Theory of Randomized Computation
A Survey for CSG714

Daria Antonova and Daniel Kunkle
CCIS, Northeastern University, Boston

April 22, 2005

Abstract

Randomized algorithms are now common in nearly all areas of computer science. The ability to make random choices has been shown to both simplify algorithms and improve their running times. The success of randomized algorithms in efficiently solving problems for which no known deterministic algorithm exists has fueled theoretical research in the area of probabilistic computation. We present the foundations and some current research related to the theory of randomized computation. Particularly, we: (1) explore the effects of randomness on computational power; (2) present a hierarchy of randomized complexity classes in relation to the traditional complexity classes; (3) discuss probabilistic models of computation and (4) summarize current research in the derandomization of randomized algorithms and complexity classes.

1 Introduction

The use of randomness in computation is now a common practice. The primary reasons for this is that it allows to simplify the design of an algorithm as well as reduce its running time. The usefulness of randomness in practical settings has fueled a large body of theoretical research dealing with randomized computations. In this survey, we present some of the foundations and current research in this area.

Section 2 presents a randomized algorithm for an important problem for which there is no known deterministic polynomial-time solution: polynomial identity testing.

Section 3 compares the computational power of machines with access to random numbers to ordinary machines. In particular, it is shown that

seemingly small changes in assumptions about what kind of randomness is used can change the computational power of the machine.

Section 4 defines randomized complexity classes PP, BPP, RP, and ZPP. These classes are then placed in the hierarchy of more well known complexity classes P, NP, and PSPACE. In the process, some features of randomized computation are explored, such as the ability to "amplify" the accuracy of any randomized algorithm with bounded error.

Section 5 introduces randomized parallel computational models and discusses the use of Markov chains for analysis of randomized algorithms.

Section 6 provides an overview of derandomization, currently a very active area of research. Derandomization includes finding deterministic counterparts for specific randomized algorithms, along with addressing the question of whether all efficient randomized algorithms have corresponding efficient deterministic algorithms.

2 A Randomized Algorithm for Polynomial Identity Testing

Probably the two most well known problems for which randomized algorithms offer efficient solution are primality and polynomial identity testing. The problem of determining if a given integer is a prime number finds many applications in computer science and was recently shown to have a deterministic polynomial-time solution [AKS02].

Polynomial identity testing is one of the remaining problems that has no known deterministic polynomial-time solution but has a polynomial-time randomized solution. We present an overview of the problem and a randomized algorithm for solving it (adapted from [Kab05]). More in depth details can be found in [MR95].

Definition 1. *Polynomial Identity Testing*: Given two polynomials p_1 and p_2 in variables $x_1, \ldots, x_n$, decide if p_1 is identically equal to p_2.

If the polynomials are given explicitly the solution is trivial: check that all of the coefficients of their corresponding monomials are equal. There is no such trivial solution when the polynomials are given implicitly, for example, as the determinants of symbolic matrices.

Let p_1 and p_2 be two polynomials over n variables. Let $h(x_1, \ldots, x_n) = p_1(x_1, \ldots, x_n) - p_2(x_1, \ldots, x_n)$. Clearly, $p_1(x_1, \ldots, x_n) = p_2(x_1, \ldots, x_n) \Leftrightarrow h(x_1, \ldots, x_n) = 0$. The following algorithm provides a probabilistic test of whether a given polynomial is equal to zero and therefor of whether two polynomials are equal.

Given the polynomial h over n variables:

1. Let $\{r_1, \ldots, r_n\}$ be randomly and independently chosen from the interval $[1 \ldots N]$
2. If $h(r_1, \ldots, r_n) = 0$, output "non-zero". Otherwise, output "probably zero".

If $h(x_1, \ldots, x_n) = 0$ then $h(r_1, \ldots, r_n) = 0$ will alway be true and the algorithm will always be right. That is, it will output "probably zero" with probability one. However, if $h(x_1, \ldots, x_n) \neq 0$ then there is some chance that the algorithm will randomly select one of the roots of h and mistakenly output "probably zero".

Let d be the degree of h (equal to the highest degree of all of the monomials of h). Then h can have at most d roots and the probability of selecting one of these roots randomly is at most $\frac{d}{N}$. There are two ways to reduce the error of this algorithm: (1) increase the range N from which the random numbers are selected; (2) repeat the algorithm and take the majority vote of the answers. As shown by the amplification lemma presented in Section 4, the second method can be used to lower the error to an exponentially small probability using only a polynomial number of iterations.

3 Computability and Randomization

Successful use of randomness to efficiently solve problems has raised the question of whether allowing a machine access to random numbers increases its computational power. More formally, can a Turing machine with access to random numbers compute more than a Turing machine without? The answer seems to depend on just what it meant by "random numbers". This is a somewhat surprising and interesting result considering that the power of Turing machines is stable even after the addition of capabilities such as nondeterminism. For this section, we will refer to Turing machines with access to random numbers as RTM and to those without as TM.

If we define a random number to be *any positive integer* then Dijkstra [Dij76] shows that RTM are in fact more powerful than TM. The key property of this type of random number is that it is *unbounded*. Dijkstra's proof using weakest preconditions is reproduced here [Dij76], pages 76-77.

> We could try to make the program S: "set x to any positive integer" with the properties:
>
> (a) $wp(S, x > 0) = T$.

(b) $(\forall s : s \geq 0 : wp(S, 0 < x < s) = F)$

Here property (a) expresses the requirement that activation of S is guaranteed to terminate with x equal to some positive value, property (b) expresses that S is a mechanism of unbounded non-determinacy, i.e. that no a priori upper bound for the final value of x can be given. For such a program S, we could, however, derive now:

$$\begin{aligned} T &= wp(S, x > 0) \\ &= wp(S, (\exists r : r \geq 0 : 0 < x < r)) \\ &= (\exists s : s \geq 0 : wp(S, 0 < x < s)) \\ &= (\exists s : s \geq 0 : F) \\ &= F \end{aligned}$$

This, however, is a contradiction: for the mechanism S "set x to any positive integer" *no* program exists!

The fact that "no program exists" implies that "set x to any positive integer" can not be performed by a TM. Therefor, if we define an RTM to have this capability then it is more powerful than a TM.

This, however, is not the standard use of the term *random*. It may be more accurate to consider this as the definition of *unbounded*. It is the possibility of an infinite number of choices available in a finite time that provides the extra power to the TM.

Now, we define a *random choice* to be one between a finite set of possibilities, with some non-zero probability of choosing each element. In this case it appears that the power of the RTM depends on the presence of a bias in the random choice.

Gill, in [Gil77], briefly mentions some early results on the power of machines with random choice. Gill's summary of these results is reproduced here.

> Computability by probabilistic machines was first studied by DrLeeuw, Moore, Shannon, and Shapiro [dLMSS56]. In that paper, the authors considered the question: Is there anything that can be done by a machine with access to random inputs that cannot be done by a deterministic machine? The random inputs were required to be drawn from a source of independent, equiprobable binary digits, and the tasks to be performed by the machines were the enumerations of sets of natural numbers. This question was answered in the negative. (In the case that

the random inputs are independent but biased, say with bias $p \neq 1/2$, then possibly more sets can be listed by probabilistic devices, but only sets that are recursively enumerable in the real number p.)

This suggests that for the common definition of *random*, where there is a equal chance of choosing between a finite set of elements, an RTM is no more powerful than a TM. We have not found any more recent results showing that biased choices may add power, or that any other property of using a random choice over a finite set adds computational power. We therefor conclude that, in general, randomness does not add power. Further, it is common in the literature to view randomness as a special case of nondeterminism, as explored further in section 4.1. It is well known that a nondeterministic TM is no more powerful than a deterministic TM, and it is therefor reasonable to assume that an RTM is no more powerful than a TM.

4 Randomized Complexity Classes

The complexity classes of randomized algorithms and how such complexity classes relate to traditional complexity classes is foundational to the theory of randomized computation. In this section, we define the complexity classes commonly used to describe randomized algorithms. We first view randomization as it relates to nondeterminism. This allows us to more easily place the randomized complexity classes in relation to the standard complexity classes P and NP.

The classes RP, ZPP, and BPP were first defined by Gill [Gil77]. The definitions given here are adapted from [MR95] and [Sip96].

4.1 Randomization and Nondeterminism

The purpose of this section is to demonstrate a relationship between randomized computation and nondeterministic computation. At first, we continue to refer informally to general machines with access to some sort of random numbers as RTM. We then provide a more formal model of randomized computation called a *probabilistic Turing machine* (PTM), which is used in later sections to define certain randomized complexity classes.

A *computation* C of a nondeterministic TM M is a tree. Each node in C is a possible configuration of M. There is an edge from configuration c_i to c_j if c_j is a possible next configuration from c_i. M is said to *accept* an input w if the computation of M on w has at least one branch of the computation that ends in the accept state. M is said to reject otherwise.

In general, nondeterministic computation can not be efficiently implemented. Because the width of the computation tree can expand exponentially, either an exponential time or an exponential number of parallel processors would be required to simulate the computation.

However, we can efficiently simulate *one possible branch* of the computation using randomization. An RTM R simulating one branch of computation from an NTM M operates as follows:

1. The starting configuration of R given input w is the starting configuration of M given input w.
2. Let the current configuration of R be denoted by c.
3. Repeat the following:
   (a) Let D be the set of possible next configurations from c, according to M.
   (b) Randomly and uniformly choose one possible configuration $c_n \in D$ to be the next configuration of R (set $c = c_n$).
   (c) If c represents an accept state according to M, R accepts. Else if c represents a reject state according to M, R rejects. Otherwise, continue.

If all computation paths of M are equally likely, the probability of R accepting w is equal to the ratio of all computation paths of M on w that are accepting. We now formalize these concepts with a model of randomized computation called a *probabilistic Turing machine*.

Definition 2. A probabilistic Turing machine M is a kind of NTM, with *coin flip choices* instead of nondeterministic choices. A coin flip choice is an unbiased random choice between two successor states. A PTM follows only one possible branch of a nondeterministic choice, whereas a NTM follows them all. The probability that M will follow a given computation branch b is

$$Pr[b] = 2^{-k}$$

where k is the number of coin flip choices on branch b. The probability that M accepts w is

$$Pr[M \text{ accepts } w] = \sum_{b \in A} Pr[b]$$

where A is the set of all accepting branches. M is said to reject w if and only if it does not accept w

$$Pr[M \text{ rejects } w] = 1 - Pr[M \text{ accepts } w]$$

As a technical convenience, we assume that all computation branches are of the same length.

The complexity classes defined below differ simply in the range of accepting and rejecting probabilities they allow when claiming that a PTM "decides" a language.

4.2 The Class BPP

Before defining BPP, we first define a class of languages known as *probabilistic polynomial-time* (PP).

Definition 3. PP is the class of languages $\{S|S = L(M), M \text{ is a PTM } \}$ such that the following hold when M is run on $w \in \Sigma^*$

$$w \in S \Rightarrow Pr[M \text{ accepts } w] > 0.5$$

$$w \notin S \Rightarrow Pr[M \text{ accepts } w] < 0.5$$

We seek to provide a machine that can decide all of the languages in PP with arbitrarily small error in polynomial time. The general method for this involves designing a PTM M that decides a language S, invoking M on w a polynomial number of times, and using the *majority decision* of all invocations. However, the probabilities of error in PP can be arbitrarily close to 0.5. In the worst cases, we may require more than a polynomial number of invocations to determine if $w \in S$. To solve this, we define a class of languages with error bounded away from 0.5, *bounded-error probabilistic polynomial-time* (BPP).

Definition 4. BPP is the class of languages $\{S|S = L(M), M\text{is a PTM}\}$ such that the following hold when M is run on $w \in \Sigma^*$

$$w \in S \Rightarrow Pr[M \text{ accepts } w] > 0.5 + \epsilon$$

$$w \notin S \Rightarrow Pr[M \text{ accepts } w] < 0.5 - \epsilon$$

where ϵ is a constant $0 < \epsilon < 0.5$.

The *amplification lemma* given below, reproduced from [Sip96], states that the any language in BPP has a decider with exponentially small error.

Lemma 1. *Let ϵ be a fixed constant strictly between 0 and 0.5. Then for any polynomial poly(n) a probabilistic polynomial time Turing machine M_1 that operates with error probability ϵ has an equivalent probabilistic time Turing machine M_2 that operates with an error probability of $2^{poly(n)}$.*

The proof idea, as alluded to above, requires M_2 to simulate M_1 a polynomial number of times and take the majority vote of the outcomes. The probability of error decreases exponentially with the number of runs of M_1.

A formal proof that the error actually decreases exponentially with only a polynomial number of runs is provided in several forms. Sipser [Sip96] provides a self-contained, highly technical, proof. The more common proof utilizes a result from probability theory known as the *Chernoff bound*. We provide such a proof here, which is adapted from [Bul05].

We first provide the Chernoff bound theorem.

Let $X_1, X_2, \ldots, X_n$ be a series of independent experiments (for example, coin flips) such that the probability of success in each of them is p.

Theorem 1 (Chernoff Bound). *If $p = \frac{1}{2} + \epsilon$ for some $\epsilon > 0$, then the probability that the number of successes in a series of n experiments is less than $\frac{n}{2}$ is at most $e^{-\frac{\epsilon^2 n}{6}}$.*

The proof of the Chernoff bound theorem (in a different but closely related form) can be found in [MR95].

Proof of Amplification Lemma. Let M_1 be a probabilistic Turing machine that such that $L(M_1) \in BPP$. On any input w, M_1 has error $0.5 - \epsilon$. We construct a machine M_2, such that $L(M_2) = L(M_1)$ and M_2 has error less than $2^{-p(|w|)}$, where $p(|w|)$ is a polynomial of the length of w.

M_2 = "on input w

1. simulate M_1 on w $t(|w|)$ times
2. if more than $\frac{t(|w|)}{2}$ runs of M_1 accept, then accept; otherwise reject"

Using the Chernoff bound, the number $t(|w|)$ must be such that

$$\begin{aligned} e^{-\frac{\epsilon^2 t(|w|)}{6}} &< 2^{-p(|w|)} \\ -\frac{\epsilon^2 t(|w|)}{6} &< -p(|w|)ln2 \\ \epsilon^2 t(|w|) &> 6p(|w|)ln2 \\ t(|w|) &> \frac{6ln2}{\epsilon^2} p(|w|) \end{aligned}$$

So, a polynomial $t(|w|)$ runs of M_1 provide an exponentially small error. □

Let co-BPP = $\{S|\bar{S} \in$BPP$\}$. It is obvious that BPP = co-BPP, because the error probabilities for $x \in S$ and $x \notin S$ are symmetric.

4.3 The Class RP

Definition 5. *RP* (randomized polynomial-time) is the set of languages $\{S|S = L(M), M \text{ is a PTM }\}$ such that the following hold when M is run on $w \in \Sigma^*$

$$w \in S \Rightarrow Pr[M \text{ accepts } w] \geq \epsilon$$

$$w \notin S \Rightarrow Pr[M \text{ accepts } w] = 0$$

where $0 < \epsilon < 1$.

RP differs from BPP in that it has only *one-sided error*. That is, it has some chance of accepting if $x \in S$ and has no chance of accepting if $x \notin S$. The choice of ϵ is unimportant, as we can use the amplification lemma for languages in RP just as it was used for those in BPP.

Unlike BPP, co-RP $\neq$ RP. We define co-RP $= \{S|\bar{S} \in \text{RP }\}$ to have a probability of error only when $w \notin S$.

Definition 6. co-RP is the set of languages $\{S|S = L(M), M \text{ is a PTM }\}$ such that the following hold when M is run on $w \in \Sigma^*$

$$w \in S \Rightarrow Pr[M \text{ accepts } w] = 1$$

$$w \notin S \Rightarrow Pr[M \text{ accepts } w] \leq \epsilon$$

where $0 < \epsilon < 1$.

4.4 The Class ZPP

Definition 7. *ZPP* (zero-error probabilistic polynomial-time) is the class of languages equal to RP $\cap$ co-RP.

A language $S \in ZPP$ has two probabilistic polynomial time Turing machines, M_1 and M_2 such that M_1 always only accepts on input w if $w \in S$ and M_2 only rejects if $w \notin S$. M_1 and M_2 can be repeatedly run on w until a definite answer is given. That is, until either M_1 accepts or M_2 rejects. We define a machine M that decides a language $S \in ZPP$ given two such machines.

$M =$ "on input w

1. Repeat the following
   (a) Run M_1 on input w, if M_1 accepts then accept. Otherwise continue.

(b) Run M_2 on input w, if M_2 rejects then reject. Otherwise continue."

Theorem 2. *M will always halt with the correct answer and the expected running time of M is polynomial.*

Proof. M is clearly correct because M_1 is defined to never give a false acceptance and M_2 is defined to never give a false rejection.

Assume that the probability that a single execution of M's loop does not produce a definite answer is 0.5. This assumption is safe because if it were not true it could be made so by application of the amplification lemma to M_1 and M_2. Let the polynomial $p(|w|)$ be the running time of each execution of M's loop. The the expected running time of M is

$$\sum_{i}^{\infty} 0.5^i ip(|w|) = 2p(|w|)$$

□

4.5 Defining "Efficient" Computation

Outside of randomized complexity classes, the typical definition of "efficient" computation is the class P. That is, the class of problems that can be solved in polynomial time with a deterministic machine. Problems in NP (that are not known to be in P) are not considered efficiently solvable because nondeterministic machines can not be efficiently simulated.

With the assumption that realizable computers can have access to random number generators, a machine corresponding to a language in any of the randomized complexity defined above is efficient. That is, they can actually be implemented to run in polynomial time. However, these machines now have some probability of error. The question of efficient computation for probabilistic machines is how large a probability of error can be tolerated.

Certainly ZPP can be considered efficient because there is zero probability of error and expected polynomial running time. Using the amplification lemma, The classes RP, co-RP, and BPP have exponentially small probability of error given polynomial running time. For all practical purposes, exponentially small error is the same as zero error. After all, if an algorithm has a probability of error of the order 2^{-100} then the chance that the algorithm will make an error is far smaller than the chance that the physical machine itself will make an error.

Problems in PP, on the other hand, may require exponential running time to achieve acceptable probabilities of error and are therefor not considered to have efficient solutions.

In general, BPP is the largest class considered to have efficient solutions.

Each of the complexity classes with efficient solutions have a class of corresponding algorithms that provide such solutions. Whereas the probabilistic Turing machines used to define the classes deal only with decision problems, these algorithms perform more general computation.

Definition 8. *Las Vegas algorithms* are those that always give the correct solution but can vary in their running time for a given input. These algorithms correspond to the class ZPP.

Definition 9. *Monte Carlo algorithms* are those that have a non-zero, bounded probability of error. For Monte Carlo algorithms corresponding to decision problems, there are two types, those with *one-sided error* and those with *two-sided error*. These correspond to RP $\cup$ co-RP and BPP respectively.

4.6 Class Containments

In this section we present the known class containments between the randomized complexity classes ZPP, RP, co-RP, BPP, PP, and the standard complexity classes P, NP, co-NP, and PSPACE.

Theorem 3. $P \subseteq ZPP$

Proof. Machines in P run in polynomial time and have zero error. They are like machines in ZPP that don't use any random numbers. □

Theorem 4. $ZPP \subseteq RP$ *and* $ZPP \subseteq coRP$.

Proof. By definition, $ZPP = RP \cap coRP$. □

Theorem 5. $RP \subseteq NP$

Proof. Let M_1 be a PTM such that $L(M_1) \in RP$. For M_1 run on w, more than half of the computation paths of M_1 accept if $w \in L(M_1)$ and no paths accept if $w \notin L(M_1)$. For a NTM M_2 such that $L(M_2) \in NP$ at least one computation path must accept if $w \in L(M_2)$ and no paths accept if $w \notin L(M_2)$. Therefor $M_1 \in NP$. □

Theorem 6. $coRP \subseteq coNP$

Proof. This is the complement of the proof that $RP \subseteq NP$. □

Theorem 7. $RP \cup coRP \subseteq BPP$

Proof. RP and coRP have bounded error on only one side. BPP has bounded error on two sides. One sided error is a special case of two sided error. □

Theorem 8. $BPP \subseteq PP$

Proof. By definition, BPP is a bounded version of PP. □

Next, we show that $NP \subseteq PP$. First we will define a class PP1 that is equivalent to PP to aid in the proof.

Definition 10. PP1 is the class of languages $\{S|S = L(M), M \text{ is a PTM }\}$ such that the following hold when M is run on $w \in \Sigma^*$

$$w \in S \Rightarrow Pr[M \text{ accepts } w] > 0.5$$

$$w \notin S \Rightarrow Pr[M \text{ accepts } w] \leq 0.5$$

The only difference between PP1 and PP is that PP1 can have a probability of accepting equal to 0.5 when $w \notin S$, where as for PP the probability of accepting must be strictly less than 0.5.

Here we provide a short proof idea that $PP1 = PP$.

$PP \subseteq PP1$ is obvious by the definition.

$PP1 \subseteq PP$ is shown as follows.

If a PTM M_1 decides a language in PP1 then

$$w \in L(M_1) \Rightarrow Pr[M \text{ accepts } w] > 0.5$$

which is equivalent to

$$w \in L(M_1) \Rightarrow Pr[M \text{ accepts } w] \geq 0.5 + \epsilon$$

where $0 < \epsilon < 0.5$.

Using M_1 we can construct a machine M_2 such that $L(M_1) = L(M_2) \in PP$ by "shifting" some of the error from the accept side to the reject side. In other words, the following will hold when M_2 is run on w

$$w \in S \Rightarrow Pr[M \text{ accepts } w] \geq 0.5 + (\epsilon - \delta)$$

$$w \notin S \Rightarrow Pr[M \text{ accepts } w] \leq 0.5 - \delta$$

as long as $\delta < \epsilon$ this is equivalent to the original definition of PP. The existence of such a δ, the ability to calculate it, and the ability to successfully "shift" the error is proved in [Gol99].

Theorem 9. $NP \subseteq PP$

Proof. We show that the NP-complete problem of satisfiability of Boolean formula is in PP1. Therefor, every problem in NP is in PP1. By the fact that PP1 is equivalent to PP, every problem in NP is in PP, and $NP \subseteq PP$.

$$SAT = \{\langle\phi\rangle | \phi \text{ is a satisfiable Boolean formula}\}$$

Construct M to be a PTM such that $SAT = L(M) \in PP1$.

$M =$ "on input $\langle\phi\rangle$

1. Choose an assignment of values to ϕ uniformly at random.
2. If the assignment makes ϕ true, accept. Otherwise, accept with probability 0.5 and reject with probability 0.5."

Therefor

$$\langle\phi\rangle \in SAT \Rightarrow Pr[M \text{ accepts } \langle\phi\rangle > 0.5$$

$$\langle\phi\rangle \notin SAT \Rightarrow Pr[M \text{ accepts } \langle\phi\rangle = 0.5$$

and $SAT \in PP1$. We know $PP1 = PP$, therefor $SAT \in PP$ and $NP \subseteq PP$.

□

Theorem 10. $coNP \subseteq PP$

Proof. This is obvious from $NP \subseteq PP$ and $PP = coPP$. □

Theorem 11. $PP \subseteq PSPACE$

The following proof is adapted from [Gol99].

Proof. Let M_1 be a PTM such that $L(M_1) \in PP$. Let $p(\cdot)$ be the polynomial bound on its running time. We construct a new machine M_2 that decides $L(M_1)$ in polynomial space.

$M_2 =$ "on input w

1. Run M_1 on w using *all possible* coin toss sequences of length $p(|w|)$.
2. If a majority of the runs of M_1 accept, then accept. Otherwise, reject."

Every invocation of M_1 requires only polynomial space (because it runs in polynomial time). Because the space for each invocation can be reused, M_2 also runs in polynomial space. M_2 answers correctly because M_1 is a PP machine that answers correctly for more than half of the possible computation paths and M_2 makes a majority decision over all computation paths of M_1. □

Figure 1 gives a graphical representation of these complexity class containments.

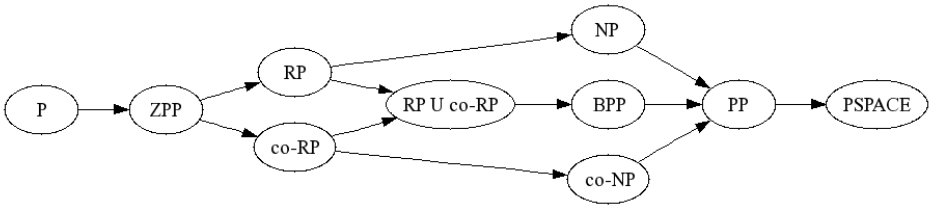

Figure 1: Hierarchy of Complexity Classes. An edge from A to B indicates that A $\subseteq$ B.

The most interesting open question in this area is determining if either $BPP \subseteq NP$ or $NP \subseteq BPP$ are true. The second possibility would be particularly unexpected because BPP is seen as containing only efficiently solvable problems and NP as containing many problems for which no efficient solution is known.

5 Probabilistic Models of Computation

The most widely used model for describing a probabilistic computation is a probabilistic Turing machine (defined in Section 4). It has been used extensively to analyze probabilistic algorithms and argue about probabilistic complexity classes. The model is general enough to describe most aspects of probabilistic computation, however it is sometimes more convenient to use other models when analyzing probabilistic algorithms. In this survey we look more closely at two such approaches: the parallel machine model and the use of Markov chains for analyzing randomized computation.

5.1 Parallel Machine Models

With the development of low-cost hardware the possibility of performing many computations in parallel have emerged and algorithms for parallel machine have been developed. Adding randomization to parallel computation created an opportunity to create and analyze many new efficient algorithms.

To standardize analysis of parallel algorithms a number of parallel machine models were constructed. The most widely used ones are fixed connection machine, shared memory model, Boolean circuit model [RR87].

A **fixed connection network** is a directed graph $G = (V, E)$ in which nodes represent processors and edges represent connections between the processors. The degree of each node is assumed to be a constant or a slowly increasing function of the number of nodes in G.

In **shared memory model** processors communicate with each other synchronously by using the commonly accessible shared memory. Each processor is a random access machine and each step of an algorithm in shared memory model is an operation, comparison or a memory access. Since all processor can potentially modify memory contents, there is a need for a mechanism to resolve read/write conflicts. EREW PRAM is the model with no simultaneous read or write access, CREW PRAM allows concurrent read and forbids concurrent write access, CRCW PRAM allows both concurrent read and write and resolves conflicts using a priority scheme.

A **Boolean circuit** on n input variables is a directed acyclic graph (DAG) with nodes of in-degree of 0 labeled with variables and their negations. The other nodes are called gates and are labeled with boolean operations. The size of such circuit is the number of gates, the fan-in is the maximum degree of any node, the depth is the depth of the underlying DAG. A Boolean circuit C accepts language $L \subset \Sigma^*$ if on any input $w \in L$, C evaluates to 1. The size complexity of L is defined to be the size of the smallest circuit that accepts L.

To formalize the notion of efficient computation on PRAM models let us define the complexity NC and RNC as follows [MR95].

Definition 11. Complexity class NC is composed of languages L for which a PRAM algorithm A exists and for any $w \in \Sigma^*$,

$$w \in L \Rightarrow A(w)\ accepts, A(w)\ rejects \text{ otherwise}$$

The number of processors used by A is polynomial in $|w|$ and the number of steps is polylogarithmic in $|w|$. See [RG95] for more detailed definitions and properties of NC.

Definition 12. Complexity class RNC consists of languages L for which a PRAM algorithm exists and for any $w \in \Sigma^*$,

$$w \in L \Rightarrow Pr[A(w)\ accepts] \geq 1/2$$
$$w \notin L \Rightarrow Pr[A(w)\ accepts] = 0$$

The number of processors used by A is polynomial in $|w|$ and the number of steps is polylogarithmic in $|w|$.

The class RNC has the same relation to NC as RP does to P and is contained in BQNC (Bounded-Error Quantum NC). One example of a problem from RNC is the maximum matching problem for bipartite graphs [MVV87].

5.2 Markov Chains

Markov chains provide a powerful tool for modeling and analyzing random processes. We will start by defining Markov process and Markov chain and then show how they can be used by analyzing a simple randomized algorithm for 3-SAT problem.

Definition 13. A stochastic process $X = \{X(t) : t \in T\}$ is a collection of random variables, where t often represents time. If the set T is countably infinite, then X is called a discrete time process.

A discrete stochastic process $X_0, X_1, X_2, ...$ is called a **Markov chain** if

$$Pr[X_t = a_t | X_{t-1} = a_{t-1}, X_{t-2} = a_{t-2}, ..., X_0 = a_0] = Pr[X_T = a_t | X_{t-1} = a_{t-1}]$$

Thus a state X_t depends on the previous state X_{t-1} but is independent of the history of how the process arrived at state X_t. For a process with state space $\{0, 1, 2...\}$ the transition probability $P_{i,j} = Pr[X_t = j | X_{t-1} = i]$ is the probability that process moves form state i to j in one step. For any $m \geq 0$ we define the m-step transition probability $P^m_{i,j} = Pr[X_{t+m} = j | X_t = i]$ as the probability that the chain moves from state i to state j in exactly m steps.

Using definitions from above we present the analysis of expected running time for a randomized algorithm that solves 3-SAT problem.

3-SAT-RAND:

1. Repeat up to m times, terminating if all clauses are satisfied:
   (a) Start with a truth assignment chosen uniformly at random
   (b) Repeat the following up to $3n$ times, terminating if the satisfying assignment is found:
      i. Choose an arbitrary clause that is not satisfied
      ii. Choose one of the literals uniformly at random and change the value of the variable in the current truth assignment
2. If a valid truth assignment has been found, return it.
3. Otherwise, return that the formula is unsatisfiable.

The above algorithm uses up to $3n$ steps starting from a random assignment. If it fails to find a satisfied assignment it restarts the search with a new assignment chosen at random. Thus the algorithm can be described as discrete time stochastic process. The start state is the initial randomly

chosen assignment, the length of the Markov chain is at most $3n$ and transition probability from one state to the next correspond to the probability of flipping the assignment of a particular literal. To argue about the expected running time of 3-SAT-RAN we need to establish bounds on how many times the search will be restarted before the satisfying assignment is found.

Let q be the probability that 3-SAT-RAND finds some satisfying assignment S in $3n$ steps starting with an arbitrary assignment. Furthermore, let q_j be a lower bound on the probability that the algorithm find S when it is started with the truth assignment that includes exactly j variables that do not agree with S. Notice that the probability of flipping assignment of k literals such that they do not agree with S and flipping assignment of $k+j$ literals such that they agree with S is given by

$$\binom{j+2k}{k}\left(\frac{2}{3}\right)^k\left(\frac{1}{3}\right)^{j+k}$$

Therefore the lower bound on the probability that the satisfying assignment is reached within $j+2k \leq 3n$ steps is

$$q_j \geq \max_{k=0,\ldots,j}\binom{j+2k}{k}\left(\frac{2}{3}\right)^k\left(\frac{1}{3}\right)^{j+k}$$
$$q_j \geq \binom{3j}{j}\left(\frac{2}{3}\right)^j\left(\frac{1}{3}\right)^{2j}, j=k$$

Using Stirling's approximation formula and simple calculations (see [MM05] for details), it can be shown that

$$\binom{3j}{j} = \frac{c}{\sqrt{j}}\left(\frac{27}{4}\right)^j, c = \sqrt{3}/8\sqrt{\pi}$$

Therefore for $j > 0$

$$\begin{aligned} q_j &\geq \binom{3j}{j}\left(\frac{2}{3}\right)^j\left(\frac{1}{3}\right)^{2j} \\ &\geq \frac{c}{\sqrt{j}}\left(\frac{27}{4}\right)^j\left(\frac{2}{3}\right)^j\left(\frac{1}{3}\right)^{2j} \\ &\geq \frac{c}{\sqrt{j}}\left(\frac{1}{2}\right)^j \end{aligned}$$

The above establishes the lower bound for $q_j, j > 0$ and by definition of q_j, $q_0 = 1$. Now we can derive the lower bound for q, that is the probability that the process reaches a satisfying assignment in $3n$ steps:

$$\begin{aligned} q &\geq \sum_{j=0}^{n} Pr[\text{random assignment has j mismatches with S}] \cdot q_j \\ &\geq \frac{1}{2^n} + \sum_{j=1}^{n} \binom{n}{j} \left(\frac{1}{2}\right)^n \frac{c}{\sqrt{j}} \left(\frac{1}{2}\right)^j \\ &\geq \frac{c}{\sqrt{n}} \frac{1}{2^n} \sum_{j=0}^{n} \binom{n}{j} \left(\frac{1}{2}\right)^j 1^{n-j} \end{aligned}$$

Now using the fact that $\sum_{j=0}^{n} \binom{n}{j} \left(\frac{1}{2}\right)^j 1^{n-j} = \left(1 + \frac{1}{2}\right)^n$, we have:

$$q \geq \frac{c}{\sqrt{n}} \left(\frac{3}{4}\right)^n$$

If the some satisfying assignment S exists, then the number of random assignments the process will try before finding S is the geometric random variable with parameter q. Thus the expected number of assignments tried is $1/q$. Since for each assignment tried the algorithm uses at most $3n$ steps, the expected number of steps before termination is $O(n^{\frac{3}{2}}(4/3)^n)$.

6 Derandomization

In Section 2 we gave an example of an algorithm that can solve a 'hard' problem given access to a random source. The interesting question is whether randomness in such algorithms is a necessary component or finding a deterministic solution to a problem solved by a randomized algorithm is just a matter of time. If a randomized algorithm has a deterministic counterpart, then we still need to discover a way to construct such deterministic algorithm. Various aspects of these questions are tackled in numerous papers and surveys on approaches to derandomization [Kab02, Imp03, SRR01].

6.1 Example Construction of a Deterministic Algorithm

The stimulus for constructing deterministic algorithms from randomized counterparts was first explored in [Lub85]. Luby presented simple Monte Carlo randomized (parallel) algorithm for finding a maximal independent

set in a graph.

Definition 14. Maximal Independent Set (MIS) in an undirected graph is a maximal collection I of vertices such that no pair of vertices in I are adjacent. The MIS problem is a problem of finding MIS in a graph G.

The input to the MIS algorithm is an undirected graph G = (V,E), the output is MIS $I \subseteq V$. For all $W \in V$, let $N(W)$ the set of vertices that are adjacent to w, $N(W) = \{w \in V : \exists v \in W, (v, w) \in E\}$. Let $d(v)$ be the degree of vertex v with respect to the graph G'. Then the following is a randomized (parallel) algorithm to find MIS in G:

```
I <- empty
G' = (V', E') <- copy of G
while V' is not empty
      X <- empty
      in parallel for all v in V'
         if d(v) = 0
               add v to X
         else
               randomly [with probability 1/2d(v)]
               choose to add v to X
      I' <- X
      in parallel for all v,w in X
         if (v,w) is in E'
               if d(v) is equal or less than d(w)
                        I' <- I' - {v}
               else
                        I' <- I' - {w}
      I <- I U I'       // Note the I' is MIS for G'
      Y <- I' U N(I')
      G' = (V',E') is the induced subgraph on V' - Y
```

For all $v \in V$, let E_v be the event that v is chosen to be added to set X. Since the algorithm has access to a source of randomness, such events will be mutually independent. Luby showed that upon assumption of mutual independence of the events $\{E_v\}$, I is the MIS for G at the termination of the algorithm and the expected number executions of the while loop is $O(log\ n)$ with very high probability. Each execution of the while loop can be implemented (using EREW PRAM model) in $O(log\ n)$ time with $O(|E|)$ processors, and $O(n)$ expected number of random bits used. Thus the expected running time of the algorithm using $O(|E|)$ processors and $O(n\ log\ n)$ random bits total is $O(log^2\ n)$.

It is also shown in [Lub85] that the above result holds when the events $\{E_v\}$ are only pairwise independent. Moreover it is possible to construct a probability distribution D such that events $\{E_v\}$ are pairwise independent and the number of points in the sample space is $O(n^2)$. Probabilities of

events in $\{E_v\}$ in the new probability distribution are related to the ones from the original distribution as follows:

$$Pr[E_i] = p'_i = \frac{\lfloor p_v \cdot q \rfloor}{q} = \frac{\lfloor \frac{q}{2d(v)} \rfloor}{q}, \quad n \leq q \leq 2n, q \text{ is a prime}$$

Taking into account this difference, a deterministic algorithm DET-MIS has all the steps from RAND-MIS with one modification:

if there is a vertex $v \in V$ such that $d(v) \geq n/16$, then v is added to the independent set I immediately and $\{v\} \bigcup N(\{v\})$ is deleted from the graph

With this modification at least 1/16 of the vertices are eliminated from the graph. If no vertex in the original graph G has degree $\geq n/16$, then:

$$\begin{gathered} \forall v \in V, d(v) < n/16 \Rightarrow \\ \frac{q}{2d(v)} \geq \frac{n}{2d(v)} > 8 \Rightarrow \\ p'_i \geq 8/q \Rightarrow \frac{\lfloor p_v \cdot q \rfloor}{8} \geq 1 \end{gathered}$$

And since $\frac{\lfloor p_v \cdot q \rfloor + 1}{q} \geq p_v$, $\frac{8}{9} p_v \leq p'_v \leq p_v$.

This ensures the correct operation of the deterministic algorithm DET-MIS that uses the distribution D instead of a random source that is available to RAND-MIS.

The impact of Luby's work on the topic of derandomization was paramount. He was the first one to notice that the analysis of a randomized algorithm used such weak properties of randomness that randomness could be eliminated, resulting in a deterministic computation. Thus a new paradigm of creating efficient deterministic algorithms by derandomizing probabilistic ones emerged. This in turn sparked a large amount of research to find effective and systematic tools for derandomization.

6.2 Method of Conditional Probabilities

One of systematic and general way to derandomize a probabilistic algorithm is to use the method of conditional probabilities [ES73, ES74]. We will illustrate this method by derandomizing the probabilistic algorithm for finding a large cut [MM05]. The randomized algorithm is extremely simple: given a graph $G = (V, E)$ we find a partition of the n vertices from V into two sets

A and B by placing each vertex independently and uniformly at random into either A or B. The expected size of such cut is $E(C(A,B) \geq m/2)$, where m is the size of the optimal cut.

Let us consider vertices from V in some order $v_1, v_2, ..., v_n$. Let $s_1, s_2, ..., s_n$ be the sets (either A or B) in which each vertex v_i is placed. Suppose that we have already placed vertices $v_1, ..., v_k$. Let $E[C(A,B)|s_1, ..., s_k]$ be the expected value of the cut if the rest of the vertices $v_{k+1}, ..., v_n$ were placed randomly (independently and uniformly). To maintain the expected value of the cut $\geq m/2$ we will need and algorithm that places the next vertex v_{k+1} in either A or B such that

$$E[C(A,B)|s_1, ..., s_k] \leq E[C(A,B)|s_1, ..., s_{k+1}] \Rightarrow$$
$$E[C(A,B)] \leq E[C(A,B)|s_1, ..., s_n]$$

First note that it does not matter whether we place the first vertex in A or B, since the problem is symmetric (i.e. swapping sets A and B does not alter the solution), thus we are guaranteed $E[C(A,B)|s_1] = E[C(A,B)]$.

Let us assume that we successfully placed vertices $v_1, ..., v_k$. Consider placing v_{k+1} randomly with probability $1/2$ in either set A or B. Let X_{k+1} be a random variable that represents the set in which v_{i+1} was placed. Then:

$$E[C(A,B)|s_1, ..., s_k] =$$
$$\frac{1}{2}E[C(A,B)|s_1, ..., s_k, X_{k+1} = A] + \frac{1}{2}E[C(A,B)|s_1, ..., s_k, X_{k+1} = B]$$
$$max(\frac{1}{2}E[C(A,B)|s_1, ..., s_k, X_{k+1} = A], \frac{1}{2}E[C(A,B)|s_1, ..., s_k, X_{k+1} = B])$$
$$\geq E[C(A,B)|s_1, ..., s_k]$$

Thus we can calculate $E[C(A,B)|s_1, ..., s_k, X_{k+1} = A]$ and $E[C(A,B)|s_1, ..., s_k, X_{k+1} = B]$ and place v_{k+1} in set A if the first quantity is larger, and in B otherwise. To compute these quantities we can calculate the number of edges that contribute to the cut, given the placement of $k+1$ vertices. For every edge, the probability that it will eventually contribute to the cut is $1/2$, since the probability of the two endpoint edges ending up in different sets is $1/2$. Using the linearity of expectation property, $E[C(A,B)|s_1, ..., s_k, X_{k+1} = A]$ is the number of edges $(u,v) \in E$ crossing the cut, such that both u and v are already placed, plus $1/2$ of the rest of edges $\in E$ ($E[C(A,B)|s_1, ..., s_k, X_{k+1} = B]$ is computed in the same way). Now we can perform the deterministic placement that would satisfy

$$E[C(A,B)|s_1,...,s_k] \leq E[C(A,B)|s_1,...,s_k,s_{k+1}]$$

Notice that in this particular problem, the placement of a vertex v_{k+1} is completely determined by whether v_{k+1} has more neighbors in set A or B. This is the case, since all the edges (u,v) such that $v,u \neq v_{k+1}$ contribute the same amount to the expectation, regardless of the placement of v_{k+1}. Thus it suffices to employ deterministic greedy strategy of maximizing the number of edges crossing the cut when placing each v_{k+1} in order to guarantee a cut of size at least $m/2$.

The method of conditional probabilities have been explored at length in [NA00] and was successfully used to derandomize many known randomized algorithms: an algorithm for set discrepancy problem, set balancing problem and lattice approximation problem, near-optimal edge coloring on simple graphs and others [Rag88, RMN89] This method was also used to derandomize the famous semi-definite programming based algorithms for MAXCUT and maximum 2-satisfiability (MAX 2SAT) due to Goemans and Williamson [GW95].

6.3 Complexity Theory and Derandomization

Over the past years a number of randomized algorithms have been used in practice to solve hard problems. Using techniques presented above (as well as other methods) it was possible to derandomize many randomized algorithms. Significant progress on derandomization as well as success in using randomized algorithms for solving hard progress in practice lead to the supposition that the class BPP (bounded error probabilistic polynomial time) is equal or at least close to P. A general question addressing this problem is posed as follows: instead of derandomizing specific problems from class BPP, would it be possible to devise a general technique to derandomize the whole class BPP. If a general derandomization that runs in polynomial time can be found, this would naturally imply that BPP is equal to P.

Recall that an algorithm A that solves a problem from BPP receives an input string w of length n and a random string r and outputs the correct answer with high probability. More formally, for all strings w of length n it is that case that:

$$if\ w \in L\ then\ Pr_{r \in R\{0,1\}^n}[A(w,r) = accept] \geq 2/3$$
$$if\ w \notin L\ then\ Pr_{r \in R\{0,1\}^n}[A(w,r) = accept] \leq 1/3$$

Based on such randomized algorithm we can construct the following deterministic algorithm:

```
A_DET = "On input w:
 1. Enumerate all possible strings of length n,
 2. For each enumerated string x, run A(w,x)
 3. Output the majority vote as the result."
```

This trivial derandomization technique would produce an exponential time algorithm, thus showing that BPP $\subseteq$ EXP.

A thread of research pioneered by [Yao82, Sha81, BM84, RI89] suggested that it is possible to efficiently convert hardness into randomness by constructing pseudorandom generators. However the approaches proposed in these works required the existence of one-way functions (which is an assumption that is stronger than P $\neq$ NP) and did not allow a construction of pseudorandom generators for an arbitrary complexity class. An improvement on the earlier results was given by Nisan and Wigderson in [NN94] where they proposed a pseudorandom generator that can be used to generate a pseudorandom string to replace the truly random one needed as part of the input to a randomized algorithm.

Essentially a pseudorandom generator is a function that maps strings of input to longer strings of output in such a way that uniform distribution on the inputs to such function induces a seemingly uniform distribution on strings of output. In other words, for any class C (P, NP, PSPACE, etc.) a function that is hard for C can be used to construct a pseudorandom generator. Any algorithm from class C would not be able to distinguish the output bit sequence produced by such pseudorandom generator from a truly random sequence.

Definition 15. A **pseudorandom Generator (PRG)** is a function $G : \{0,1\}^{m(n)} \rightarrow \{0,1\}^n, n > 0$ such that

$$\text{for any circuit C, } |C| < n^k$$
$$Pr[C(U_n) = 1] - Pr[C(G(U_{m(n)})) = 1] \leq 1/n^k,$$

where U_i is a uniform distribution on $\{0,1\}^i$ and k is some polynomial of n. The resulting distribution $D_n = G(U_{m(n)})$ is said to be pseudorandom.

Once a pseudorandom generator is constructed, a random string $r \in U_n$ of length n given to a randomized algorithm can be replaced by a pseudorandom string $G(s) \in \{0,1\}^n$, where s is a shorter random seed of length $m(n)$ and $s \in U_{m(n)}$. The main implication of [NN94] in terms of the earlier question of whether BPP is equal to P can be stated as follows:

If there exists a function f computable in $DTIME(2^{O(n)})$ then:

- if f cannot be approximated by polynomial size circuits, then $BPP \subset \cap_{\epsilon>0} DTIME(2^{n^\epsilon})$

- if f cannot be approximated by circuits of size $2^{n^\epsilon}, \epsilon > 0$, then for some constant $c, BPP \subset DTIME(2^{(log\ n)^c})$
- if f has hardness $2^{\epsilon n}, \epsilon > 0$, then BPP = P

Subsequent work by Impagliazzo and Wigderson [IW97] improved the above result by only requiring existence of function in $E = DTIME(2^{O(n)})$ of complexity $2^{\Omega}(n)$.

Another approach to general derandomization was proposed by Andreev, Clementi and Rolim in [ACR97, ACR98] where they showed that hitting set generators (HSG) can be used to derandomize two-sided error probabilistic algorithms.

Definition 16. A **hitting set generator (HSG)** is a function $H = \{H_n : \{0,1\}^{m(n)} \rightarrow \{0,1\}^n, n > 0\}$ that, for any sufficiently large n and for any n-input Boolean circuit C with size at most n such that $Pr[C(U_n) = 1] \geq 1/n$, is required to provide just one "example" $w \in U_n$ for which $C(w) = 1$. In other words

$$\exists v \in \{0,1\}^{m(n)} \text{ such that } C(H_n(v)) = 1$$

The main theorem from [ACR98] states that there exists a deterministic algorithm A that, given a circuit C such that $n \leq q(n) \leq 2^n, |C| \leq q(n)$, and access to a quick hitting set generator $H : m(n) \rightarrow n, m(n) = \Omega(log\ n)$ computes a value A(C) such that

$$Pr[C = 1] - A(C) \leq 1/ \leq 1/q(n)$$

in time polynomial in $2^{m(q(n)^{O(1)})}$.

A corollary from this statement is that if there exists a quick HSG (a Boolean operator is quick if it is computable in time polynomial in the length of its output [NN94]) and we let $m(n) = O(log\ n)$ then BPP = P.

7 Conclusion

In this survey, we have given an overview of the foundations of the theory of randomized computation along with an examination of current research.

The foundational material consists mainly of determining the computational power of machines with access to random numbers and definition of randomized complexity classes. The randomized complexity classes PP,

BPP, RP, and ZPP were defined and a containment hierarchy of these classes, along with P, NP, and PSACE, was provided. We also presented models of computation and techniques used for analysis of randomized algorithms.

The current research described here pertains primarily to derandomization. Both the derandomization of specific algorithms and the derandomization of entire classes of algorithms was examined. After studying advances in the field of randomized computation we conclude that, in general, access to random numbers was not shown to increase the computational power of a machine. However the problem of formally proving that randomness does not result in additional power remains an open question.

Because adding random choice to algorithms can often provide benefits in both running time and simplicity, randomized computation will likely be a prolific area of research, both in practical and theoretical areas.

References

[ACR97] Alexander E. Andreev, Andrea E. F. Clementi, and Jose D. P. Rolim. Worst-case hardness suffices for derandomization: A new method for hardness-randomness trade-offs. In *ICALP '97: Proceedings of the 24th International Colloquium on Automata, Languages and Programming*, pages 177–187, London, UK, 1997. Springer-Verlag.

[ACR98] Alexander E. Andreev, Andrea E. F. Clementi, and Jose D. P. Rolim. A new general derandomization method. *J. ACM*, 45(1):179–213, 1998.

[AKS02] M. Agrawal, N. Kayal, and N. Saxena. Primes is in p, 2002.

[BM84] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Comput.*, 13(4):850–864, 1984.

[Bul05] A. Bulatov. Lecture notes on complexity, 2005. http://www.cs.sfu.ca/ abulatov/CMPT710/.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[dLMSS56] K. de Leeuw, E. F. Moore, C. E. Shannon, and N. Shapiro. Computability by probabalistic machines. In *Automata Sudies, Annals of Mathematics Studies*, volume 34. Princeton University Press, Princeton, N.J., 1956.

[ES73] P. Erdos and J. L. Selfridge. On a combinatorial game. In *Journal of Combinatorial Theory*, pages 14:298–301, 1973.

[ES74] P. Erdos and J. H. Spencer. Probabilistic methods in combinatorics. In *Academic Press*, 1974.

[Gil77] J. Gill. Computational complexity of probabilistic turing machines. *SIAM Journal on Computing*, 6(4):675–695, 1977.

[Gol99] O. Goldreich. Lecture notes on complexity theory, 1999. http://www.wisdom.weizmann.ac.il/mathusers/oded/cc99.html.

[GW95] Michel X. Goemans and David P. Williamson. Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming. *J. Assoc. Comput. Mach.*, 42:1115–1145, 1995.

[Imp03] Russell Impagliazzo. Hardness as randomness: a survey of universal derandomization. *CoRR*, cs.CC/0304040, 2003.

[IW97] Russell Impagliazzo and Avi Wigderson. P = bpp if e requires exponential circuits: derandomizing the xor lemma. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 220–229, New York, NY, USA, 1997. ACM Press.

[Kab02] Valentine Kabanets. Derandomization: A brief overview. In *Bulletin of the European Association for Theoretical Computer Science*, pages 88–103, 2002.

[Kab05] V. Kabanets. Polynomial identity testing; interactive proofs, 2005. http://www.cs.sfu.ca/ kabanets/cmpt308/lectures/23.txt.

[Lub85] M Luby. A simple parallel algorithm for the maximal independent set problem. In *STOC '85: Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pages 1–10, New York, NY, USA, 1985. ACM Press.

[MM05] Eli Upfal Michael Mitzenmacher. *Probability and Computing*. Cambridge University Press, 2005.

[MR95] R. Motwani and P. Raghavan. *Randomized algorithms*. Cambridge University Press, New York, NY, USA, 1995.

[MVV87] Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani. Matching is as easy as matrix inversion. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 345–354, New York, NY, USA, 1987. ACM Press.

[NA00] Joel H. Spencer Noga Alon. *The Probabilistic Method*. Wiley, 2000.

[NN94] A. Wigderson N. Nisan. Hardness vs. randomness. In *Journal of Computer and System Sciences*, 1994.

[Rag88] R. Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. In *Journal of Computer and System Science*, pages 37:130–143, 1988.

[RG95] Walter L. Ruzzo Raymond Greenlaw, H. James Hoover. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, 1995.

[RI89] M. Luby R. Impagliazzo, L. Levin. Pseudo-random generation from one way functions. In *STOC*, 89.

[RMN89] J. Naor R. Motwani and M. Naor. The probabilistic method yields deterministic parallel algorithms. In *391h Annual Symposium on the Foundations of Compuler Science*, 1989.

[RR87] S. Rajasekaran and J.H. Reif. Randomized parallel computation. In *Fundamentals of Computation Theory Conference, Kazen, USSR*. Springer-Verlag Lecture Notes in Computer Science, 1987.

[Sha81] Adi Shamir. On the generation of cryptographically strong pseudo-random sequences. In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, pages 544–550, London, UK, 1981. Springer-Verlag.

[Sip96] M. Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[SRR01] J.H. Reif S. Rajasekaran, P.M. Pardalos and J. Rolim. *Handbook of Randomized Computing*. Springer, 2001.

[Yao82] A.C. Yao. Theory and applications of trapdoor functions. In *Proceedings of the 23th Symposiuin on the Foundation of Computer Science*, pages 80–91, 82.