Randomness and non-uniformity

Felix Weninger

April 2006

Abstract

In the first part, we introduce randomized algorithms as a new notion of efficient algorithms for decision problems. We classify randomized algorithms according to their error probabilities, and define appropriate complexity classes. (**RP**, **coRP**, **ZPP**, **BPP**, **PP**). We discuss which classes are realistic proposals for design of probabilistic algorithms. We cover the implementation of randomized algorithms using different non-ideal random sources. We introduce the concept of derandomization and the "hardness vs. randomness" paradigm.
Second, we illustrate non-uniform complexity in terms of Boolean circuits and Turing machines that take advice. We demonstrate the power of non-uniform complexity classes. We show the relevance of non-uniform polynomial time for complexity theory, especially the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ question.

Contents

1 Randomized algorithms

1.1 Introduction

During the last three decades, usage of randomized algorithms has rapidly expanded and has significantly changed the notion of efficient computation and of algorithms themselves. It has turned out that for many decision and function problems, a randomized algorithm is either the simplest or the fastest algorithm available. Some fields of scientific computing, like numerical simulation, are inherently based on randomization.

Randomized algorithms are characterized informally by two facts: They use randomness to make decisions, and they may answer incorrectly. Analysis of randomized algorithms, especially their probability of error, is an important application for probability theory, and randomized algorithms are often also called *probabilistic*.

Naturally, since randomized computation is an enormous field, this paper is far from comprehensive. It focuses mainly on *time and error bounds for decision problems*. A lot of other aspects, like space bounds and randomized algorithms for function problems are covered in [1], [2] and [4].

1.2 Example: Polynomial identity testing

Next we introduce a simple randomized algorithm for *polynomial identity testing*: Given two polynomials p_1 and p_2 over some field $\mathbb{F}$, test whether they have identical coefficients (note that in the case of a finite field, this is not necessarily the same as testing whether they have the same value for every argument).

Of course, this is trivial if p_1 and p_2 are given explicitly. But as we will see later, there are some interesting cases where polynomials are given implicitly and evaluation is much less costly than determining their coefficients.

The description of the algorithm follows.

Algorithm 1 (Polynomial identity testing).

1. ***choose*** $S \subset \mathbb{F}$, $x \in S$
2. $y := p_1(x) - p_2(x)$
3. ***if*** $(y = 0)$ ***return*** "$p_1 \equiv p_2$" ***else return*** "$p_1 \not\equiv p_2$"

This simple algorithm already shows some key aspects of randomized algorithms. It it obvious that if $p_1 \equiv p_2$, the algorithm always answers correctly. However, if $p_1 \not\equiv p_2$, it can happen that a root of $p_1 - p_2$ is choosen as x in step 1 and therefore the algorithm returns "$p_1 \equiv p_2$". This event, however, occurs with a probability that is bounded by $\frac{d}{|S|}$, where d is the maximum degree of p_1 and p_2. This holds because $p_1 - p_2$ can have at most d roots; the worst case is that all roots are contained in S. The running

time of the algorithm is determined by the complexity of evaluating p_1 and p_2. In most cases this can be done in polynomial time, so the algorithm represents the class of *Monte Carlo algorithms* with polynomial running time and bounded error probability. There is also a significantly different kind of randomized algorithm (*Las Vegas* algorithm), which we will discuss later.

As shown in [1] and [2], an extension of this algorithm that works with *multivariate polynomials*, that is polynomials with several variables, is possible. This leads to an important application for algorithm 1: *determinants of symbolic matrices*, that is matrices with entries that are, in general, multivariate polynomials. Testing whether the determinant of such a matrix is identically zero is a common problem, for example in graph theory. Here we have implicitly given polynomials, which can be evaluated in polynomial time by substituting numeric values for the variables and performing Gaussian elimination on the resulting numeric matrix. It can be shown that the size of the matrix entries during this process is polynomial, whereas *symbolic* Gaussian elimination creates matrix entries of exponential size ([2]). In fact, no deterministic polynomial time algorithm is known for computing the determinant of a symbolic matrix, not even for testing whether it is zero or not. Note that the naive algorithm even takes $n!$ time.

1.3 Classification of randomized algorithms

In this section, we introduce some basic terms related to classification of randomized algorithms. Since we consider only decision problems (in terms of binary languages), we have the following cases:

- The algorithm decides correctly.
- An input x that belongs to the language is rejected by the algorithm (*false negative*).
- An input x that does not belong to the language is accepted by the algorithm (*false positive*).

Denote the probability of a false negative by p_1 and the probability of a false positive with p_2. If either $p_1 = 0$ or $p_2 = 0$, the algorithm is called a *one-sided error* algorithm: If $p_1 = 0$, a "no" answer is a definitive one; if $p_2 = 0$, a "yes" answer is. Otherwise, it is called a *two-sided error* algorithm.

It is worth noting that deciding a language by "coin toss" is a two-sided error algorithm with $p_1 = p_2 = \frac{1}{2}$. This "pathological example" should be kept in mind when considering error bounds for two-sided error algorithms (on the other hand, we will see that a one-sided error algorithm with any error probability bounded away from 1 by a fixed amount is acceptable!)

1.4 Nondeterminism and randomness

Nondeterminism, as defined in terms of nondeterministic Turing machines, is intuitively related to randomized computation: A nondeterministic Turing machine can be considered as a machine that chooses from possible computation steps uniformly at random. As a consequence, we can view results of nondeterministic computation as *events* in the sense of probability theory.

We next introduce a concept which makes probabilistic analysis of nondeterministic Turing machines easier: *standardized nondeterministic Turing machines* (SNDTM). These are nondeterministic Turing machines that, for every configuration, have exactly two possible alternatives. Obviously such a machine has a computation tree that is a full binary tree of depth $2^{-f(|x|)}$, where $f(|x|)$ is the time bound of the machine, depending on the size of the input x.

With the concepts from the previous section, we now analyze machines deciding languages in the class **NP**. From the definition of **NP**, we conclude that if one computation of such a machine is chosen at random, the probability of a false positive is zero, whereas the fact that only one accepting computation is required for an input that belongs to the language implies that the probability of a false negative can be as high as $1 - 2^{-p(|x|)}$, where $p(.)$ is the polynomial time bound of the machine. Clearly for practical purposes this error probability is not acceptable. Therefore we are motivated to look for a subset of **NP** such that the probability of a false negative is "decently" bounded away from 1.

1.5 Randomized complexity classes

1.5.1 The class RP

The considerations at the end of the previous section lead to the definition of the complexity class **RP**, as follows.

Definition 2. *A language L is in* **RP** *if there exists a SNDTM M deciding L and a polynomial p, such that for every input x, M halts after $p(|x|)$ steps and the following holds:*

1. $x \in L \Rightarrow \mathbf{prob}[M(x) = 0] \leq \frac{1}{2}$ *(false negative)*
2. $x \notin L \Rightarrow \mathbf{prob}[M(x) = 1] = 0$ *(false positive)*

It should be noted that the constant $\frac{1}{2}$ in Definition 2 is arbitrary. Any constant strictly between 0 and 1 results in the same complexity class. To see why, let $\mathbf{RP}'$ be defined as above, but with a probability of $\frac{2}{3}$ for a false negative. Let us assume that $L \in \mathbf{RP}'$ and is decided by a Turing machine M'. We build a Turing machine M that executes the following procedure (note that Turing machines can simulate other Turing machines):

1. Invoke $M'(x)$ three times.

2. Accept x iff M' has accepted x at least once.

We see that M fulfills the requirements of Definition 2: For any $x \in L$, $\mathbf{prob}[M(x) = 0] \leq \left(\frac{2}{3}\right)^3 = \frac{8}{27} \leq \frac{1}{2}$ while M still rejects any $x \notin L$. We conclude that $\mathbf{RP}' = \mathbf{RP}$.

From the above construction, we also see that generally the probability of false negatives exponentially reduces in the number of executions of an **RP** algorithm.

Similar constructions show that the probability of a false negative does not even have to be constant, but can be inverse polynomial in the size of the input. Note that this is still fundamentally different from the definition of **NP**, where an *exponentially small* fraction of accepting computations is allowed.

1.5.2 The class coRP

Obviously, there is some "asymmetry" in the definition of **RP**. It is not clear at all whether **RP** is closed under complement - in fact, this is an open question and motivation for the definition of the class **coRP** for languages whose complement is in **RP**:

Definition 3. *A language L is in* **coRP** *if there exists a SNDTM M deciding L and a polynomial p, such that for every input x, M halts after $p(|x|)$ steps and the following holds:*

1. $x \in L \Rightarrow \mathbf{prob}[M(x) = 0] = 0$ *(false negative)*

2. $x \notin L \Rightarrow \mathbf{prob}[M(x) = 1] \leq \frac{1}{2}$ *(false positive)*

We defined **RP** as a subset of **NP**. Obviously, **coRP** is a subset of **coNP**.

An important example of a **coRP** algorithm is the famous Miller-Rabin primality test with a probability of $\frac{1}{4}$ for a false positive.

1.5.3 The class ZPP

We now consider the set of languages $\mathbf{RP} \cap \mathbf{coRP}$. A language L in this set has two probabilistic polynomial algorithms: Denote with A_1 the algorithm that does not produce false positives, and with A_2 the one that does not produce false negatives. If we run both algorithms in parallel for k times, we get a definitive result with probability $1 - 2^{-k}$. To see why, let us assume that an input x does not belong to L. Now, since A_1 by definition always answers "$x \notin L$", the only event where we do not get a definitive result is that A_2 keeps producing false positives. It is evident from the construction in the section 1.5.1 that this event has probability 2^{-k}.

A somewhat archetypical problem in $\mathbf{RP} \cap \mathbf{coRP}$ is primality testing (note that the **RP** primality test, presented by Adleman and Huang in 1987, is a quite complicated one, in contrast to the relatively simple Miller-Rabin test).

The existence of such problems is motivation for the definition of the complexity class **ZPP** (for "zero probability of error polynomial time"), as follows:

Definition 4. $\mathbf{ZPP} := \mathbf{RP} \cap \mathbf{coRP}$

The algorithm outlined above is called a *Las Vegas Algorithm*: It has zero probability of error, but its running time is not bounded *a priori*. However, it can be shown that the average running time is polynomial, which makes **ZPP** a class quite close to **P**.

1.5.4 The class BPP

In section 1.3, we introduced two-sided error algorithms. The complexity classes we have seen so far all concern one-sided error algorithms. We now define a complexity class for algorithms with *bounded probability* of error and *polynomial* running time (**BPP**):

Definition 5. *A language L is in* **BPP** *if there exists a SNDTM M deciding L and a polynomial p, such that for every input x, M halts after $p(|x|)$ steps and the following holds:*

$$\mathbf{prob}[M(x) = \chi_L(x)] \geq \frac{3}{4}$$

where $\chi_L(x)$ is the characteristic function *of L, defined as*

$$\chi_L(x) := \begin{cases} 1 & x \in L \\ 0 & \textit{otherwise} \end{cases}$$

Again, the constant in the above definition is arbitrary and can be replaced by any constant strictly between 1/2 and 1, or by $\frac{1}{2} + q(|x|)^{-1}$, with $q(.)$ being a suitable polynomial.

Note that it is an open question whether $\mathbf{BPP} \subseteq \mathbf{NP}$, or whether $\mathbf{NP} \subseteq \mathbf{BPP}$ - we do not get it from the definition like in the previous sections. The reason is that on the one hand, the bound on the probability of a false negative in the definition of **BPP** is stronger than in **NP**; on the other hand, the definition of **NP** requires a zero probability of false positives, which is only bounded in the case of a **BPP** algorithm.

However, the latter inclusion is considered unlikely because it would imply that every **NP**-complete problem has an efficient probabilistic algorithm. In contrast, the first inclusion is believed to be true (see 1.7).

1.5.5 The class PP

We now have a look at "majority problems". An important example is MAJSAT, a variant of the boolean satisfiability problem: Given a boolean formula φ with n variables, is it true that the majority of truth assignments satisfy it?

The obvious Turing machine which nondeterministically chooses truth assignments and tests whether they satisfy the formula has an acceptance probability as low as $\frac{1}{2} + 2^{-n}$ for a "yes" instance, because there might be only $2^{n-1} + 1$ satisfying truth assignments. Therefore **BPP** seems to be an inappropriate complexity class for this problem. Note that this problem is also unlikely to be in **NP**, because the obvious certificate for a "yes" instance ($2^{n-1} + 1$ satisfying assignments) is far from succinct!

We want a suitable complexity class for such "majority problems". In a first step, we define a class **PP**' that allows arbitrarily small differences between the number of accepting and rejecting computations:

Definition 6. *A language L is in* **PP**' *if there exists a SNDTM M deciding L and a polynomial p, such that for every input x, M halts after $p(|x|)$ steps and the following holds:*

$$\mathbf{prob}[M(x) = \chi_L(x)] > \frac{1}{2}$$

Note that this definition still does not capture the difficulty of MAJSAT because a "no" instance might have exactly half, i.e. 2^{n-1}, satisfying assignments. On the other hand, we cannot replace the "$>$" in the definition by a "$\geq$" since that would define a meaningless class (remember the statement about a "coin-toss" decision in the introduction). By a construction described in [4], however, it is possible to change "$>$" to "$\geq$" *for "no" instances only*, without changing the complexity class.

Definition 7. *A language L is in* **PP** *if there exists a SNDTM M deciding L and a polynomial p, such that for every input x, M halts after $p(|x|)$ steps and the following holds:*

1. $x \in L \Rightarrow \mathbf{prob}[M(x) = 1] > \frac{1}{2}$
2. $x \notin L \Rightarrow \mathbf{prob}[M(x) = 0] \geq \frac{1}{2}$

Theorem 8. $\mathbf{PP}' = \mathbf{PP}$.

This is perhaps the weakest possible definition for a probabilistic algorithm: "**p**robabilistically **p**olynomial time". It is argued that this class should rather be called "Majority-**P**" (in analogy to $\#\mathbf{P}$ or $\oplus\mathbf{P}$). The reason is that **PP** is *not* a realistic proposal of probabilistic algorithms, as we will see in the next section. First, we establish another result that outlines the power of **PP**:

Theorem 9. $\mathbf{NP} \subseteq \mathbf{PP}$.

Proof. Let $L \in \mathbf{NP}$ decided by a SNDTM N. We build a machine N' such that its computation tree consists of a root node which is connected to the root of N's computation tree and another computation tree of the same depth which has only accepting computations. Case distinction on membership of input x in L:

1. $x \in L$: Since N has, by definition of **NP**, at least one accepting computation, N' has at least one more accepting than rejecting computations.

2. $x \notin L$: Since N has, by definition of **NP**, only rejecting computations, N' has exactly half accepting computations.

By definition 7, $L \in \mathbf{PP}$. The theorem follows. □

1.5.6 Efficient experimentation

Although the definitions of the complexity classes presented so far seem quite similar, there is an important difference between them, concerning the design of concrete algorithms for problems in these classes. A most central question in analysis of probabilistic algorithms is whether they allow *efficient experimentation*: How often has the algorithm to be repeated so that the result can be considered correct with reasonable confidence?

For the class **RP**, this question is easy to answer: If an algorithm for a problem in this class is repeated n times, then if at least one "no" answer occurs, this is the correct one; otherwise the probability of n "yes" answers being incorrect is at most 2^{-n}. The same holds (vice versa) for **coRP**.

However, for two-sided error algorithms, this question is trickier since neither answer is surely correct. The obvious solution is to take the "majority vote" of n runs. The task is now to estimate the error probability of this procedure.

Here a lemma from probability theory comes most handy, which is presented in a version suited to randomized algorithms:

Lemma 10 (The Chernov bound for probabilistic algorithms). *Let A be a two-sided error algorithm that anwers correctly with probability $\frac{1}{2}+\epsilon$. Let Y denote the number of correct answers after n independent executions of A: Y is a* binomial random variable. *Then, for any $0 < \epsilon < \frac{1}{2}$,*

$$\mathbf{prob}\left[Y \leq \frac{n}{2}\right] \leq e^{-\frac{\epsilon^2 n}{6}}.$$

By choosing $n = \frac{c}{\epsilon^2}$, a suitable c can make this probability arbitrarily small.

For **BPP**, ϵ is at least inverse polynomial or constant. For **PP**, however, ϵ can be arbitrarily (i.e. even expontentially) small. If we equate efficient experimentation with polynomial running time, we can immediately conclude:

Corollary 11. BPP *can be efficiently experimented.* **PP** *cannot.*

1.6 Random sources

Practical implementation of algorithms closely depends on availability of random sources, i.e. sources of random bit strings $x_1 \dots x_n$. The most commonly used random sources in modern computer systems can be classified into *hardware random sources* and *pseudorandom number generators.*

Whether a random source can be used to drive a randomized algorithm depends mainly on two properties. A *perfect random source* is characterized by *independency* (i.e. the value of bit x_i is not influenced by the values of $x_1 \dots x_{i-1}$) and *fairness* (i.e. $\mathbf{prob}[x_i = 1] = \frac{1}{2}$). It turns out that the "difficult" property is independency ([2]).

1.6.1 Hardware random sources

Randomness can be found in many physical processes, such as nuclear decay and lava lamps (the latter being more comfortable to the average user...) Digital measurement of these processes (Geiger counters, digital imaging) is a common method to generate "random" bit strings. Another approach, which is often found in end-user cryptography software, is to use randomness inherent in computer systems (e.g. swap files or interrupts from I/O devices). The drawback is that those strings are not random at all: clearly the property that the described sources lack is independency. The definition of *slightly* random sources deals with this fact:

Definition 12 (δ-random source). *Let $0 < \delta \leq \frac{1}{2}$, and let $p : \{0,1\}^* \to [\delta, 1-\delta]$ be an arbitrary function. A $\delta-$random source S_p is a sequence of bits $x_1 \dots x_n$ such that, for $1 \leq i \leq n$,*

$$\mathbf{prob}[x_i = 1] = p(x_1 \dots x_{i-1})$$

The key aspect of this definition is that *arbitrary dependencies on previous outcomes* are allowed. As a consequence, slightly random sources cannot be used to drive randomized algorithms directly. The reason is that an arbitrarily dependent random source could be an *adversary* which knows the randomized algorithm and outputs "random" bits exactly such that the algorithm makes choices that lead to an incorrect result!

To gain further insight into the impact of non-ideal random sources on randomized computability, we informally define the complexity class $\delta -$ **BPP** (see [2] for a formal definition):

Definition 13. *A language is in* $\delta - \mathbf{BPP}$ *if it is decided by a SNDTM driven by a* δ*-random source, such that the probability of a false answer is less than* $\frac{1}{4}$.

To see what is meant by "driven by a δ-random source", consider the following example:

$$\delta = 0.1, p = \{(\lambda, 0.4), (0, 0.3), (1, 0.8), \dots\}$$

The first bit is 1 with probability 0.4 and 0 with probability 0.6. If the first bit is 0, then the second bit will be 1 with probability 0.3 and 0 with probability 0.7. Oterwise, the second bit will be 1 with probability 0.8 and 0 with probability 0.2, and so on. The Turing machine uses those random bits to make its choices.

Theorem 14. $\delta - \mathbf{BPP} = \mathbf{BPP}$.

Proof. Show that slightly random sources can be used to simulate any randomized algorithm with cubic loss of efficiency (see [2]). □

Perhaps it could be argued that this is more a theoretical result because a cubic loss of efficiency is unacceptable for most applications.

1.6.2 Pseudorandom number generators

Pseudorandom number generators (PRNG) are, informally, deterministic algorithms that turn a "short" seed (start value) into a "long" sequence of random bits. A quality measure for PRNGs is how close the distribution on the output sequence is to a real uniform distribution. Since the 1950's, the most commonly used PRNG algorithms are linear congruential generators (LCG) which are blazingly fast, use little memory, but have some severe statistical flaws. For example, it is known that if they are used to choose points in n-dimensional space, those points lie on discrete hyperplanes.

A stronger notion of a PRNG is closely related to cryptography. In this field, the intention is that any attacker with limited computational resources will fail to predict the outcome of the PRNG. Here so-called "one-way functions" are very useful: These functions are bijections that are easy to compute, but arguably hard to invert. An example is the *discrete logarithm*: Given $a, b \in \mathbb{Z}$ and $p \in \mathbb{P}$, $a^b \mod p$ is fast to compute (using successive squaring), but there is no known polynomial-time algorithm that, given p, a and $a^b \mod p$, computes b (the discrete equivalent of the logarithm to base a).

The problem is that the existence of one-way functions is based on strong complexity assumptions (mainly that there is no polynomial-time algorithm for the discrete logarithm problem) and can therefore only be conjectured.

1.7 Derandomization

Since the dependency on random sources seems to be quite an issue in implementing randomized algorithms, the concept of *derandomization*, that is, the removal of any use of random sources, might be promising. There is a natural way of derandomizing a **BPP** algorithm: Iterate over all possible random strings, then take the majority of the outcomes. This approach is clearly a deterministic algorithm and it is always correct, by the definition of **BPP**. Of course, this takes exponential running time (note that even problems in **PP** can be addressed in that way). In contrast, the belief is that a non-trivial derandomization of **BPP** is possible, meaning that we can take a subset of the random strings such that the majority of correct answers is preserved.

There is an interesting connection to PRNGs: Consider a PRNG that turns a seed of size $m(n) \ll n$ into a pseudorandom sequence of length n. If the output of the PRNG looks uniform to any polynomial-size circuit (see section 2.3 on circuit complexity; polynomial-size circuits are more powerful than polynomial-time algorithms), then, to any algorithm in **BPP**, there is no more randomness in the set of all possible random strings than in the set that is generated by the PRNG. This means that iterating over all seeds of the PRNG is sufficient for a derandomization of a **BPP** algorithm.

Theorem 15. *If there exists a PRNG G that turns a seed of size $m(n) \ll n$ into a pseudorandom sequence of length n, then* **BPP** *can be derandomized in DTIME(time$(G) \cdot 2^m$).*

Proof. See [3]. □

Early derandomization approaches (Yao 1982) used cryptographically secure PRNGs, which lead to a subexponential derandomization under the assumption that one-way functions exist.

In 1994, Nisan and Wigderson presented a completely different PRNG (NWPRNG) that loosened from the cryptographic background and is based on general hard-to-compute functions (not just in one way). Superpolynomial running time of the PRNG is normally not acceptable in cryptographic applications, but okay for derandomization purposes. The advantage of the NWPRNG is that, informally speaking, it is "modular" in the sense that you can "plug in" different hardness assumptions and end up with derandomizations in different running times. Using their own PRNG, Nisan and Wigderson showed a subexponential derandomization of **BPP**.

A complete derandomization of **BPP** (that is, a polynomial-time one) was established by Impagliazzo and Wigderson in 1997, using a somewhat strong hardness assumption in terms of boolean circuits.

Theorem 16. *If there is a language $L \in \mathbf{E} := \bigcup_c DTIME(2^{cn})$ which, for almost all inputs of size n, requires Boolean circuits of size $2^{\epsilon n}$ for some $\epsilon > 0$, then* **BPP** $=$ **P**.

The problem is that the proof of lower bounds for circuit sizes seems to be one of the hardest subjects in theoretical computer science.

However, the above results lead to the "hardness versus randomness" paradigm, perhaps one of the most interesting results of modern theoretical computer science, as formulated in [3]:

Either there exist provably hard functions or randomness extends the class of efficient algorithms.

It follows from the implication in theorem 16 that not both of these assumptions can be true. Since most experts conjecture the existence of provably hard functions, it is believed that $\mathbf{P} = \mathbf{BPP}$, which implies that all complexity classes considered so far, with the exception of **PP**, are equal from a set-theoretic point of view - but that does not necessarily mean that the concepts of probabilistic computation are useless: First, **P** and **BPP** refer only to decision problems, which are only one field of application of randomized algorithms. Second, there are problems like primality testing that do have a deterministic polynomial-time algorithm (PRIMES: $O(\log^{11} n)$), but also a significantly faster (PRIMES: $O(\log^3 n)$) randomized polynomial-time algorithm which is preferable in almost every practical respect.

2 Non-uniformity

2.1 Turing machines with advice

Randomized polynomial time can be illustrated by two-string Turing machines which take an additional random string of polynomial size as input. A computation model which is intuitively a generalization of this concept is *computation with advice*. By that we mean informally that a Turing machine is allowed to read a string which not only tells them how to choose from alternative computations, but actively aids their decision in arbitrary ways. Clearly it is pointless to allow a different advice string for every input, as that advice string could just be the value of the characteristic function of the language to decide. So a natural restriction is to allow exactly *one* advice string for *all* inputs of a certain length. We will see later that randomized computation is indeed a special case of advised computation as specified above.

Next we define the family of complexity classes $\mathbf{P}/f(n)$ that capture advised computation.

Definition 17. *A language L is in $\mathbf{P}/f(n)$ if there exists a polynomial-time two-input deterministic Turing machine M, a complexity function $f(n)$ and a sequence (a_n) of advice strings such that:*

- $\forall n : |a_n| \leq f(n)$ *(advice is space-bounded)*

- $\forall x \in \{0,1\}^n : M(a_n, x) = \chi_L(x)$ *(M decides L using a_n as advice)*

We see that advised computation is *non-uniform* in the sense that the construction of the advice strings is not specified; in fact, the sequence (a_n) does not even have to be computable!

2.2 Non-uniform polynomial time: the class $\mathbf{P}/poly$

Note that it makes no sense to allow advice of superpolynomial size in the above definition, since such advice could never be read in polynomial time. Furthermore, in a machine model with random access to the string, an advice string of exponential length could be a lookup table for the language to decide. These considerations lead to the definition of $\mathbf{P}/poly$: the class of problems decided by Turing machines that take advice of *polynomial length*.

Definition 18. $\mathbf{P}/poly := \bigcup_k \mathbf{P}/n^k$

It is clear that $\mathbf{P} \subseteq \mathbf{P}/poly$, since $\mathbf{P}$ can be considered to be the class of problems decided by Turing machines with empty advice.

2.3 Circuit complexity

Complexity classes are usually defined in terms of Turing machines. Another, interesting view of complexity uses *Boolean circuits*.

Definition 19. *A* Boolean circuit *is a dag (V, E) with a labelling function $s : V \to \{\neg, \vee, \wedge, x_1, \ldots, x_n, 0, 1, out\}$, such that*

- $s(v) = \neg \Rightarrow deg^+(v) = 1$ *(NOT gate)*
- $s(v) = \vee$ *or* $s(v) = \wedge \Rightarrow deg^+(v) = 2$ *(AND/OR gates)*
- $s(v) = x_1, \ldots, x_n, 0, 1 \Rightarrow deg^+(v) = 0$ *(input)*
- $s(v) = out \Rightarrow deg^-(v) = 0,\ deg^+(v) = 1$ *(output)*
- *The labels $x_1, \ldots, x_n, out$ are used exactly once.*

A boolean circuit C with inputs $x_1 \ldots x_n$ is usually more succinct than an equivalent boolean expression $\varphi(x_1 \ldots x_n)$. The reason is that in the above definition, we do not limit the indegree of gates, which makes it possible to connect the output of a gate (which corresponds to an expression like "$x_3 \vee x_4$") to the input of several other gates, so that expressions that occur more than once in a Boolean formula need to be realized only once in the corresponding circuit ("shared expressions").

Using Boolean circuits, the complexity of deciding a language L can be formulated as follows: Given an input string x in binary encoding, what is

the *size* (number of gates) of a Boolean circuit which has the bits of x as input and $\chi_L(x)$ as output?

Like **P** is often considered the class of problems with efficient algorithms, we might say that a language has succinct circuits if their size is polynomially bounded.

Definition 20. *A language $L \subseteq \{0,1\}^*$ has polynomial circuits if there exists a sequence (C_n) of Boolean circuits and a polynomial p such that:*

- $\forall n : size(C_n) \leq p(n)$
- *C_n has n inputs, and the output of C_n is $\chi_L(x)$ $\forall x \in \{0,1\}^n$.*

Polynomial circuits as defined above are again non-uniform since there need not be an explicit description of the circuits!

The definitions of languages decided by polynomial circuits and languages decided by Turing machines with advice of polynomial length resemble each other closely. In fact, they denote the same class of languages:

Theorem 21. *A language L has polynomial circuits if and only if $L \in \mathbf{P}/poly$.*

Proof.

- "$\Rightarrow$": Let L be a language that has a family C_n of polynomial circuits. We can build a Turing machine that uses binary encodings of C_n as advice strings a_n. Since C_n is polynomially bounded, so is a_n; it is a standard result in complexity theory that the value of a Boolean circuit can be computed in polynomial time (CIRCUIT VALUE is a **P**-complete problem). It follows that $L \in \mathbf{P}/poly$.
- "$\Leftarrow$": Let $L \in \mathbf{P}/poly$. Then there exists a polynomial-time Turing machine M with polynomial advice strings a_n. It is a standard result that for some n, we can build a polynomial-time Turing machine M' that behaves like M with a_n "hard-wired" into it. We now encode the *computation matrix* of M', which represents the input/output string over time, as a Boolean circuit whose input gates represent the initial string and whose output gate is a Boolean value indicating whether the machine halts in an accepting state. This circuit has polynomial size: Since M' is polynomially time-bounded in n, so are its state space and alphabet. It follows that the matrix entries (consisting of a character, a state and a Boolean indicator for the position of the head) have logarithmic size. Any matrix entry $m_{i,j}$ depends only on the entries $m_{i-1,j-1}$, $m_{i-1,j}$ and $m_{i-1,j+1}$ since in one time step the head can move at most one character right or left. Therefore the computation matrix can be represented by a polynomial number of circuits with a logarithmic number of input gates. It is left to show that any Boolean

circuit with k input gates has size at most $k2^k$. See [4], chapter 2, for a detailed proof.

□

2.4 The power of $\mathbf{P}/poly$

The motivation for the introduction of $\mathbf{P}/poly$ was to generalize randomized computation to obtain a higher "upper bound on efficient computation". The following result states that indeed non-uniform polynomial time is a generalization of randomized polynomial time:

Theorem 22 (Adleman's theorem).

$$\mathbf{BPP} \subseteq \mathbf{P}/poly.$$

Proof. The basic proof idea is indeed to use random strings r as advice strings. We have to show that for every n, there exists *one* random string that makes a Turing machine decide correctly on *all* inputs of length n.
Let $L \in \mathbf{BPP}$ be decided by a Turing machine M that is time-bounded by $p(|x|)$. Let

$$\text{bad}(x) := \{r \in \{0,1\}^{p(|x|)} : M(x,r) \neq \chi_L(x)\}$$

In other words, bad(x) denotes the set of random strings that make M decide incorrectly on input x. We can assume without loss of generality that M has an error probability of at most $\frac{1}{3^n}$: If M has a higher error probability, we can use the amplification described previously to obtain a polynomial-time machine with the desired error probability. It follows that $\mathbf{prob}_{r\in\{0,1\}^{p(|x|)}}[r \in \text{bad}(x)] = \frac{1}{3^n}$ (in the following statement, the probabilities are implicitly taken over all possible random strings). We have:

$$\mathbf{prob}\left[r \in \bigcup_{x\in\{0,1\}^n} \text{bad}(x)\right] \leq \sum_{x\in\{0,1\}^n} \mathbf{prob}\left[r \in \text{bad}(x)\right] = \frac{2^n}{3^n} < 1$$

The first expression denotes the probability that one random string is "bad" for *any* input x. The first inequality comes from the *union bound*: There might be strings which are bad for more than one x; these occur only once in the union of all bad(x).
The above inequality, read from left to right, states that there are more random strings than there are random strings that are "bad" for *any* input. It follows that there must exist at least one random string that is "good" for *all* inputs of length n. The desired conclusion follows. □

The above proof is an example of the *probabilistic proof technique*, a non-constructive proof of existence.

The inherent non-uniformity in $\mathbf{P}/poly$ actually makes this class (and therefore polynomial circuits) far more powerful than polynomial-time algorithms:

Theorem 23. $\mathbf{P}/poly$ *contains non-recursive languages.*

Proof. The theorem is obtained from the following two claims:

1. Every *unary language* $L \subseteq \{1\}^*$ is in $\mathbf{P}/poly$.
   Proof: Since in any unary language there is at most one string of length n, we can simply define the characteristic function of L as advice strings:
   $$a_n := \begin{cases} 1 & 1^n \in L \\ 0 & \text{otherwise} \end{cases}$$
2. There are non-recursive unary languages.
   Proof: Given any non-recursive $L \subseteq \{0,1\}^*$, define
   $$U := \{1^n \mid \text{binary expansion of } n \text{ is in } L\}$$
   U is non-recursive since there is an (exponential) reduction from L to U.

□

We now attempt to rid $\mathbf{P}/poly$ of this absurdness and define languages with *uniform* circuits, i.e. languages that have circuits that can be explicitly described and efficiently constructed:

Definition 24. *A language has uniform polynomial circuits if it can be decided by a sequence of polynomially sized circuits* (C_n) *(as in definition 20) and there exists a polynomially time-bounded Turing machine which, for all* n*, on input* 1^n *outputs a description of* C_n*.*

The following result proposes that this restriction has a somewhat extreme consequence:

Theorem 25. *A language* $L \subseteq \{0,1\}^*$ *has uniform polynomial circuits if and only if* $L \in \mathbf{P}$*.*

Proof.

- "$\Rightarrow$": If L has polynomial circuits, then one can construct a Turing machine that simulates the machine constructing C_n and then evaluates C_n on the input. It is clear that this machine has polynomial running time, therefore $L \in \mathbf{P}$.
- "$\Leftarrow$": Let $L \in \mathbf{P}$ be decided by a machine M. For any n, we can build a polynomial-time Turing machine that performs the construction of a circuit C_n which encodes the computation of M (as outlined in the proof of theorem 21).

□

Note that from a practical point of view, this theorem basically states that problems with efficient algorithms can be solved by succinct and easily constructible hardware.

2.5 On $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$

We have seen that while the inherent non-uniformity in $\mathbf{P}/poly$ leads to some absurdities, the introduction of uniformity reduces this class to $\mathbf{P}$. Besides the fact that $\mathbf{P}/poly$ cannot serve as a realistic computational model, it is of great theoretical interest. In fact, the $\mathbf{P}$ vs. $\mathbf{NP}$ question was a motivation for the development of the concept of circuit complexity: If we could prove that $\mathbf{NP}$ has no polynomial circuits, it would immediately follow that $\mathbf{P} \neq \mathbf{NP}$.

In this section, we present two theorems related to the $\mathbf{P}$ vs. $\mathbf{NP}$ question. First have to introduce the definition of *density* of languages.

Definition 26. *A language $L \subseteq \{0,1\}^*$ is* sparse *if there exists a polynomial p such that*

$$\forall n : |L \cap \{0,1\}^n| \leq p(n)$$

Otherwise, L is dense.

Example 27. *Every unary language is sparse (take $p(.) \equiv 1$). Every known* $\mathbf{NP}$*-complete language is dense.*

Lemma 28. *Every sparse language is in* $\mathbf{P}/poly$.

Proof. Let L be a sparse language. Define as advice string a_n a concatenation of all strings of length n in L. By assumption, a_n has polynomial size $(n \cdot p(n))$ and L can therefore be decided in polynomial time, by scanning a_n appropriately. □

Theorem 29 (Fortune). $\mathbf{P} = \mathbf{NP}$ *iff every $L \in \mathbf{NP}$ Karp-reduces to a sparse language.*

Proof. See [4], chapter 8. □

Recall that Karp reductions (also called polynomial-time transformations) are the kind of reductions commonly used in connection with $\mathbf{NP}$-completeness. If every $L \in \mathbf{NP}$ Karp-reduced to a sparse language, there would be a sparse $\mathbf{NP}$-complete language, which is strongly believed not to be the case (example 27).

However, one could argue that things could be different if one allowed more powerful reductions, for example *Cook reductions*.

Definition 30. *A language L* Cook-reduces *to L' if L can be decided in polynomial time, using polynomially many queries of the type "$x \in L'$?" to an oracle for L'.*

It is evident that a Karp reduction is a special Cook reduction, where there is only *one* oracle query allowed, and only *at the end*. In contrast, in a Cook reduction the outcome of oracle queries can be used in the further course of the reduction.

Theorem 31 (Karp and Lipton). $\mathbf{NP} \subseteq \mathbf{P}/poly$ *if and only if every* $L \in \mathbf{NP}$ *Cook-reduces to a sparse language.*

Proof. See [4], chapter 8. □

In terms of circuits, both theorems are quite parallel: Theorem 29 refers to the belief that **NP** has no uniform polynomial circuits (that is, $\mathbf{P} \neq \mathbf{NP}$); theorem 31 refers to the (stronger) conjecture that **NP** has no polynomial circuits, *uniform or not*.

References

[1] Rajeev Motwani and Prabhakar Raghavan: Randomized Algorithms. Cambridge University Press, 1995.

[2] Christos H. Papadimitriou: Computational Complexity. Addison Wesley, 1994.

[3] Avi Wigderson and Irit Dinur: Derandomizing **BPP** and Pseudorandomness - A Survey (2002).

[4] Oded Goldreich: Introduction to Complexity Theory - Lecture Notes (1999).