf{BPTIME}(t(t(n^c)))$ *for a time-constrictible function* f*, then:*

$$\mathbf{BPP} \subseteq \bigcup_{c\in\mathbb{N}} \mathbf{DTIME}(n^c \cdot 2^{t^{-1}(n)})$$

and the simulation is correct for at least $1 - \frac{1}{n^c}$ *fraction of inputs of size* n*.*

Chapter 6

Uniform Derandomization of RP

Valentine Kabanets obtained in [Kab00] another derandomizing result, for **RP** this time, under the -weaker- assumption that $\mathbf{EXP} \neq \mathbf{ZPP}$. The simulation he obtained is based also on a *easiness* assumption: If there is an efficient algorithm constructing the inputs on which the generator fails, we can use it to construct another algorithm which can efficiently guess a Boolean function that is sufficiently hard for the hardness-based generators we saw in the previous chapters.

First, we present an elegant formalization of languages that cannot be efficiently distinguished, the parameters of this setting, e.g. the computational abilities of an adversary that tries to separate them, and some of its properties.

6.1 Formalizing Computational Indistinguishability

In Chapter 1 we exposed different views for randomness, used by several theories. In our perspective, we consider two subjects as *equal*, if we cannot separate (or distinguish) them by any efficient procedure.

In order to develop a formalization for this kind of equality, we introduce the notion of *refuters*, which are deterministic Turing Machines that try to separate a language from another.

6.1.1 Deterministic Refuters

Definition 6.1. *A refuter is a (length-preserving) Turing Machine R, such that $R(1^n) \in \{0,1\}^n$. Refuters can be deterministic, non-deterministic, or probabilistic. In the case of non-determinism, refuter's each nondeterministic branch, on input 1^n, either produces a string in $\{0,1\}^n$, or is marked* *with reject.*

Intuitively, a refuter is an adversasy, who given specific computational power, tries to distinguish a language (or a Boolean function) from another. If it fails, we consider the two languages as equal, i.e. no adversary with these computational powers can find a string that is in the one language and isn't in the other (a string in the symmetric difference of the two languages). We can formalize the above idea as follows:

Definition 6.2. *Let $t(n)$ be a time bound. Two languages $L, M \subseteq \{0,1\}^*$ are $t(n)$-*indistinguishable*, denoted as $L \overset{t(n)}{=} M$, if for every deterministic $t(n)$-time refuter R we have $R(1^n) \notin L \triangle M$ for all but finitely many n's, where $\triangle$ denotes the symmetric difference of the two sets.*

So, we can write $L \overset{\mathbf{P}}{=} M$ if $L \overset{p(n)}{=} M$ for every polynomial $p \in poly(n)$, that is if L and M cannot be distinguished by any polynomial-time refuter. Similarly, we write $L \overset{\mathbf{EXP}}{=} M$ for indistinguishability with respect to an exponential-time refuter.

Using this equality

Definition 6.3. *For a complexity class $\mathcal{C}$ of languages over $\{0,1\}$, we can define the complexity class:*

$$\texttt{pseudo}_P\mathcal{C} = \{L \subseteq \{0,1\}^* | \exists M \in \mathcal{C} \text{ such that } L \overset{\boldsymbol{P}}{=} M\}$$

- The refuters above are required to fail almost everywhere at producing a certain string ($\in L \triangle M$).
  This requirement can be relaxed in **i.o. complexity** setting.

-

Theorem 6.1. *For any complexity class $\mathcal{C} \subseteq \boldsymbol{EXP}$, we have that:*

$$\mathcal{C} = \mathbf{EXP} \cap \texttt{pseudo}_{EXP}\mathcal{C}$$

Proof: Since $\mathcal{C} \subseteq \mathbf{EXP}$, and obviously $\mathcal{C} \subseteq \texttt{pseudo}_{EXP}\mathcal{C}$, we have that $\mathcal{C} \subseteq \mathbf{EXP} \cap \texttt{pseudo}_{EXP}\mathcal{C}$. In order to prove the opposite inclusion, let $L \in \mathbf{EXP} \cap \texttt{pseudo}_{EXP}\mathcal{C}$ such that $L \overset{\mathbf{EXP}}{=} M$ for some language $M \in \mathcal{C}$ (that is, L and M cannot be distinguished by every refuter running in exponential time). So, consider an exponential-time refuter R:

> On input 1^n, R goes though all n-bit strings checking if any of them is in $L \triangle M$, and outputs the lexicographically first string in $L \triangle M$ if such a string exists, or 0^n otherwise.

The above checking can be done easily in exponential time, since $L, M \in$ **EXP**. Now, suppose that L and M differ for infinitely many input lengths. Then, R will succeed infinitely often to provide at least one such string, so $L \overset{\mathbf{EXP}}{\neq} M$, contradicting our hypothesis. Hence, L and M must coincide for all but finitely input lengths, so $L \in \mathcal{C}$ almost everywhere. $\square$

We can prove a "*Time Hierarchy Theorem*" for the `pseudo` setting.

Theorem 6.2. *Let $t_2(n)$ be a constructible function, and let $t_1(n)\log t_1(n) \in o(t_2(n))$. Then, for infinitely many input sizes:*

$$\mathbf{DTIME}(t_2(n)) \nsubseteq \texttt{pseudo}\mathbf{DTIME}(t_1(n))$$

Using refuters, we can express **BPP** Derandomization Theorem as follows:

Theorem 6.3 (Theorem 5.1 restated)**.** *If **BPP** $\neq$ **EXP**, then, for infinitely many input sizes:*

$$\boldsymbol{BPP} \subseteq \texttt{pseudo}_{BPP}\boldsymbol{SUBEXP}$$

6.1.2 Probabilistic Refuters

We can "enhance" refuters' power by allowing them to use randomness. We can have two "versions" of such refuters: bounded-error probabilistic (corresponding to class **BPP**), and zero-error probabilistic (corresponding to class **ZPP**):

Definition 6.4 (Bounded-error probabilistic refuters)**.** *Let $t(n)$ be a time bound. Two languages $L, M \subseteq \{0,1\}^*$ are* bounded-error *probabilistically* $t(n)$-indistinguishable*, denoted as $L \overset{BP-t(n)}{=} M$, if for every probabilistic $t(n)$-time refuter R we have:*

$$\boldsymbol{Pr}[R(1^n) \notin L \triangle M] \geq 1 - n^{-c}$$

for every $c \in \mathbb{N}$, and for all but finitely many n's.

Similarly:

Definition 6.5 (Zero-error probabilistic refuters)**.** *Let $t(n)$ be a time bound. Two languages $L, M \subseteq \{0,1\}^*$ are* zero-error *probabilistically $t(n)$-indistinguishable, denoted as $L \overset{ZP-t(n)}{=} M$, if for every probabilistic refuter R which halts within time $t(n)$ with probability at least n^{-c}, for some $c \in \mathbb{N}$ and for all by finitely many n's we have: $R(1^n) \notin L \triangle M$ for at least one legal computation of R on input 1^n which halts in time $t(n)$.*

6.2 Main Results

The easiness assumtpion Kabanets used is similar to Natural Proofs we saw in section **??**. This conjecture, that there is no n^c-useful natural predicate $\mathcal{P}$ for some $c \in \mathbb{N}$, can be expressed it the terms of Boolean Circuits, as follows:
"For every $\{C_n\}_{n\in\mathbb{N}} \subseteq \mathbf{P}_{/\mathbf{poly}}$ such that almost every C_n accepts at least a polynomial fraction of all n-bit inputs, there exists a $d \in \mathbb{N}$ such that almost every C_n accepts the n-bit prefix of the truth table of a $\lceil \log n \rceil$-variable Boolean function with hardness at most $\lceil \log n \rceil^d$."

We can use the above by replacing the random strings with the truth tables of easy Boolean functions, and accepts if at least one of them works. If this simulation fails, we can obtain a natural predicate $\mathcal{P}$ (recall Definition **??**) which can used as a hardness test.

The main result is the following, implying that either **BPP** "collapses" to **ZPP**, either every **RP** algorithm can be simulated deterministicallly in subexponential time:

Theorem 6.4. *At least one of the following holds:*

1. $\mathbf{ZPP} = \mathbf{BPP}$
2. $\mathbf{RP} \subseteq \mathtt{pseudo}_{ZPP}\mathbf{SUBEXP}$ *infinitely often.*

Proof: Let S_m^δ, for $m \in \mathbb{N}$ and $\delta > 0$, be the set of *truth tables* of all $\lceil \log m \rceil$-variable Boolean functions of hardness at most m^δ. Also, let $A \in \mathbf{RP}$ that on input x, $|x| = n$, uses atr most $m = n^\alpha$ random bits. Consider the *deterministic* algorithm B_A^ϵ, for an arbitrary $\epsilon > 0$, which for inputs x, $|x| = n$, accepts x iff $A(x)$ accepts for at least one $\alpha \in S_m^{\epsilon'}$ used as random string, where $\epsilon' = \frac{\epsilon}{2\alpha}$. The running time of B_A^ϵ is at most 2^{n^ϵ} (`why?`).

If, for every $A \in \mathbf{RP}$ and every $\epsilon > 0$ it holds that $L(A) \overset{ZPP}{=} L(B_A^\epsilon)$, then $\mathbf{RP} \subseteq \mathtt{pseudo}_{ZPP}\mathbf{SUBEXP}$, and the proof is complete.

Otherwise, there exists an $\hat{A} \in \mathbf{RP}$, a constant $\hat{\epsilon} > 0$ and a probabilistic polynomial-time refuter R, such that, for $L_1 = L(\hat{A})$ and $L_2 = (B_{\hat{A}}^{\hat{\epsilon}})$, we have that $R(1^n) \in L_1 \triangle L_2 \Rightarrow R(1^n) \in L_1 \backslash L_2$ (since $L_2 \subseteq L_1$) for *almost every* n, since $R(1^n)$ halts.

Now if $R(1^n)$ halts, then $\hat{A}(R(1^n))$ can be viewed as a Boolean circuit C^{hard}, that accepts a significant fraction of all m-bit strings, and every accepted string consists the *truth table* of a $\lceil \log m \rceil$-variable Boolean function $f_{\lceil \log m \rceil}$, with size[1] $\mathbf{SIZE}\left(f_{\lceil \log m \rceil}\right) > m^{\epsilon'}$, where, as before, $\epsilon' = \frac{\hat{\epsilon}}{2\alpha}$.
Since $R(1^n)$ halts with significant probability and always outputs a string in $L_1 \backslash L_2$, we have a *zero-error* probabilistic algorithm for constructing such circuits C^{hard}, that run in expected polynomial time.

[1] Recall Definition 3.3

We will now show that we can construct a pseudorandom generator G, with stretch function $S(k) = 2^{k/d}$ and hardness $H(G) > k$ for some $d \in \mathbb{N}$, in zero-error probabilistic polynomial time $poly(k)$. For given ϵ', concider c and d as in Theorem 5.1 (Recall the generator's construction in Section 5.2.2), and let $m = n^{\alpha} = k^{c}$. Consider the algorithm that first constructs a testing circuit C^{hard}, then chooses a string $\beta \in \{0,1\}^m$ uniformly at random, which is accepted by C^{hard}, and it uses β to construct a generator G_β, as described above. It follows that, for all sufficiently large k's, G_β has hardness greater than k.

The first two stages of the above algorithm can be executed by a **ZPP** algorithm. The third stage can be done in deterministic polynomial time. So, for every $L \in$ **BPP**, we have that $L \in$ **ZPP**, so **BPP** $\subseteq$ **ZPP**, and since the other inclusion is trivial, we have that **BPP** = **ZPP** . □

Now, using Theorem 6.4 we can prove a similar to **BPP** Derandomization Theorem (Theorem 5.1):

Theorem 6.5. *If* ***ZPP*** $\neq$ ***EXP****, then, for infinitely many input sizes:*

$$\boldsymbol{RP} \subseteq \texttt{pseudo}_{ZPP}\boldsymbol{SUBEXP}$$

Proof: Suppose, for the sake of contradiction, that **RP** $\nsubseteq$ $\texttt{pseudo}_{ZPP}$**SUBEXP** infinitely often. Then, Theorem 6.4 implies that **ZPP** = **BPP**, and so **BPP** $\nsubseteq$ $\texttt{pseudo}_{BPP}$**DTIME** $\left(2^{n^{\varepsilon}}\right)$, for some $\varepsilon > 0$. Hence, by Theorem 5.1 (or 6.3), we have that **BPP** = **EXP** $\Rightarrow$ **ZPP** = **EXP**, which contradicts our hypothesis. □

We end this section by stating Theorem 6.5 in a "gap" theorem form, exactly like the Impagliazzo-Wigderson result in the previous chapter. So, either no derandomization of **ZPP** is possible, or else **RP** has a non-trivial deterministic simulation:

Theorem 6.6. *Either:*

1. ***ZPP*** = ***EXP***
2. ***RP*** $\subseteq$ $\texttt{pseudo}_{ZPP}$***SUBEXP*** *infinitely often.*

Chapter 7

Uniform Derandomization of AM

7.1 Nondeterministic Derandomization

As we mentioned in Chapter 1, the class **AM** is, by definition, the probabilistic analogue of **NP** (usually denoted as **AM** = **BP**· **NP**). Also, the next enhances our intuition that **BPP** collapsing to **P** is analogue to **AM** collapsing to **NP**:
For a complexity class $\mathcal{C}$, we define the class:

$$almost\mathcal{C} = \left\{ L | \mathbf{Pr}\left[L \in \mathcal{C}^A \right] = 1 \right\}$$

where the probability is taken all over random choices of oracle A. Bennett and Gill in their classical paper [BG81], proved that:

Theorem 7.1. $almost\mathbf{P} = \mathbf{BPP}$

A few years later, Nisan and Wigderson in [NW94], using the NW-generator we presented in Section 4.4, proved that also:

Theorem 7.2. $almost\mathbf{NP} = \mathbf{AM}$

Also, we have the surprising fact that random oracles do *not* help Polynomial Hierarchy:

Theorem 7.3. $almost\mathbf{PH} = \mathbf{PH}$

7.2 Main Results

Theorem 7.4. *At least one of the following holds:*

1. **AM** = **NP**

2. $\mathbf{NP} \subseteq \mathtt{pseudo}_{NP}\mathbf{SUBEXP}$ *infinitely often.*

Proof: We will try to simulate **NP** by using "easy" functions as potential witnesses. If this succeeds, we could have an efficient simulation of **NP**. Otherwise, we will have a resource of hard functions which we'll use to construct pseudorandom sequences.

Let A be a language in **NP**, but *not* in $\mathtt{pseudo}_{NP}\mathbf{DTIME}\left(2^{n^{\varepsilon}}\right)$, for some $\varepsilon \in (0,1)$. Since $A \in \mathbf{NP}$, there exists by definition a polynomial-time computable relation M, and a polynomial $m = poly(n)$, such that $\forall x \in \{0,1\}^n$: $x \in A$ if and only if there exists a $y \in \{0,1\}^m$ such that $M(x,y) = 1$.

We also denote by S_m^{δ} the set of truth tables of all $\lceil \log m \rceil$-Boolean functions of hardness at most m^{δ}, using a $\mathtt{SAT}$ oracle, that is, in $\mathbf{SIZE}^{\mathtt{SAT}}\left(m^{\delta}\right)$ (by using Chapter 2 notation). Note that S_m^{δ} contains at most $2^{m^{2\delta}}$ truth tables (the maximum number of possible circuits of this hardness).

Now, consider the *deterministic* procedure D_M^{δ}, which, for $x \in \{0,1\}^n$, accepts x if and only if there exists a truth table $y \in S_m^{\delta}$ such that $M(x,y) = 1$. A $\mathtt{SAT}$ gate in a circuit of size m^{δ} can be evaluated in deterministic time $2^{\mathcal{O}(m^{\delta})}$, each truth table in S_m^{δ} can be generated in this time. So, $D_M^{\delta} \in \mathbf{DTIME}\left(2^{m^{c\delta}}\right)$, for some $c \in \mathbb{N}$, and by choosing the constant δ so that $m^{c\delta} \leq n^{\varepsilon}$, we also have that $D_M^{\delta} \in \mathbf{DTIME}\left(2^{n^{\varepsilon}}\right)$.

Also, from our assumption that $A \notin \mathtt{pseudo}_{NP}\mathbf{DTIME}\left(2^{n^{\varepsilon}}\right)$, for some $\varepsilon > 0$, there is a (nondeterministic) polynomial-time refuter R, such that for *almost every* n, every string produced in a branch of $R(1^n)$ will be "misclassified" by D_M^{δ}: A string is misclassified only when $M(x,y) = 0 \forall y \in S_m^{\delta}$ but $M(x,y) = 1$ for some $y \in \{0,1\}^m \backslash S_m^{\delta}$.

Let $\ell = \lceil \log m \rceil$. We have now a nondeterministic polynomial time procedure for producing the truth table of an ℓ-variable Boolean function which is *not* in $\mathbf{SIZE}^{\mathtt{SAT}}\left(2^{\delta\ell}\right)$, for almost every ℓ:

- Use R to (nondeterministically) produce a misclassified input x
- Guess y of length 2^{ℓ} and produce it if $M(x,y) = 1$, for a misclassified x.

As we mentioned in the beginning, this "misclassification" provided us with a method to find a *hard* function. Using the following Lemma, (which we present without proof, the reader is referred to [KvM99]), we have a method to construct a pseudorandom generator in time $2^{\mathcal{O}(\ell)}$ producing pseudorandom sequences that look random to every circuit in $\mathbf{SIZE}^{\mathtt{SAT}}\left(2^{\delta\ell}\right)$.

Lemma 7.5 (from [KvM99])**.** *Let A be any language, and suppose that f is a Boolean function of size $\mathbf{SIZE}^{A}\left(f_{\ell}\right) = 2^{\Omega(\ell)}$. Then, there is a procedure running in deterministic time $2^{\mathcal{O}(\ell)}$ that transforms the truth table of f_{ℓ} into a pseudorandom sequence that looks random to all circuits in $\mathbf{SIZE}^{A}\left(2^{\Omega(\ell)}\right)$.*

We still have to connect somehow this results with Arthur-Merlin games. The following Lemma gives us a general derandomization result for the class **AM**:

Lemma 7.6. *If a pseudorandom sequence that looks random to circuits in* $\boldsymbol{SIZE}^{\mathit{SAT}}(n)$ *can be produced in nondeterministic time* $t(n)$*, then:*

$$\boldsymbol{AM} \subseteq \boldsymbol{NTIME}(poly\,(t(poly(n))))$$

Proof (of Lemma 7.6): Let L be a language in **AM**. Then, there exists by definition a polynomial-time computable relation M, and a polynomial $m = poly(n)$ such that, for every $x \in \{0,1\}^n$:

$$x \in L \Rightarrow \mathbf{Pr}_{y \in \{0,1\}^m}\,[\exists z \in \{0,1\}^m \ s.t.\ M(x,y,z) = 1] \geq \frac{3}{4}$$

$$x \notin L \Rightarrow \mathbf{Pr}_{y \in \{0,1\}^m}\,[\exists z \in \{0,1\}^m \ s.t.\ M(x,y,z) = 1] < \frac{1}{4}$$

For any fixed x, the *predicate* "$\exists z \in \{0,1\}^m : M(x,y,z) = 1$" on y is in $\mathbf{SIZE}^{\mathtt{SAT}}(m^c)$ for some constant $c \in \mathbb{N}$. We use the nondeterministic procedure, running in time $t(m^c) = t(n^{c'}) = t(poly(n))$, to produce a pseudorandom set $G = \{g_1, g_2, \ldots, g_{|G|}\}$ that looks random to all circuits in $\mathbf{SIZE}^{\mathtt{SAT}}(m^c)$. Then:

$$x \in L \Leftrightarrow \mathbf{Pr}_{y \in G}\,[\exists z \in \{0,1\}^m \ s.t.\ M(x,y,z) = 1] \geq \frac{1}{2}$$

To decide L, we (nondeterministically) guess strings $z_1, z_2, \ldots, z_{|G|}$ from $\{0,1\}^m$, and accept x if and only if $M(x, g_i, z_i) = 1$ for most i (the majority vote). As $|G| \leq t(poly(n))$, this procedure runs in nondeterministic time $poly(t(poly(n)))$. □

So, since our pseudorandom sequence can be produced in nondeterministic polynomial time, Lemma 7.6, for $t(n) = poly(n)$, implies that **AM** = **NP**.

□

It it worth to notice that since the Graph Nonisomorphism Problem (`GNI`) belongs to **AM** and co**NP**[1] (and thus in their intersection), the above Theorem imply that either `GNI` is in **NP**, or that can be simulated in nondeterministic subexponential time, so that the simulation appears correct with respect to any *nondeterministic* polynomial-time refuter, for infinitely many n's.

[1] Recall the discussion in page 17, and Theorem 2.7

Theorem 7.7. *$co\mathbf{NP} \cap \mathbf{AM} \subseteq \bigcap_{\varepsilon>0} \mathtt{pseudo}_{NP}\mathbf{NTIME}\left(2^{n^{\varepsilon}}\right)$ infinitely often.*

Proof: Since $\mathtt{pseudo}_{NP}\mathbf{SUBSEXP}$ is *closed* under complement, Theorem 7.4 implies that, for infinitely many input sizes, either:

- $co\mathbf{NP} \subseteq \mathtt{pseudo}_{NP}\mathbf{SUBSEXP} \subseteq \bigcap_{\varepsilon>0} \mathtt{pseudo}_{NP}\mathbf{NTIME}\left(2^{n^{\varepsilon}}\right)$, or
- $\mathbf{AM} = \mathbf{NP} \subseteq \bigcap_{\varepsilon>0} \mathtt{pseudo}_{NP}\mathbf{NTIME}\left(2^{n^{\varepsilon}}\right)$ (trivially)

Hence, we have that $co\mathbf{NP} \cap \mathbf{AM} \subseteq \bigcap_{\varepsilon>0} \mathtt{pseudo}_{NP}\mathbf{NTIME}\left(2^{n^{\varepsilon}}\right)$

□

So, we obtain the following remarkable conclusion, the first non-trivial derandomization result for `GNI`, stating that this problem has subexponential-size proofs infinitely often, without *any assumption*:

Corollary 7.8. *$\mathtt{GNI} \in \bigcap_{\varepsilon>0} \mathtt{pseudo}_{NP}\mathbf{NTIME}\left(2^{n^{\varepsilon}}\right)$, for infinitely many input sizes.*

We can also have a more general result for these tradeoffs:

Theorem 7.9. *Either:*

- $\mathbf{NP} \subseteq \mathtt{pseudo}_{NP}\mathbf{DTIME}(t(n))$, *or*
- $\mathbf{AM} \subseteq \mathbf{NTIME}\left(\exp\left(\log t^{-1}(\exp n)\right)\right)$

for any $t(n) = \Omega(n)$.

Proof:
but we'll use, instead of Lemma 7.5, a more general result, presented also in [KvM99]:

Lemma 7.10 (from [KvM99]). *Let A be any language, and suppose f is a Boolean function with size $\mathbf{SIZE}^A(f_\ell) \geq m(\ell)$. There is a procedure running in deterministic time $2^{\mathcal{O}(\ell)}$ that transforms the truth table of f_ℓ into a pseudorandom sequence that looks random to all circuits in $\mathbf{SIZE}^A(m^{\varepsilon}(\ell^{\varepsilon}))$ for some constant $\varepsilon > 0$.*

□

Using the above theorem, by setting $t(n) = 2^{\log^{o(1)} n}$, we obtain the following (better than Corollary(7.8)) simulation:

Corollary 7.11. *$\mathtt{GNI} \in \mathtt{pseudo}_{NP}\mathbf{NTIME}\left(2^{2^{\log^{o(1)} n}}\right)$, for infinitely many input sizes.*

The techniques we developed above, can be used in the same way to provide also space-time trade-offs between complexity classes. We expose some results from [Lu00]:

Theorem 7.12. *Either:*

1. $\mathbf{DTIME}(t(n)) \subseteq \bigcap_{\varepsilon>0} \boldsymbol{DSPACE}(t^{\varepsilon}(n))$ *infinitely often for any function* $t(n) = 2^{\Omega(n)}$*, or*
2. $\mathbf{P} = \mathbf{BPP}$ *and* $\mathbf{AM} = \boldsymbol{NP}$ *and* $\boldsymbol{PH} \subseteq \oplus\boldsymbol{P}$

Theorem 7.13. *Either:*

1. $\boldsymbol{DTIME}(t(n)) \subseteq \bigcap_{\varepsilon>0} \boldsymbol{DSPACE}\left(2^{\log^{\varepsilon} t(n)}\right)$ *infinitely often for any function* $t(n) = 2^{\Omega(n)}$*, or*
2. $\boldsymbol{BPP} \subseteq \boldsymbol{QuasiP}$ *and* $\boldsymbol{AM} \subseteq \boldsymbol{NQuasiP}$ *and* $\boldsymbol{PH} \subseteq \oplus\boldsymbol{QuasiP}$

Theorem 7.14. *Either:*

1. $\boldsymbol{DTIME}(t(n)) \subseteq \boldsymbol{DSPACE}(poly(\log t(n)))$ *infinitely often for any function* $t(n) = 2^{\Omega(n)}$*, or*
2. $\boldsymbol{BPP} \subseteq \boldsymbol{SUBEXP}$ *and* $\boldsymbol{AM} \subseteq \boldsymbol{NSUBEXP}$ *and* $\boldsymbol{PH} \subseteq \oplus\boldsymbol{SUBEXP}$

7.3 Gap Theorems for Arthur-Merlin Games

7.3.1 The High-End

The following theorem consists a non-deterministic analogue of Impagliazzo and Wigderson Theorem for **AM**. While the IW-theorem works in the low-end setting, this works in the high-end.

Theorem 7.15. *If* $\mathbf{E} \nsubseteq \mathbf{AM} - \mathbf{TIME}(2^{\epsilon n})$*, for some* $\epsilon > 0$*, then for all* $c > 0$*, and infinitely many input sizes, we have:*

$$\boldsymbol{AM} \subseteq \mathtt{pseudo}_{NTIME(n^c)}\boldsymbol{NP}$$

The above theorem can be stated also as a *gap* theorem for **AM**: Either Arthur-Merlin protocols are very strong and everything in **E** can be proved to a subexponential-time verifier, or they are very weak and Merlin can prove nothing that cannot be proven in the pseudo setting by standard **NP**-proofs.

We also have a similar gap-theorem for $\mathbf{AM} \cap co\mathbf{AM}$:

Theorem 7.16. *If* $\mathbf{E} \nsubseteq \mathbf{AM} - \mathbf{TIME}(2^{\epsilon n})$*, for some* $\epsilon > 0$*, then, for infinitely many input sizes:*

$$\mathbf{AM} \cap co\mathbf{AM} \subseteq \mathbf{NP} \cap co\mathbf{NP}$$

The class $\mathbf{AM} \cap co\mathbf{AM}$ has a very special interest since it contains the class **SZK** (Statistical Zero-Knowledge), and therefore it contains some very natural problems that are not known to be in **NP**, e.g. `GNI`, approximation of shortest and closest vector in a lattice, Statistical Difference etc.

It is a significant result that for the class $\mathbf{AM} \cap co\mathbf{AM}$, we can (*non-trivially of course*) get rid of "*infinitely often*" setting, and go up to "*almost everywhere*" complexity:

Theorem 7.17. *If* $\mathbf{E} \nsubseteq \mathbf{AM} - \mathbf{TIME}(2^{\epsilon n})$ *infinitely often, for some* $\epsilon > 0$, *then, for all but finitely many input sizes:*

$$\mathbf{AM} \cap co\mathbf{AM} \subseteq \mathbf{NP} \cap co\mathbf{NP}$$

We state another gap theorem concerns nondeterministic exponential time. In the non-uniform results, when moving from **BPP** to **AM** we can allow the hard function to be in $\mathbf{NE} \cap co\mathbf{NE}$, because this affects only the complexity of the generator: instead of being computable in deterministic polynomial time, it in now computable in $\mathbf{NP} \cap co\mathbf{NP}$. Since the application of the generator to derandomize **AM** already uses nondeterminism, this still gives the same reslt.

Theorem 7.18. *If* $\mathbf{NE} \cap co\mathbf{NE} \nsubseteq \mathbf{AM} - \mathbf{TIME}(2^{\delta n})$, *for some* $\delta > 0$, *then, for infinitely many n's, and for every* $c, \varepsilon > 0$*:*

$$\mathbf{AM} \subseteq \mathtt{pseudo}_{NTIME(n^c)}\mathbf{NTIME}\left(2^{n^{\varepsilon}}\right)$$

And also, for every $\varepsilon > 0$*:*

$$\mathbf{AM} \cap co\mathbf{AM} \subseteq \mathbf{NTIME}\left(2^{n^{\varepsilon}}\right) \cap co\mathbf{NTIME}\left(2^{n^{\varepsilon}}\right)$$

The above result states that either randomness is helpful and every proof that requires exponentially long witnesses can be replaced by a much more efficient Arthur-Merlin Game, or else, rendomness is relatively weak and every (polynomial-time) Arthur-Merlin Game can be replaced by a proof that does not use randomness while paying at most a subexponential cost in efficiency.

7.3.2 The Low-End Extension

The above gap theorems for AM and $\mathbf{AM} \cap co\mathbf{AM}$ in the above section are "High-End" results (recall the "**AM-TIME**$(2^{\epsilon n})$" condition). Its proof was based in a"resiliency" property of a Hitting-Set Generator construction, which works only in the High-End.

Using a variant of the aforementioned techniques, since the above dont't work when use time bound for **AM** smaller than $2^{\sqrt{n}}$, we can obtain and a "Low-End" result, presented in [SU07a].

Theorem 7.19. *There exists a language A complete for* **E** *(resp.* **EXP***), such that for every time-constructible function t: $m < t(m) < 2^m$, either:*

1. *A has an Arthur-Merlin protocol running in time $t(m)$*

2. *for any language $L \in$* **AM** *there is a nondeterministic machine M that runs in time $2^{\mathcal{O}(m)}$ (resp. $2^{m^{\mathcal{O}(1)}}$) on inputs of length:*

$$n = t(m)^{\Theta(1/(\log m - \log\log t(m))^2)}$$

(resp. $n = t(m)^{\Theta(1/(\log m)^2)}$) such that for any refuter R running in time $t(m)$ when producing strings of length n, there are infinitely many n's on which L and $L(M)$ are $t(m)$-indistinguishable.

In other words, either **E** is computable by an Arthur-Merlin protocol in time $s(\ell)$, or for every **AM** language L there exists a nondeterministic TM M that runs in time exponential in ℓ and solves L correctly on feasibly generated inputs of length $n = t(m)^{\Theta(1/(\log m - \log\log t(m))^2)}$. This is an extension of the gap theorems of the previous section.

The analogue for $\mathbf{AM} \cap co\mathbf{AM}$ follows:

Theorem 7.20. *There exists a language A complete for **E** (resp. **EXP**), such that for every time-constructible function t: $m < t(m) < 2^m$, either:*

1. *A has an Arthur-Merlin protocol running in time $t(m)$*

2. *for any language $L \in \mathbf{AM} \cap co\mathbf{AM}$ there is a nondeterministic machine M that runs in time $2^{\mathcal{O}(m)}$ (resp. $2^{m^{\mathcal{O}(1)}}$) on inputs of length:*

$$n = t(m)^{\Theta(1/(\log m - \log\log t(m))^2)}$$

(resp. $n = t(m)^{\Theta(1/(\log m)^2)}$) such that for any refuter R running in time $t(m)$ when producing strings of length n, there are infinitely many n's on which L and $L(M)$ are $t(m)$-indistinguishable.

Either **E** is computable by Arthur-Merlin protocols within time $s(\ell)$, or for any $\mathbf{AM} \cap co\mathbf{AM}$ language L there exists a non-deterministic (and co-nondeterministic) machine M that runs in time exponential in ℓ and solves L correctly on all inputs of length $n = t(m)^{\Theta(1/(\log m - \log\log t(m))^2)}$.

The non-standard implicit way of measuring the running time of M exists because it is not possivle to express the running time of M as a function of its input length in a closed form that covers all possible choices of $s(\ell)$.

And, like the theorem of the previous section, we can extract the "infinitely often" setting and have a more general result:

Theorem 7.21. *There exists a language A complete for **E** (resp. **EXP**), such that for every time-constructible function t: $m < t(m) < 2^m$, either:*

1. *A has an Arthur-Merlin protocol running in time $t(m)$ which agrees with L on infinitely many inputs (on other inputs the Arthur-Merlin protocol does not necessarily have a non-negligible gap between completeness and soundness), or*

2. *for any language $L \in \mathbf{AM} \cap co\mathbf{AM}$ there is a nondeterministic machine M that runs in time $2^{\mathcal{O}(m)}$ (resp. $2^{m^{\mathcal{O}(1)}}$) on inputs of length:*

$$n = t(m)^{\Theta(1/(\log m - \log\log t(m))^2)}$$

(resp. $n = t(m)^{\Theta(1/(\log m)^2)}$) such that $L = L(M)$.

Chapter 8

Derandomization vs Lower Bounds

As we saw in Chapters 4 and 5, the following three basic questions in Complexity Theory were proven to be equivalent in the *nonuniform* setting:

1. Existence of worst-case complexity problems in **E**.
2. Existence of worst-case complexity problems in **E**.
3. The existence of pseudorandom generators providing subexponential or even polynomial-time simulations of **BPP**.

8.1 Derandomization vs Circuit Lower Bounds

Although certain ciruit lower bounds imply Derandomization, they have been proven, as we saw, very tricky enough to prove, so we have to make asssumptions and conjectures for derandomization without them.

However, Impagliazzo, Kabanets and Wigderson proved in 2001 that derandomizing **MA** would imply lower bounds for **NEXP**,and, conversely, that it is impossible to separate **NEXP** and **MA** without proving that $\mathbf{NEXP} \not\subseteq \mathbf{P_{/poly}}$. We formalize the previous conclusions as follows:

Theorem 8.1.

$$\boldsymbol{NEXP} \subseteq \boldsymbol{P_{/poly}} \Rightarrow \boldsymbol{NEXP} = \boldsymbol{EXP} = \boldsymbol{MA}$$

Firstly, we show a "weaker" theorem (which captures only deterministic exponential time), proved by Babai, Fortnow, Nisan and Wigderson in 1993:

Theorem 8.2.

$$\boldsymbol{EXP} \subseteq \boldsymbol{P_{/poly}} \Rightarrow \boldsymbol{EXP} = \boldsymbol{MA}$$

Proof: Suppose that $\mathbf{EXP} \subseteq \mathbf{P_{/poly}}$. Then, by Meyer's Theorem (Theorem 3.4):

$$\Sigma_2^p = \mathbf{PSPACE} = \mathbf{IP} = \mathbf{EXP} \subseteq \mathbf{P_{/poly}}$$

So, every $L \in \mathbf{EXP}$ has an Interactive Proof, and since we have assumed that $\mathbf{EXP} \subseteq \mathbf{P_{/poly}}$, the Prover can be replaced by a polynomial circuit family $\{C_n\}$. We can now describe an (one-round) interactive proof between Prover Merlin and Verifier Arthur (i.e. the definitive condition of the class **MA**):

- Given input string x, with $|x| = n$, *Merlin* sends Arthur a polynomial-size circuit C, which is supposed to be the circuit C_n for the Prover's strategy for L.
- *Arthur* simulates the interactive proof for L, using C as the Prover, and making random choices to simulate the Verifier. If the input is *not* in the language, then *no* Prover has a chance of convincing the Verifier, and so C cannot prove the Verifier.

The **MA** protocol we described takes any $L \in \mathbf{EXP}$, and so it implies that $\mathbf{MA} \subseteq \mathbf{EXP} \Rightarrow \mathbf{MA} = \mathbf{EXP}$. $\square$

8.1.1 Relativizions Of The Above

We can get some weak relativizations of Theorem 8.1:

Theorem 8.3. *For any A in* $\boldsymbol{EXP}$*, if* $\boldsymbol{EXP}^A$ *is in* $\boldsymbol{P}^A_{/poly}$*, then:* $\boldsymbol{NEXP}^A = \boldsymbol{EXP}^A$.

We can do better if A is complete for some level of the polynomial-time hierarchy! In this (final) section, we will prove the following theorem:

Theorem 8.4. *Let A be complete for Σ_k^p, for any $k \geq 0$. If* $\boldsymbol{NEXP}^A$ *is in* $\boldsymbol{P}^A_{/poly}$*, then* $\boldsymbol{NEXP}^A = \boldsymbol{MA}^A = \boldsymbol{EXP}$.

From the above theorem, we have a very interesting corollary:

Theorem 8.5. *There is at most one k such that* $\boldsymbol{NEXP}^{\Sigma_k^p}$ *is in* $\boldsymbol{P}^{\Sigma_k^p}_{/poly}$.

In order to prove Theorem 8.4, we expose some necessary tools:

Theorem 8.6. *For any A, if* $\boldsymbol{EXP}$ *is in* $\boldsymbol{P}^A_{/poly}$*, then* $\boldsymbol{EXP} \subseteq \boldsymbol{MA}^A$.

Theorem 8.7. *For all A, if* $\boldsymbol{NEXP}^A$ *is in* $\boldsymbol{P}^A_{/poly}$ *and* $\boldsymbol{EXP}^A$ *is in* $\boldsymbol{AM}^A$*, then* $\boldsymbol{NEXP}^{NP} = \boldsymbol{NEXP}$.

Theorem 8.8. *For any $k \geq 0$, if* $\boldsymbol{EXP}^{\Sigma_k^p} \subseteq \boldsymbol{EXP}_{/poly}$*, then* $\boldsymbol{EXP}^{\Sigma_k^p} = \boldsymbol{EXP}$

Appendix A

Quantifier Characterizations

A.1 Complexity Classes

A.1.1 Introduction

We present an alternative characterization of complexity classes using *quantifiers*, and especially those needed for the quantification implied by the definition of each class. This notation provides a uniform description of complexity classes defined in various contexts (deterministic, probabilistic, interactive), and we'll be able to obtain immediate relations and inclusions among them.

For complexity classes like **P**, **NP** and their generalizations, the classical existential and universal quantifiers suffice, but in order to describe classes using Probabilistic Turing Machines, we will need a new one, which assures that a computation has "probabilistic" advantage:

Definition A.1 (Majority Quantifier)**.** *Let* $R : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$ *be a predicate, and* ε *a rational number in* $\left(0, \frac{1}{2}\right)$*. We denote by* $(\exists^+ y, |y| = k)R(x,y)$ *the following predicate:*

> *"There exist at least* $\left(\frac{1}{2} + \varepsilon\right) \cdot 2^k$ *strings* y *of length* k *for which* $R(x,y)$ *holds."*

We call $\exists^+$ *the* overwhelming majority *quantifier.*

The overwhelming quantifier provides a "threshold" for the number of certificates, assuring that the fraction of 2^k possible strings in $\{0,1\}^k$ (that is, of length k) which accepts the computation (or satisfies the predicate R) is bounded away from 50% by a fixed amount ε.

We can generalize this quantifier by attaching the fraction of accepting computations as a parameter. That is, $\exists_r^+$ means that the fraction r of the possible certificates of a certain length satisfy the predicate for the certain input. It is easy to see that: $\exists^+ = \exists^+_{1/2+\varepsilon} = \exists^+_{2/3} = \exists^+_{3/4} = \exists^+_{0.99} = \exists^+_{1-2^{p(|x|)}}$, where $|x|$ denotes the length of the input x. Intuitively, this means that we can "increase" the fraction of the accepting branches (the acceptance probability) by indepedent repetitions of the computation.

We also introduce a new notation for an arbitrary complexity class, which utilizes the quantifiers' role in the classical definition:

Definition A.2. *We denote as* $\mathcal{C} = (Q_1/Q_2)$, *where* $Q_1, Q_2 \in \{\exists, \forall, \exists^+\}$, *the class* $\mathcal{C}$ *of languages* L *satisfying:*

- $x \in L \Rightarrow Q_1 y \ R(x, y)$
- $x \notin L \Rightarrow Q_2 y \ \neg R(x, y)$

In the above definition, we easily notice that:

$$co\mathcal{C} = co(Q_1/Q_2) = (Q_2/Q_1)$$

So, using the classical existential and universal quantifiers we can define the basic complexity classes, by implying their definitional properties. For example, for languages in class **P** there is a computation path which either accepts, either rejects. So, it is easy to see that $\mathbf{P} = (\forall/\forall)$.

On the other hand, for languages in class **NP** there is a computation tree for each input, and we accept it if *there is* an accepting branch, or we reject it if *all* the branches reject. Hence, we have that: $\mathbf{NP} = (\exists/\forall)$. The complementary class *co***NP** can be also defined as $co\mathbf{NP} = (\forall/\exists)$.

A family of complexity classes that are naturally defined by alternating quantifiers is the Polynomial Hierarchy. These classes can be considered as a natural generalization of **NP**. Recall that:

Definition A.3 (Polynomial-Time Hierarchy)**.** *A language* $L \in \Sigma_k^p$, $k \in \mathbb{N}$, *iff there exists a polynomial-time computable predicate* $R(x, y_1, y_2, \ldots, y_k)$, *such that, for* $|y_i| \leq p(n)$, $i \in \{1, \ldots, k\}$, $p \in poly(n)$*:*

$$x \in L \Leftrightarrow \exists y_1 \forall y_2 \exists y_3 \cdots Q_k y_k \ R(x, y_1, y_2, \ldots, y_k)$$

where Q_k *is* $\exists$ *if* k *is odd, and* $\forall$ *if* k *is even.*

Also, a language $L \in \Pi_k^p$ *iff there exists a polynomial-time computable predicate* $R(x, y_1, y_2, \ldots, y_k)$, *such that, for* $|y_i| \leq p(n)$, $i \in \{1, \ldots, k\}, p \in poly(n)$*:*

$$x \in L \Leftrightarrow \forall y_1 \exists y_2 \forall y_3 \cdots Q_k y_k \ R(x, y_1, y_2, \ldots, y_k)$$

where Q_k *is* $\forall$ *if* k *is odd, and* $\exists$ *if* k *is even.*

An equivalent definition can be given recursively using oracles: $\Sigma_k^p = \mathbf{NP}^{\Sigma_{k-1}^p}$ and $\Pi_k^p = co\mathbf{NP}^{\Sigma_{k-1}^p}$, while $\Sigma_0^p = \Pi_0^p = \mathbf{P}$. So, we have that $\Sigma_1^p = \mathbf{NP}$, $\Pi_1^p = co\mathbf{NP}$, $\Sigma_2^p = \mathbf{NP}^{\mathbf{NP}}$ and so on.
Using quantifier notation, we can re-define these complexity classes as:

- $\Sigma_2^p = (\exists\forall/\forall\exists)$, $\Pi_2^p = (\forall\exists/\exists\forall)$, and in general:
- $\Sigma_k^p = (\exists\forall \cdots Q_m)/\forall\exists \cdots Q_n)$, where:
  - Q_m represents $\exists$, if k is odd, or $\forall$, if k is even, and
  - Q_n represents $\forall$, if k is odd, or $\exists$, if k is even.
- $\Pi_k^p = (\forall\exists \cdots Q_m/\exists\forall \cdots Q_n)$, where:
  - Q_m represents $\forall$, if k is odd, or $\exists$, if k is even.
  - Q_n represents $\exists$, if k is odd, or $\forall$, if k is even.

A.1.2 Randomized Classes

Using the overwhelming majority quantifier, the following characterizations are immediate from the definition of each class:

- **BPP** (Bounded two-sided error, *"Monte-Carlo"*):

  By **BPP** 's definition we have:

$$\begin{cases} x \in L \Rightarrow \mathbf{Pr}\left[accept\right] \geq 2/3 \\ x \notin L \Rightarrow \mathbf{Pr}\left[reject\right] \geq 2/3 \end{cases} \Rightarrow$$

$$\begin{cases} x \in L \Rightarrow \mathbf{Pr}\left[R(x)\right] \geq 2/3 \\ x \notin L \Rightarrow \mathbf{Pr}\left[\neg R(x)\right] \geq 2/3 \end{cases}, \text{ for a predicate } R \in \mathbf{P} \Rightarrow$$

$$\begin{cases} x \in L \Rightarrow \exists^+ y \ R(x,y) \\ x \notin L \Rightarrow \exists^+ y \ \neg R(x,y) \end{cases} \Rightarrow \mathbf{BPP} = (\exists^+/\exists^+)$$

- **RP** (Bounded one-sided error, *"Atlantic City"*):
  Similarly:

$$\begin{cases} x \in L \Rightarrow \mathbf{Pr}\left[accept\right] \geq 2/3 \\ x \notin L \Rightarrow \mathbf{Pr}\left[reject\right] = 1 \end{cases} \Rightarrow$$

$$\begin{cases} x \in L \Rightarrow \mathbf{Pr}\left[R(x)\right] \geq 2/3 \\ x \notin L \Rightarrow \mathbf{Pr}\left[\neg R(x)\right] = 1 \end{cases}, \text{ for a predicate } R \in \mathbf{P} \Rightarrow$$

$$\begin{cases} x \in L \Rightarrow \exists^+ y \ R(x,y) \\ x \notin L \Rightarrow \forall y \ \neg R(x,y) \end{cases} \Rightarrow \mathbf{RP} = (\exists^+/\forall)$$

- Obviously, $co\mathbf{RP} = (\forall/\exists^+)$

So, we have created alterative definitions for the most usual complexity classes. Now, we can explore what kind of "operations" we can perform with these quantifiers. Firstly, we determine when we can swap $\forall$ and $\exists^+$:

Lemma A.1 (Swapping Lemma). *Let $R(x, y, z)$ be a predicate that holds only if $|y| = |z| = p(n)$ for some polynomial p, where $n = |x|$, and let C be a set of strings such that $\forall v \in C\ |v| = p(n)$ and $|C| \leq p(n)$. Then, for $|y| = |z| = p(n)$:*

i. $\forall y \exists^+ z\ R(x, y, z) \Rightarrow \exists^+ C \forall y\ \bigvee_{z \in C} R(x, y, z)$

ii. $\forall z \exists^+_{1-2^{-n}} y\ R(x, y, z) \Rightarrow \forall C \exists^+ y\ \bigwedge_{z \in C} R(x, y, z)$

Proof: (i) Assume that $\forall y \exists^+ z\ R(x, y, z)$ holds. Let $p \in poly(n)$ such that for all y with $|y| \leq p(n)$ and considering only z with $|z| \leq p(n)$: $\mathbf{Pr}\left[\{z|\ R(x, y, z)\}\right] > \frac{1}{2} + \varepsilon$. Also, let $q(n) = p(n) + 3$. We will estimate the probability of the event $\neg \forall y\ \bigvee_{z \in C} R(x, y, z)$:

$$\mathbf{Pr}\left[\left\{C \mid\ \exists y : \bigwedge_{z \in C} \neg R(x, y, z)\right\}\right] = \mathbf{Pr}\left[\bigcup_{|y| \leq p(n)} \left\{C \mid \bigwedge_{z \in C} \neg R(x, y, z)\right\}\right]$$

$$\leq \sum_{|y| \leq p(n)} \mathbf{Pr}\left[\left\{C \mid \bigwedge_{z \in C} \neg R(x, y, z)\right\}\right] \leq \sum_{|y| \leq p(n)} \prod_{i=1}^{q(n)} \frac{1}{2} \leq 2^{p(n)+1} \cdot \left(\frac{1}{2}\right)^{q(n)} \leq \frac{1}{4}$$

Note that the predicate $R'(x, y, z) = \bigvee_{z \in C} R(x, y, z)$ is polynomial-time computable, therefore for most of the C: $\bigvee_{z \in C} R(x, y, z)$, that is $\exists^+ C \forall y\ \bigvee_{z \in C} R(x, y, z)$.

(ii) Without loss of generality, we can assume that $\forall x \forall z\ \mathbf{Pr}\left[\{z|\ R(x, y, z)\}\right] \geq 1 - 1/2^{p(n)}$ for some $p \in poly(n)$. So, for any z, $|z| = p(n)$, we have that $\mathbf{Pr}\left[\neg R(x, y, z)\right] \leq 2^{p(n)}$. For a given C, $|C| \leq q(n)$:

$$\mathbf{Pr}\left[\left\{y \mid \bigvee_{z \in C} \neg R(x, y, z)\right\}\right] \leq \sum_{z \in C} \mathbf{Pr}\left[\{y | \neg R(x, y, z)\}\right] \leq \frac{q(n)}{2^{p(n)}} < \frac{1}{4}$$

for sufficiently large n. Therefore, we have that $\forall C \exists^+ y\ \bigwedge_{z \in C} R(x, y, z)$. $\square$

The above lemma, can be viewed in terms of a binary matrix A of size $2^{p(n)} \times 2^{p(n)}$, with $A(y, z) \Leftrightarrow R(x, y, z)$. The (i) part states that if every row of A has more than $(2/3)p(n)$ many 1's, then for the majority of the choices of $p(n)$ many columns, every row of A contains at least one 1 in these columns. Similarly for part (ii).

We can prove, using the Swapping Lemma, an alternative, "decisive" characterization of **BPP**, stated in the following theorem:

Theorem A.2 (BPP Theorem). *The following are equivalent:*

i. $L \in \mathbf{BPP}$.

ii. There exists a polynomial-time computable predicate R and a polynomial p, such that for all x, with $|x| = n$, and $|y| = |z| = p(n)$:

$$x \in L \Rightarrow \exists^+ y \forall z \ R(x, y, z)$$

$$x \notin L \Rightarrow \forall y \exists^+ z \ \neg R(x, y, z)$$

iii. There exists a polynomial-time computable predicate R and a polynomial p, such that for all x, with $|x| = n$, and $|y| = |z| = p(n)$:

$$x \in L \Rightarrow \forall y \exists^+ z \ R(x, y, z)$$

$$x \notin L \Rightarrow \exists^+ y \forall z \ \neg R(x, y, z)$$

Proof: $(i \Rightarrow ii)$ Let $L \in \mathbf{BPP}$. Then, by definition, there exists a polynomial-time computable predicate Q and a polynomial q such that for all x's of length n:

$$x \in L \Rightarrow \exists^+ y \ Q(x, y)$$
$$x \notin L \Rightarrow \exists^+ y \ \neg Q(x, y)$$

Using Lemma A.1(i) we have[1], for all x's of length n and for some $y, z, |y| = |z| = q(n)$:
$x \in L \Rightarrow \exists^+ z \ Q(x, z) \Rightarrow \forall y \exists^+ z \ Q(x, y \oplus z) \Rightarrow \exists^+ C \forall y \, [\exists (z \in C) \ Q(x, y \oplus z)]$,
where C denotes (as in the Swapping's Lemma formulation) a set of $q(n)$ strings, each of length $q(n)$.
On the other hand, by using Lemma A.1(ii) we similarly have:
$x \notin L \Rightarrow \exists^+ y \ \neg Q(x, z) \Rightarrow \forall z \exists^+ y \ \neg Q(x, y \oplus z) \Rightarrow \forall C \exists^+ y \, [\forall (z \in C) \ \neg Q(x, y \oplus z)]$.
Now, we only have to assure that the appeared predicates $\exists z \in C \ Q(x, y \oplus z)$ and $\forall z \in C \ \neg Q(x, y \oplus z)$ are computable in polynomial time (Note that the above expressions are equivalent to $\bigvee_{z \in C} \neg R(x, y, z)$ and $\bigwedge_{z \in C} \neg R(x, y, z)$ we met in Swapping Lemma.): Recall that in Swapping Lemma's formulation we demanded $|C| \leq p(n)$ and that for each $v \in C : \ |v| = p(n)$. This means that we seek if a string of polynomial length *exists*, or if the predicate holds *for all* such strings in a set with polynomial cardinality, procedure which can be surely done in polynomial time.

$(ii \Rightarrow i)$ Conversely, assume that *there exists* a predicate R and a polynomial p, as stated is (ii). Then, for each string w of length $2p(n)$, we "divide" it in two halfs w_1, w_2, such that $w = w_1 \circ w_2$ and $|w_1| = |w_2| = p(n)$. Then, for each x with $|x| = n$, and $|y| = |z| = p(n)$:

[1] We define the XOR (eXclusive OR) operator $\oplus$ of two strings of the equal length as the bit-by-bit $mod2$ addition. That is: $0 \oplus 0 = 1 \oplus 1 = 0$, and $0 \oplus 1 = 1 \oplus 0 = 1$.

$x \in L \Rightarrow \exists^+ y \forall z\ R(x,y,z) \Rightarrow \exists^+ w(|w| = 2p(n))\ R(x, w_1, w_2)$
$x \notin L \Rightarrow \forall y \exists^+ z\ R(x,y,z) \Rightarrow \exists^+ w(|w| = 2p(n))\ \neg R(x, w_1, w_2)$

($i \Rightarrow iii$) It follows immediately from the fact that **BPP** is closed under complementation (co**BPP** = **BPP**). □

In other words, Theorem A.2 states that:

$$\mathbf{BPP} = (\exists^+\forall / \forall\exists^+) = (\forall\exists^+ / \exists^+\forall) \tag{A.1}$$

The above characterization of **BPP** is *decisive* in the sense that if we replace the $\exists^+$ quantifier with $\exists$ (if "+" is dropped), then we can decide whether $x \in L$ or $x \notin L$. That is, the two predicates are still complementary[2] to each other, so exactly one holds for x. Note that this doesn't hold for the $(\exists^+/\exists^+)$ characterization of **BPP**, because if we replace the $\exists^+$ quantifier with $\exists$, the two resulting predicates are not complementary, and they do not define a complexity class.

By replacing in (A.1) the quantifier $\exists^+$ with $\exists$ (*why is this possible?*) we can obtain immediately the following result, known as the Sipser-Gács Theorem:

Corollary A.3. $\mathbf{BPP} \subseteq \Sigma_2^p \cap \Pi_2^p$

Theorem A.2 can be generalized for sequences of quantifiers (denoted as $\mathbf{Q}_i$):

Corollary A.4.

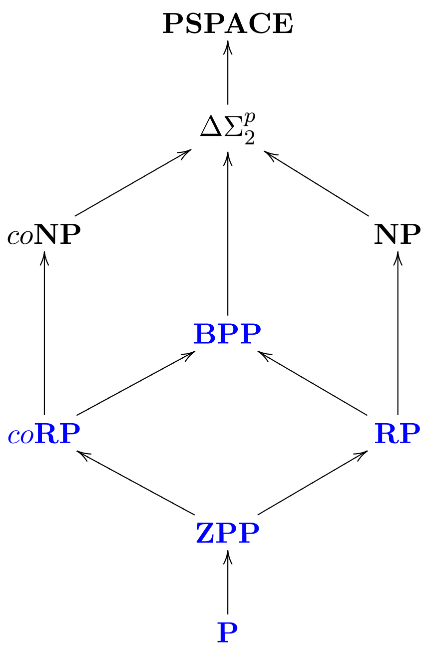
$$(\mathbf{Q}_1\exists^+\mathbf{Q}_2/\mathbf{Q}_3\exists^+\mathbf{Q}_4) = (\mathbf{Q}_1\exists^+\forall\mathbf{Q}_2/\mathbf{Q}_3\forall\exists^+\mathbf{Q}_4) = (\mathbf{Q}_1\forall\exists^+\mathbf{Q}_2/\mathbf{Q}_3\exists^+\forall\mathbf{Q}_4)$$

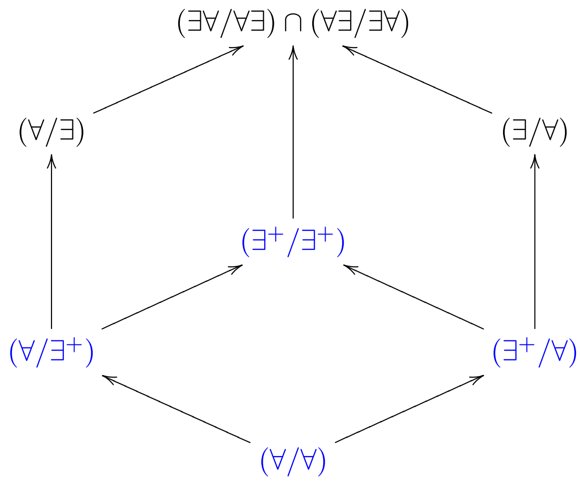
Using quantifier characterizations, we also have trivially many inclusions between complexity classes:

- $\mathbf{P} \subseteq \mathbf{RP}$, since $(\forall/\forall) \subseteq (\exists^+/\forall)$ (*for all* implies *for most*).
- $\mathbf{RP} \subseteq \mathbf{BPP}$, since $(\exists^+/\forall) \subseteq (\exists^+/\exists^+)$ (same reason).
- $\mathbf{RP} \subseteq \mathbf{NP}$, since $(\exists^+/\forall) \subseteq (\exists/\forall)$ (*for most* implies *for at least one*).

The main inclusions are depicted in the following Hasse diagrams ("$\rightarrow$" denotes "$\subseteq$"):

[2] Two predicates R and P are called complementary if $R \Rightarrow \neg P$.





A.2 Arthur-Merlin Games

A.2.1 Introduction

In this section, we consider the *interaction* model between two Turing Machines as a "game". This setting is very useful to Complexity Theory, for placing upper bounds in problems' complexity, and on the other hand in Cryptography, for proving the security of cryptographic protocols against (efficient) computational attacks. The terminology used in this games is mainly anthropomoprphic, known as "Arthur-Merlin" Games.

> "King Arthur recognizes the supernatural intellectual abilities of Merlin, but doesnt trust him. How should Merlin convince the intelligent but impatient King that a string x belongs to a given language L? If $L \in \mathbf{NP}$, Merlin will be able to present a witness which Arthur can check in polynomial time." From [Bab85]

In the above, Arthur is an ordinary player with the ability of making coin tosses (i.e. randomization), and Merlin is a powerful player capable of optimizing his winning chances at every move. The two players alternate moves, the history of the game is known to both, and after k moves there is a deterministic polynomial-time Turing Machine that reads the history and decides who wins. We state the formal definition:

Definition A.4 (Arthur-Merlin Games)**.** *An Arthur-Merlin Game is a pair of interactive Turing Machines* $\boldsymbol{A}$ *and* $\boldsymbol{M}$*, and a predicate* ρ *such that:*

- *On an input x, with length $|x| = n$, exactly $q(n)$ messages of length $m(n)$ each are exchanged, where $q, m \in poly(n)$.*
- Arthur *plays first, and at iteration $1 \leq i \leq q(n)$ chooses* uniformly at random *a string r_i, where $|r_i| = m(n)$.*
- Merlin*'s reply in the i^{th} iteration, denoted y_i, is a function of all previous choices of Arthur and x. That is: $y_i = M(x, r_1, r_2, \ldots, r_i)$. In other words, M is the strategy of Merlin.*
- *For every Turing Machine $\mathbf{M}'$, a conversation between $\boldsymbol{A}$ and $\mathbf{M}'$ on input x is a string:*
$$r_1 y_1 r_2 y_2 \cdots r_{q(n)} y_{q(n)}$$
where for every $1 \leq i \leq q(n)$: $y_i = \mathbf{M}'(x, r_1 r_2 \cdots r_i)$
- *The predicate ρ maps x and a conversation $r_1 y_1 r_2 y_2 \cdots r_{q(n)} y_{q(n)}$ to $\{accept, reject\}$ in polynomial time, and it is called* value-of-the game *predicate.*

Now we need to determine how to test the membership for a language L using an Arthur-Merlin game: Firstly, we define the set of all conversations between Arthur and Merlin as $CONV_x^M$. Obviously, we have that $|CONV_x^M| = 2^{q(n)m(n)}$. We also define the set of accepting conversations $ACC_x^{\rho,M}$ as:

$$\{r_1 \cdots r_{q(n)} | \exists (y_1 \cdots y_{q(n)}) : (r_1 y_1 \cdots r_{q(n)} y_{q(n)}) \in CONV_x^M \wedge \rho(r_1 y_1 \cdots r_{q(n)} y_{q(n)}) = accept\}$$

Intuitively, $ACC_x^{\rho,M}$ is the set of all random choices leading Arthur to accept the input x when interacting with Merlin, and it depends only on Merlin and the pridecate ρ, given that Arthur follows the protocol. The probability that Arthur accepts x is:

$$\mathbf{Pr}[Arthur\ accepts\ x] = \frac{|ACC_x^{\rho,M}|}{|CONV_x^M|}$$

Definition A.5. *A language L is in $\mathbf{AM}[k]$ if there exists a k-move Arthur-Merlin protocol such that for every $x \in \Sigma^*$:*

- *If $x \in L$, there exists a strategy for Merlin such that :*
$$\mathbf{Pr}[Arthur\ accepts\ x] \geq \frac{2}{3}$$
- *If $x \notin L$, for every strategy for Merlin we have:*
$$\mathbf{Pr}[Arthur\ accepts\ x] \leq \frac{1}{3}$$

The first is known as completeness *condition, and the second as* soundness *condition.*

The class $\mathbf{MA}[k]$ is defined by similar way, but Merlin plays first.

A.2.2 Quantifier Characterizations

We denote by $\mathbf{AM} = \mathbf{AM}[2]$, and by $\mathbf{MA} = \mathbf{MA}[2]$. Following [Bab85], we consider as Merlin an **NP** machine, and as Athur a **BPP** machine. So, we can interpret Arthur-Merlin games in terms of quantifiers:

$$\mathbf{AM} = (\exists^+\exists/\exists^+\forall) = \mathcal{BP}\cdot\mathbf{NP}$$

$$\mathbf{MA} = (\exists\exists^+/\forall\exists^+) = \mathcal{N}\cdot\mathbf{BPP}$$

where $\mathcal{BP}\cdot$ and $\mathcal{N}\cdot$ is the bounded-probabilistic and the nondeterministic quantifiers respectively (see Appendix A.3 for definitions). It is well known that we can obtain *perfect completeness* for interactive proof systems, by simulating the given protocol by another. This cannot be obtained in the soundness condition, because this would be equal to a deterministic verifier, so by definition that class collapses to **NP**. We prove perfect completeness for Arthur-Merlin games in the following theorem:

Theorem A.5. *i.* $\mathbf{AM} = (\exists^+\exists/\exists^+\forall) = (\forall\exists/\exists^+\forall)$

ii. $\mathbf{MA} = (\exists\exists^+/\forall\exists^+) = (\exists\forall/\forall\exists^+)$

iii. In general, for even k and $\mathbf{AM}[k] = (\mathbf{Q}_1/\mathbf{Q}_2)$*:*

- $\mathbf{AM}[k+1] = (\mathbf{Q}_1\exists^+/\mathbf{Q}_2\exists^+) = (\mathbf{Q_1}\forall/\mathbf{Q_2}\exists^+)$
- $\mathbf{AM}[k+2] = (\mathbf{Q}_1\exists^+\exists/\mathbf{Q}_2\exists^+\forall) = (\mathbf{Q_1}\forall\exists/\mathbf{Q_2}\exists^+\forall)$

Proof: (i) $\mathbf{AM} = (\exists^+\exists/\exists^+\forall) = (\forall\exists^+\exists/\exists^+\forall\forall)$ (by Corollary A.4)
$\subseteq (\forall\exists\exists/\exists^+\forall\forall) = (\forall\exists/\exists^+\forall)$ (by quantifier contraction).
The other direction is trivial: $(\forall\exists/\exists^+\forall) \subseteq (\exists^+\exists/\exists^+\forall) = \mathbf{AM}$.

(ii) $\mathbf{MA} = (\exists\exists^+/\forall\exists^+) = (\exists\exists^+\forall/\forall\forall\exists^+)$ (by Corollary A.4)
$\subseteq (\exists\exists\forall/\forall\forall\exists^+) = (\exists\forall/\forall\exists^+)$ (by quantifier contraction).
The other direction is trivial: $(\exists\forall/\forall\exists^+) \subseteq (\exists\exists^+/\forall\exists^+) = \mathbf{MA}$.

(iii) $\mathbf{AMA} = (\exists^+\exists\exists^+/\exists^+\forall\exists^+) = (\forall\exists\exists^+/\exists^+\forall\exists^+)$ (by (ii))
$= (\forall\exists\exists^+\forall/\exists^+\forall\forall\exists^+)$ (by Corollary A.4)
$= (\forall\exists\forall/\exists^+\forall\exists^+)$ (by quantifier contraction)
and so on for $\mathbf{AM}[k]$. □

We also prove the following useful lemma:

Lemma A.6. $(\exists\forall/\forall\exists^+) \subseteq (\forall\exists/\exists^+\forall)$

Proof: Let $L \in (\exists\forall/\forall\exists^+)$. Then,
$x \notin L \Rightarrow \forall y\ \exists^+ z\ \neg P(x,y,z)$
$\Rightarrow \exists^+ C\ \forall y\ \exists z \in C\ \neg P(x,y,z)$ (by the Swapping Lemma A.1i)
$\Rightarrow \exists C\ \forall y\ \exists z \in C\ \neg P(x,y,z)$

$\Rightarrow \forall y\ \exists z\ \neg P(x, y, z)$
$\Rightarrow x \notin L$
which means that all logical implications are indeed equivalences, and the second and third lines emply that $L \in (\forall\exists/\exists^{+}\forall)$. □

From the above theorem and lemma, we have the following immediate inclusions:

Corollary A.7. $\mathbf{MA} \subseteq \mathbf{AM}$

Corollary A.8. $\mathbf{AM} \subseteq \Pi_2^p$ *and* $\mathbf{MA} \subseteq \Sigma_2^p \cap \Pi_2^p$

Lemma A.6 can be generalized as follows:

Corollary A.9.

$$(\mathbf{Q}_1\exists\forall\mathbf{Q}_2/\mathbf{Q}_3\forall\exists^{+}\mathbf{Q}_4) \subseteq (\mathbf{Q}_1\forall\exists\mathbf{Q}_2/\mathbf{Q}_3\exists^{+}\forall\mathbf{Q}_4)$$

If we consider the complexity classes $\mathbf{AM}[k]$ (the languages that have Arthur-Merlin proof systems of a bounded number of rounds), they form an *hierarchy*:

$$\mathbf{AM}[0] \subseteq \mathbf{AM}[1] \subseteq \cdots \subseteq \mathbf{AM}[k] \subseteq \mathbf{AM}[k+1] \subseteq \cdots$$

Unlike the Polynomial Hierarchy, in which we believe the inclusions are *proper*, Arthur-Merlin Hierarchy collapses to the second level (which is why we usually denote as **AM** the class $\mathbf{AM}[2]$):

Theorem A.10. *For constants* $k \geq 2$, $\mathbf{AM}[k] = \mathbf{AM}[2]$.

Proof. We show as special case the inclusion $\mathbf{MAM} \subseteq \mathbf{AM}$:
$\mathbf{MAM} = (\exists\exists^{+}\exists/\forall\exists^{+}\forall) \subseteq (\exists\exists^{+}\forall\exists/\forall\forall\exists^{+}\forall)$ (by the **BPP** Theorem A.2)
$\subseteq (\exists\forall\exists/\forall\exists^{+}\forall)$ (by quantifier contraction)
$\subseteq (\forall\exists\exists/\exists^{+}\forall\forall)$ (by Lemma A.6)
$\subseteq (\forall\exists/\exists^{+}\forall) = \mathbf{AM}$ (by quantifier contraction) □

We give an alternative proof of a result which provides us with strong evidence that $co\mathbf{NP} \nsubseteq \mathbf{AM}$, originally proved in [BHZ87]:

Theorem A.11. *If* $co\mathbf{NP} \subseteq \mathbf{AM}$*, then:*

i. **PH** *collapses at the second level, and*

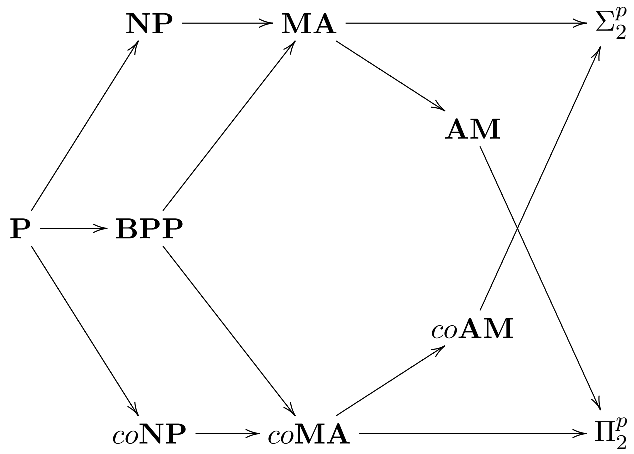
ii. $\mathbf{PH} = \mathbf{AM}$.

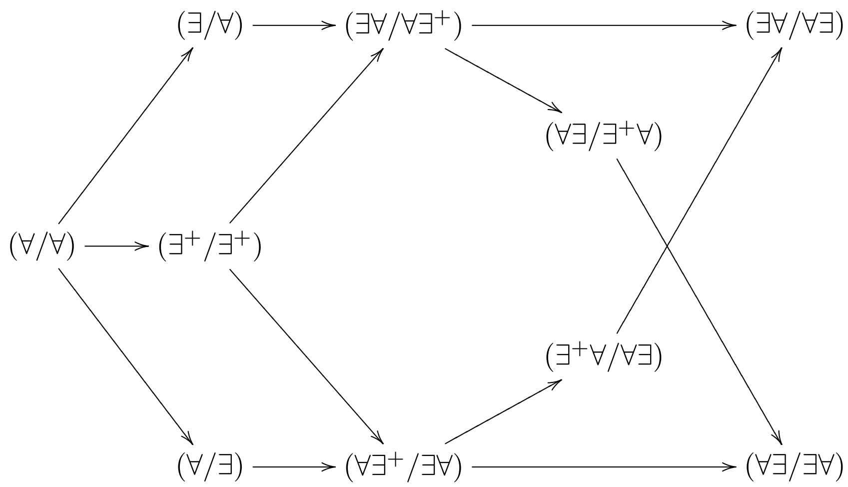
Proof: Since $co\mathbf{NP} \subseteq \mathbf{AM}$, we have that $(\forall/\exists) \subseteq (\forall\exists/\exists^{+}\forall)$ as assumption. Then:

$$\Sigma_2^p = (\exists\forall/\forall\exists) \subseteq (\exists\forall\exists/\forall\exists^{+}\forall) \subseteq (\forall\exists\exists/\exists^{+}\forall\forall) = (\forall\exists/\exists^{+}\forall) = \mathbf{AM} \subseteq (\forall\exists/\exists\forall) = \Pi_2^p$$

The first inclusion holds from our hypothesis, the second by Lemma A.6. □

The following Hasse diagrams captures the inclusions between the most important complexity classes we've seen so far, the former in classic and the latter in quantifier notation:





A.3 Operators on Complexity Classes

Definition A.6 (Operators on Complexity Classes)**.** *Let* $\mathbf{C}$ *be an arbitrary complexity class. We define:*

1. *The* ***complement*** *operator co*$\mathbf{C}$*:*
   A language $L \in co\mathbf{C}$ *if there exists an* $L' \in \mathbf{C}$ *such that:*
   - *If* $x \in L \Rightarrow x \notin L'$
   - *If* $x \notin L \Rightarrow x \in L'$
2. *The* ***nondeterministic*** *operator* $\mathcal{N}$*:*
   A language $L \in \mathcal{N} \cdot \mathbf{C}$ *if there exists an* $L' \in \mathbf{C}$ *such that:*

Class	Definition		Notation
P	$x \in L \Rightarrow R(x)$	$x \notin L \Rightarrow \neg R(x)$	$(\forall/\forall)$
NP	$x \in L \Rightarrow \exists y\ R(x,y)$	$x \notin L \Rightarrow \forall y\ \neg R(x,y)$	$(\exists/\forall)$
co**NP**	$x \in L \Rightarrow \forall y\ R(x,y)$	$x \notin L \Rightarrow \exists y\ \neg R(x,y)$	$(\forall/\exists)$
Σ_2^p	$x \in L \Rightarrow \exists y \forall z\ R(x,y,z)$	$x \notin L \Rightarrow \forall y \exists z\ \neg R(x,y,z)$	$(\exists\forall/\forall\exists)$
Π_2^p	$x \in L \Rightarrow \forall y \exists z\ R(x,y,z)$	$x \notin L \Rightarrow \exists y \forall z\ \neg R(x,y,z)$	$(\forall\exists/\exists\forall)$
RP	$x \in L \Rightarrow \exists^+ y\ R(x,y)$	$x \notin L \Rightarrow \forall y\ \neg R(x,y)$	$(\exists^+/\forall)$
co**RP**	$x \in L \Rightarrow \forall y\ R(x,y)$	$x \notin L \Rightarrow \exists^+ y\ \neg R(x,y)$	$(\forall/\exists^+)$
BPP	$x \in L \Rightarrow \exists^+ y\ R(x,y)$	$x \notin L \Rightarrow \exists^+ y\ \neg R(x,y)$	$(\exists^+/\exists^+)$
	Alternative characterization [ZH86]:		$(\exists^+\forall/\forall\exists^+)$
	Alternative characterization [ZH86]:		$(\forall\exists^+/\exists^+\forall)$
PP	$x \in L \Rightarrow \exists_{1/2} y\ R(x,y)$	$x \notin L \Rightarrow \exists_{1/2} y\ \neg R(x,y)$	$(\exists_{1/2}/\exists_{1/2})$
AM	$x \in L \Rightarrow \exists^+ y\ R(x,y)$	$x \notin L \Rightarrow \exists^+ y\ \neg R(x,y)$	$(\exists^+\exists/\exists^+\forall)$
	Alternative characterization [ZF87]:		$(\forall\exists/\exists^+\forall)$
MA	$x \in L \Rightarrow \exists^+ y\ R(x,y)$	$x \notin L \Rightarrow \exists^+ y\ \neg R(x,y)$	$(\exists\exists^+/\forall\exists^+)$
	Alternative characterization [ZF87]:		$(\exists\forall/\forall\exists^+)$

Table A.1: Quantifier Notation of the usual Complexity Classes

- *If* $x \in L \Rightarrow \exists y\ R_{L'}(x,y)$
- *If* $x \notin L \Rightarrow \forall y\ \neg R_{L'}(x,y)$

3. *The **intersection** operator* Δ*:*
A language $L \in \Delta\cdot \mathbf{C}$ *if* $L \in \mathbf{C}$ *and also* $\overline{L} \in \mathbf{C}$*, that is if* $L \in \mathbf{C} \cap co\mathbf{C}$*.*

4. *The **bounded-probabilistic** operator* $\mathcal{BP}$*:*
A language $L \in \mathcal{BP}\cdot \mathbf{C}$ *if there exists an* $L' \in \mathbf{C}$ *such that:*

   - *If* $x \in L \Rightarrow \exists^+ y\ R_{L'}(x,y)$
   - *If* $x \notin L \Rightarrow \exists^+ y\ \neg R_{L'}(x,y)$

5. *The **probabilistic** operator* $\mathcal{P}$*:*
A language $L \in \mathcal{P}\cdot \mathbf{C}$ *if there exists an* $L' \in \mathbf{C}$ *such that:*

   - *If* $x \in L \Rightarrow \exists_{1/2} y\ R_{L'}(x,y)$
   - *If* $x \notin L \Rightarrow \exists_{1/2} y\ \neg R_{L'}(x,y)$

6. *The **probabilistic** operator* $\mathcal{R}$*:*
A language $L \in \mathcal{R}\cdot \mathbf{C}$ *if there exists an* $L' \in \mathbf{C}$ *such that:*

   - *If* $x \in L \Rightarrow \exists^+ y\ R_{L'}(x,y)$
   - *If* $x \notin L \Rightarrow \forall y\ \neg R_{L'}(x,y)$

In the above definitions, $|y| \leq poly(|x|)$, and R_L is a polynomial-time computable predicate responding to the membership question for L. That is, $R_L(x) = 1$ iff $x \in L$ and $R_L(x, y) = 1$ iff $x; y \in L$. Note that the above operations require that $\mathbf{C}$ is closed under padding.

	$(\mathbf{Q}_1/\mathbf{Q}_2)$	**P**	**NP**	*co***NP**	Σ_i^p	Π_i^p	**PP**	**BPP**	**RP**	**ZPP**
$co\ \cdot$	$(\mathbf{Q}_2/\mathbf{Q}_1)$	**P**	*co***NP**	**NP**	Π_i^p	Σ_i^p	**PP**	**BPP**	*co***RP**	**ZPP**
$\mathcal{N}\ \cdot$	$(\exists\mathbf{Q}_1/\forall\mathbf{Q}_2)$	**NP**	**NP**	Σ_2^p	Σ_i^p	Σ_{i+1}^p		**MA**		
$\Delta\ \cdot$		**P**	$\mathbf{NP} \cap co\mathbf{NP}$	$\mathbf{NP} \cap co\mathbf{NP}$	$\Sigma_i^p \cap \Pi_i^p$	$\Sigma_i^p \cap \Pi_i^p$	**PP**	**BPP**	**ZPP**	**ZPP**
$\mathcal{BP}\ \cdot$	$(\exists^+\mathbf{Q}_1/\exists^+\mathbf{Q}_2)$	**BPP**	**AM**	*co***AM**				**BPP**		
$\mathcal{P}\ \cdot$	$(\exists_{1/2}\mathbf{Q}_1/\exists_{1/2}\mathbf{Q}_2)$	**PP**					**PP**			
$\mathcal{RP}\ \cdot$	$(\exists^+\mathbf{Q}_1/\forall\mathbf{Q}_2)$	**RP**							**RP**	

Bibliography

[AAB+10] Scott Aaronson, Baris Aydinlioglu, Harry Buhrman, John M. Hitchcock, and Dieter van Melkebeek. A note on exponential circuit lower bounds from derandomizing Arthur-Merlin games. *Electronic Colloquium on Computational Complexity (ECCC)*, 17:174, 2010.

[Aar11] Scott Aaronson. Why philosophers should care about computational complexity. *Electronic Colloquium on Computational Complexity (ECCC)*, 18:108, 2011.

[AAW10] Eric Allender, Vikraman Arvind, and Fengming Wang. Uniform derandomization from pathetic lower bounds. *Electronic Colloquium on Computational Complexity (ECCC)*, 17:69, 2010.

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, 1 edition, April 2009.

[ABHH93] Eric Allender, Richard Beigel, Ulrich Hertrampf, and Steven Homer. Almost-everywhere complexity hierarchies for nondeterministic time. *Theor. Comput. Sci.*, 115:225–241, July 1993.

[AK01] V. Arvind and Johannes Köbler. On pseudorandomness and resource-bounded measure. *Theor. Comput. Sci.*, 255:205–221, March 2001.

[All08] Eric Allender. Cracks in the defenses: Scouting out approaches on circuit lower bounds. In Edward A. Hirsch, Alexander A. Razborov, Alexei L. Semenov, and Anatol Slissenko, editors, *Computer Science - Theory and Applications, Third International Computer Science Symposium in Russia, CSR 2008, Moscow, Russia, June 7-12, 2008, Proceedings*, volume 5010 of *Lecture Notes in Computer Science*, pages 3–10. Springer, 2008.

[AvM10] Scott Aaronson and Dieter van Melkebeek. A note on circuit lower bounds from derandomization. *Electronic Colloquium on Computational Complexity (ECCC)*, 17:105, 2010.

[Bab85] L Babai. Trading group theory for randomness. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, STOC '85, pages 421–429, New York, NY, USA, 1985. ACM.

[BF99] Harry Buhrman and Lance Fortnow. One-sided versus two-sided error in probabilistic computation. In *STACS*, pages 100–109, 1999.

[BFL91] László Babai, Lance Fortnow, and Carsten Lund. Non-deterministic exponential time has two-prover interactive protocols. *Computational Complexity*, 1:3–40, 1991.

[BFNW93] László Babai, Lance Fortnow, Noam Nisan, and Avi Wigderson. BPP has subexponential time simulations unless EXPTIME has publishable proofs. *Comput. Complex.*, 3(4):307–318, 1993.

[BFP03] Harry Buhrman, Lance Fortnow, and Aduri Pavan. Some results on Derandomization. In *STACS '03: Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science*, pages 212–222, London, UK, 2003. Springer-Verlag.

[BG81] Charles H. Bennett and John Gill. Relative to a random oracle A, $P^A \neq NP^A \neq coNP^A$ with probability 1. *SIAM J. Comput.*, 10(1):96–113, 1981.

[BHZ87] R. B. Boppana, J. Håstad, and S. Zachos. Does co-NP have short interactive proofs? *Inf. Process. Lett.*, 25:127–132, May 1987.

[BM84] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Comput.*, 13:850–864, November 1984.

[BM88] László Babai and Shlomo Moran. Arthur-Merlin Games: A randomized proof system, and a hierarchy of complexity classes. *J. Comput. Syst. Sci.*, 36(2):254–276, 1988.

[BW89] Ronald V. Book and Osamu Watanabe. A view of structural complexity theory. *Bulletin of the EATCS*, 39:122–138, 1989.

[CRT98] Andrea E. F. Clementi, José D. P. Rolim, and Luca Trevisan. Recent advances towards proving P = BPP. *Bulletin of the EATCS*, 64, 1998.

[DK00] Ding-Zhu Du and Ker-I Ko. *Theory of Computational Complexity.* Wiley-Interscience, January 2000.

[FGM$^+$89] Martin Fürer, Oded Goldreich, Yishay Mansour, Michael Sipser, and Stathis Zachos. On completeness and soundness in interactive proof systems. *Advances in Computing Research: Randomness and Computation (S. Micali, editor), JAI Press, Greenwich, CT*, 5:25–32, 1989.

[For97] Lance Fortnow. Counting Complexity. In *Complexity Theory Retrospective II*, pages 81–107. Springer-Verlag, 1997.

[For01] Lance Fortnow. Comparing notions of full Derandomization. In *CCC '01: Proceedings of the 16th Annual Conference on Computational Complexity*, page 28, Washington, DC, USA, 2001. IEEE Computer Society.

[For05] Lance Fortnow. Beyond np: the work and legacy of larry stockmeyer. In Harold N. Gabow and Ronald Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 120–127. ACM, 2005.

[For09] Lance Fortnow. The status of the P versus NP problem. *Commun. ACM*, 52:78–86, September 2009.

[FS07] Lance Fortnow and Rahul Santhanam. Time hierarchies: A survey. *Electronic Colloquium on Computational Complexity (ECCC)*, 14(004), 2007.

[FST05] Lance Fortnow, Rahul Santhanam, and Luca Trevisan. Hierarchies for semantic classes. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, STOC '05, pages 348–355, New York, NY, USA, 2005. ACM.

[GK09] Dan Gutfreund and Akinori Kawachi. Derandomizing Arthur-Merlin Games and Approximate Counting implies exponential-size lower bounds. *Electronic Colloquium on Computational Complexity (ECCC)*, 16:146, 2009.

[GKL88] Oded Goldreich, Hugo Krawczyk, and Michael Luby. On the existence of pseudorandom generators. In *CRYPTO*, pages 146–162, 1988.

[GL89] O. Goldreich and L. A. Levin. A hard-core predicate for all one-way functions. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, STOC '89, pages 25–32, New York, NY, USA, 1989. ACM.

[GL08] Xiaoyang Gu and Jack H. Lutz. Dimension characterizations of complexity classes. *Computational Complexity*, 17(4):459–474, 2008.

[GNW98] Oded Goldreich, Noam Nisan, and Avi Wigderson. On Yao's XOR-Lemma. Technical report, Electronic Colloquium on Computational Complexity, 1998.

[Gol00] Oded Goldreich. Pseudorandomness. In *ICALP*, pages 687–704, 2000.

[Gol08] Oded Goldreich. *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, 1 edition, April 2008.

[Gol10] Oded Goldreich. In a world of P=BPP. *Electronic Colloquium on Computational Complexity (ECCC)*, 17:135, 2010.

[Gol11] Oded Goldreich. Two comments on targeted canonical derandomizers. *Electronic Colloquium on Computational Complexity (ECCC)*, 18:47, 2011.

[GS86] S Goldwasser and M Sipser. Private coins versus public coins in interactive proof systems. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, STOC '86, pages 59–68, New York, NY, USA, 1986. ACM.

[GSTS07] Dan Gutfreund, Ronen Shaltiel, and Amnon Ta-Shma. If NP languages are hard on the worst-case, then it is easy to find their hard instances. *Computational Complexity*, 16(4):412–441, 2007.

[GV08] Dan Gutfreund and Salil P. Vadhan. Limitations of hardness vs. randomness under uniform reductions. *Electronic Colloquium on Computational Complexity (ECCC)*, 15(007), 2008.

[GW00] Oded Goldreich and Avi Wigderson. On pseudorandomness with respect to deterministic observers. *Electronic Colloquium on Computational Complexity (ECCC)*, 7(56), 2000.

[GW02] Oded Goldreich and Avi Wigderson. Derandomization that is rarely wrong from short advice that is typically good. *Electronic Colloquium on Computational Complexity (ECCC)*, (039), 2002.

[HILL99] Johan Håstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM J. Comput.*, 28(4):1364–1396, 1999.

[HLM05] J.M. Hitchcock, J.H. Lutz, and E. Mayordomo. The fractal geometry of complexity classes. *SIGACT News*, 36:24–38, 2005.

[IKW02] Russell Impagliazzo, Valentine Kabanets, and Avi Wigderson. In search of an easy witness: exponential time vs. probabilistic polynomial time. volume 65, pages 672–694, Orlando, FL, USA, 2002. Academic Press, Inc.

[IM09] Russell Impagliazzo and Philippe Moser. A zero-one law for RP and Derandomization of AM if NP is not small. *Inf. Comput.*, 207:787–792, July 2009.

[Imp95] Russell Impagliazzo. A personal view of average-case complexity. In *Structure in Complexity Theory Conference*, pages 134–147, 1995.

[Imp97] Russell Impagliazzo. Using hard problems to derandomize algorithms: An incomplete survey. In *RANDOM*, pages 165–173, 1997.

[Imp03] Russell Impagliazzo. Hardness as randomness: a survey of universal Derandomization. *CoRR*, cs.CC/0304040, 2003.

[Imp05] Russell Impagliazzo. Computational complexity since 1980. In *FSTTCS*, pages 19–47, 2005.

[Imp06] Russell Impagliazzo. Can every randomized algorithm be derandomized? In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, STOC '06, pages 373–374, New York, NY, USA, 2006. ACM.

[ISW99] Russell Impagliazzo, Ronen Shaltiel, and Avi Wigderson. Near-optimal conversion of hardness into pseudo-randomness. In *FOCS*, pages 181–190, 1999.

[ISW00] Russell Impagliazzo, Ronen Shaltiel, and Avi Wigderson. Extractors and pseudo-random generators with optimal seed length. *Electronic Colloquium on Computational Complexity (ECCC)*, 7(9), 2000.

[ISW06] Russell Impagliazzo, Ronen Shaltiel, and Avi Wigderson. Reducing the seed length in the Nisan-Wigderson Generator. *Combinatorica*, 26:647–681, December 2006.

[IW97] Russell Impagliazzo and Avi Wigderson. *P=BPP* unless *E* has sub-exponential circuits: Derandomizing the *XOR* lemma. In *In Proceedings of the 29th STOC*, pages 220–229. ACM Press, 1997.

[IW98] Russell Impagliazzo and Avi Wigderson. Randomness vs Time: De-Randomization under a Uniform Assumption. In *FOCS*, pages 734–743, 1998.

[IW01] Russell Impagliazzo and Avi Wigderson. Randomness vs Time: Derandomization under a Uniform Assumption. *J. Comput. Syst. Sci.*, 63:672–688, December 2001.

[Kab00] Valentine Kabanets. Easiness assumptions and hardness tests: Trading time for zero error. In *Proceedings of the 15th Annual IEEE Conference on Computational Complexity*, COCO '00, pages 150–, Washington, DC, USA, 2000. IEEE Computer Society.

[Kab02] Valentine Kabanets. Derandomization: a brief overview. *Bulletin of the EATCS*, 76:88–103, 2002.

[KI03] Valentine Kabanets and Russell Impagliazzo. Derandomizing polynomial identity tests means proving circuit lower bounds. In *STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 355–364, New York, NY, USA, 2003. ACM.

[KL80] Richard M. Karp and Richard J. Lipton. Some connections between nonuniform and uniform complexity classes. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, STOC '80, pages 302–309, New York, NY, USA, 1980. ACM.

[KvM99] Adam R. Klivans and Dieter van Melkebeek. Graph nonisomorphism has subexponential size proofs unless the polynomial-time hierarchy collapses. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, STOC '99, pages 659–667, New York, NY, USA, 1999. ACM.

[KvMS09] Jeff Kinne, Dieter van Melkebeek, and Ronen Shaltiel. Pseudorandom generators and typically-correct derandomization. In *APPROX-RANDOM*, pages 574–587, 2009.

[KvMS10] Jeff Kinne, Dieter van Melkebeek, and Ronen Shaltiel. Pseudorandom generators, typically-correct derandomization, and circuit lower bounds. *Electronic Colloquium on Computational Complexity (ECCC)*, 17:129, 2010.

[Lu00] Chi-Jen Lu. Derandomizing Arthur-Merlin Games under Uniform Assumptions. *Computational Complexity*, 10:247–259, 2000.

[Lut03] Jack H. Lutz. Dimension in complexity classes. *SIAM Journal on Computing*, 32(5):1236–1259, 2003.

[Mos03] Philippe Moser. BPP has effective dimension at most 1/2 unless BPP=EXP. *Electronic Colloquium on Computational Complexity (ECCC)*, 10(029), 2003.

[Mos08] Philippe Moser. Resource-bounded measure on probabilistic classes. *Inf. Process. Lett.*, 106(6):241–245, 2008.

[MV99] Peter Bro Miltersen and N. V. Vinodchandran. Derandomizing Arthur-Merlin Games Using Hitting Sets. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 71, Washington, DC, USA, 1999. IEEE Computer Society.

[Nis91] Noam Nisan. Pseudorandom bits for constant depth circuits. *Combinatorica*, 11(1):63–70, 1991.

[Nis96] Noam Nisan. Extracting randomness: How and why (a survey). In *IEEE Conference on Computational Complexity*, pages 44–58, 1996.

[NW94] Noam Nisan and Avi Wigderson. Hardness vs Randomness. *J. Comput. Syst. Sci.*, 49(2):149–167, 1994.

[Pap94] Christos H. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.

[PY12] Periklis A. Papakonstantinou and Guang Yang. A remark on one-wayness versus pseudorandomness. *Electronic Colloquium on Computational Complexity (ECCC)*, 19:5, 2012.

[PZ82] Christos H. Papadimitriou and Stathis Zachos. Two remarks on the power of counting. In *Proceedings of the 6th GI-Conference on Theoretical Computer Science*, pages 269–276, London, UK, 1982. Springer-Verlag.

[RR94] Alexander A. Razborov and Steven Rudich. Natural proofs. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, STOC '94, pages 204–213, New York, NY, USA, 1994. ACM.

[RR97] Alexander A. Razborov and Steven Rudich. Natural proofs. *J. Comput. Syst. Sci.*, 55(1):24–35, 1997.

[San07] Rahul Santhanam. Circuit lower bounds for Merlin-Arthur Classes. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, STOC '07, pages 275–283, New York, NY, USA, 2007. ACM.

[Sch89] Uwe Schöning. Probabilistic complexity classes and lowness. *J. Comput. Syst. Sci.*, 39(1):84–100, 1989.

[Sha10] Ronen Shaltiel. Typically-correct derandomization. *SIGACT News*, 41(2):57–72, 2010.

[Sip88] Michael Sipser. Expanders, randomness, or time versus space. *J. Comput. Syst. Sci.*, 36(3):379–383, 1988.

[Sip92] Michael Sipser. The history and status of the P versus NP question. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, STOC '92, pages 603–618, New York, NY, USA, 1992. ACM.

[Sto76] Larry J. Stockmeyer. The polynomial-time hierarchy. *Theor. Comput. Sci.*, 3(1):1–22, 1976.

[SU05] Ronen Shaltiel and Christopher Umans. Simple extractors for all min-entropies and a new pseudorandom generator. *J. ACM*, 52:172–216, March 2005.

[SU07a] Ronen Shaltiel and Christopher Umans. Low-end uniform hardness vs. randomness tradeoffs for AM. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, STOC '07, pages 430–439, New York, NY, USA, 2007. ACM.

[SU07b] Ronen Shaltiel and Christopher Umans. Pseudorandomness for approximate counting and sampling. *Computational Complexity*, 15(4):298–341, 2007.

[SU09] Ronen Shaltiel and Christopher Umans. Low-end uniform hardness versus randomness tradeoffs for AM. *SIAM J. Comput.*, 39(3):1006–1037, 2009.

[Tod91] Seinosuke Toda. PP is as hard as the polynomial-time hierarchy. *SIAM J. Comput.*, 20:865–877, October 1991.

[Tre99] Luca Trevisan. Construction of extractors using pseudo-random generators (extended abstract). In *STOC*, pages 141–148, 1999.

[Tre01] Luca Trevisan. Extractors and pseudorandom generators. *J. ACM*, 48(4):860–879, 2001.

[Tre06] Luca Trevisan. Pseudorandomness and combinatorial constructions. *CoRR*, abs/cs/0601100, 2006.

[TV07] Luca Trevisan and Salil P. Vadhan. Pseudorandomness and average-case complexity via uniform reductions. *Computational Complexity*, 16(4):331–364, 2007.

[Vad07] S. Vadhan. The unified theory of pseudorandomness: guest column. *SIGACT News*, 38:39–54, September 2007.

[Val92] Leslie G. Valiant. Why is boolean complexity theory difficult? In *Poceedings of the London Mathematical Society symposium on Boolean function complexity*, pages 84–94, New York, NY, USA, 1992. Cambridge University Press.

[VW97] Heribert Vollmer and Klaus W. Wagner. Measure one results in computational complexity theory. In *Advances in Algorithms, Languages, and Complexity*, pages 285–312, 1997.

[Wan97] Jie Wang. *Average-case computational complexity theory*, pages 295–328. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[Yao82] Andrew C. Yao. Theory and application of trapdoor functions. *SFCS '82: Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pages 80–91, 1982.

[Zac82] Stathis Zachos. Robustness of probabilistic computational complexity classes under definitional perturbations. *Information and Control*, (54), 1982.

[Zac86] Stathis Zachos. Probabilistic quantifiers, adversaries, and complexity classes: an overview. In *Proc. of the conference on Structure in complexity theory*, pages 383–400, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[Zac88] Stathis Zachos. Probabilistic quantifiers and games. *J. Comput. Syst. Sci.*, 36:433–451, June 1988.

[ZF87] Stathis Zachos and Martin Fürer. Probabilistic quantifiers vs. distrustful adversaries. In Kesav Nori, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 287 of *Lecture Notes in Computer Science*, pages 443–455. Springer Berlin / Heidelberg, 1987.

[ZH86] Stathis Zachos and Hans Heller. A decisive characterization of BPP. *Information and Control*, 69(1-3):125–135, 1986.

[Zuc90] D. Zuckerman. General weak random sources. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 534–543 vol.2, Washington, DC, USA, 1990. IEEE Computer Society.

[Zuc96] David Zuckerman. Simulating BPP using a general weak random source. *Algorithmica*, 16(4/5):367–391, 1996.