Discrete Applied Mathematics 34 (1991) 165–201
North-Holland

An introduction to randomized algorithms

Richard M. Karp*

Computer Science Division, University of California, Berkeley, CA 94720, USA; and International Computer Science Institute, Berkeley, CA 94704, USA

Received 6 November 1989
Revised 6 April 1990

Abstract

Karp, R.M., An introduction to randomized algorithms, Discrete Applied Mathematics 34 (1991) 165–201.

Research conducted over the past fifteen years has amply demonstrated the advantages of algorithms that make random choices in the course of their execution. This paper presents a wide variety of examples intended to illustrate the range of applications of randomized algorithms, and the general principles and approaches that are of greatest use in their construction. The examples are drawn from many areas, including number theory, algebra, graph theory, pattern matching, selection, sorting, searching, computational geometry, combinatorial enumeration, and parallel and distributed computation.

1. Foreword

This paper is derived from a series of three lectures on randomized algorithms presented by the author at a conference on combinatorial mathematics and algorithms held at George Washington University in May, 1989. The purpose of the paper is to convey, through carefully selected examples, an understanding of the nature of randomized algorithms, the range of their applications and the principles underlying their construction. It is not our goal to be encyclopedic, and thus the paper should not be regarded as a comprehensive survey of the subject.

This paper would not have come into existence without the magnificent efforts of Professor Rodica Simion, the organizer of the conference at George Washington University. Working from the tape-recorded lectures, she created a splendid transcript that served as the first draft of the paper. Were it not for her own reluctance she would be listed as my coauthor.

* Research supported by: NSF Grant CCR-8411954.

2. Introduction

A *randomized algorithm* is one that receives, in addition to its input data, a stream of random bits that it can use for the purpose of making random choices. Even for a fixed input, different runs of a randomized algorithm may give different results; thus it is inevitable that a description of the properties of a randomized algorithm will involve probabilistic statements. For example, even when the input is fixed, the execution time of a randomized algorithm is a random variable.

Isolated examples of randomized algorithms can be traced back to the very early days of computer science, but the central importance of the concept became generally recognized only about fifteen years ago. Among the key early influences were the randomized primality test developed by Solovay and Strassen [45] and a paper by Rabin [37] which drew attention to the general concept of a randomized algorithm and gave several nice applications to number theory and computational geometry. Also noteworthy is an early paper by Gill [19] which laid the foundations for the extension of abstract computational complexity theory to include randomized algorithms.

By now it is recognized that, in a wide range of applications, randomization is an extremely important tool for the construction of algorithms. There are two principal types of advantages that randomized algorithms often have. First, often the execution time or space requirement of a randomized algorithm is smaller than that of the best deterministic algorithm that we know of for the same problem. But even more strikingly, if we look at the various randomized algorithms that have been invented, we find that invariably they are extremely simple to understand and to implement; often, the introduction of randomization suffices to convert a simple and naive deterministic algorithm with bad worst-case behavior into a randomized algorithm that performs well with high probability on every possible input.

In the course of these lectures we will touch on a wide range of areas of application for randomized algorithms. We will discuss randomized algorithms in number theory and algebra, randomized algorithms for pattern matching, sorting and searching, randomized algorithms in computational geometry, graph theory and data structure maintenance, and randomized techniques in combinatorial enumeration and distributed computing. This means that there will be many kinds of mathematics involved, but we will omit hard proofs and we will only draw on elementary mathematical methods.

The unifying theme of these lectures will be the fact that a handful of basic principles underly the construction of randomized algorithms, in spite of the wide variety of their application. These principles will become more meaningful as we progress through the lectures, but let us mention some of them as a preview.

Abundance of witnesses. Randomized algorithms often involve deciding whether the input data to a problem possesses a certain property; for example, whether an integer can be factored. Often, it is possible to establish the property by finding a certain object called a *witness*. While it may be hard to find a witness deterministically, it is often possible to show that witnesses are quite abundant in a certain probability space, and thus one can search efficiently for a witness by repeatedly sampling from the probability space. If the property holds, then a witness is very likely to be found within a few trials; thus, the failure of the algorithm to find a witness in a long series of trials gives strong circumstantial evidence, but not absolute proof, that the input does not have the required property.

Foiling the adversary. A game-theoretic view is often useful in understanding the advantages of a randomized algorithm. One can think of the computational complexity of a problem as the value of certain zero-sum two-person game in which one of the players is choosing the algorithm and the other player, often called the *adversary*, is choosing the input data to foil the algorithm. The adversary's payoff is the running time of the algorithm on the input data chosen by the adversary. A randomized algorithm can be viewed as a probability distribution over deterministic algorithms, and thus as a mixed strategy for the player choosing the algorithm. Playing a mixed strategy creates uncertainty as to what the algorithm will actually do on a given input, and thus makes it difficult for the adversary to choose an input that will create difficulties for the algorithm.

Fingerprinting. This is a technique for representing a large data object by a short ''fingerprint'' computed for it. Under certain conditions, the fact that two objects have the same fingerprint is strong evidence that they are in fact identical. We will see applications of fingerprinting to pattern matching problems.

Checking identities. It is often possible to check whether an algebraic expression is identically equal to zero by plugging in random values for the variables, and checking whether the expression evaluates to zero. If a nonzero value ever occurs, then the expression is not an identity; if the value zero occurs repeatedly, then one has strong evidence that the expression is identically zero.

Random sampling, ordering and partitioning. Randomized algorithms for tasks such as sorting and selection gather information about the distribution of their input data by drawing random samples. For certain problems it is useful to randomize the order in which the input data is considered; in such cases, one can show that, for every fixed array of input data, almost all orderings of the data lead to acceptable performance, even though some orderings may cause the algorithm to fail. In a similar way, randomized divide-and-conquer algorithms are often based on random partitioning of the input.

Rapidly mixing Markov chains. Several randomized algorithms for the approximate solution of combinatorial enumeration problems are based on the ability to sample randomly from a large, structured set of combinatorial objects. The sampling process is based on a Markov chain whose states correspond to these combinatorial objects. A crucial issue in such cases is to show that the Markov chain converges rapidly to its stationary distribution.

Load balancing. In the context of distributed computation, randomization is often used to even out the assignment of computational tasks to the various processing units.

Symmetry breaking. In distributed computation, it is often necessary for a collection of computational processes to collectively make an arbitrary but consistent choice from a set of essentially indistinguishable possibilities. In such cases, randomization is useful for breaking symmetry.

In addition to describing and illustrating how the above ideas are used in the construction of randomized algorithms, we shall briefly discuss some general concepts related to randomized algorithms. Among these are the following.

Randomized complexity classes. Within computational complexity theory there has been an effort to study the class of problems solvable in polynomial time by randomized algorithms. Because of the probabilistic nature of the performance guarantees for randomized algorithms, several quite different reasonable definitions of this class have been proposed.

Interactive proofs. In an interactive proof, a prover demonstrates that a theorem is true by performing some task that would be impossible to perform if the theorem were false. Usually this task consists of giving the correct answers to a sequence of questions drawn at random from some set.

Randomness as a computational resource. In practice, randomized algorithms do not have access to an unlimited supply of independent, unbiased random bits. Physical sources of randomness tend to produce correlated sequences of bits. There is a large body of current research concerned with using imperfect sources of randomness and with stretching short random bit strings into much longer strings which, although not random, cannot easily be distinguished from truly random strings by computational tests.

Eliminating randomization. One way to obtain a deterministic algorithm is to first construct a randomized algorithm and then eliminate the randomization. We shall briefly touch on methods for accomplishing this.

3. Number theory

During the 1970s, a number of powerful randomized algorithms were discovered in the field of number theory. These algorithms were an important early stimulus to the study of randomized algorithms. In order to describe the properties of some of these algorithms, we will need a little elementary number theory.

Let p be a prime and let $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$. $\mathbb{Z}_p^*$ is a cyclic group under the operation of multiplication modulo p. There are low-degree polynomial-time algorithms to compute the inverse a^{-1} of an element $a \in \mathbb{Z}_p^*$ (this can be done by the Euclidean algorithm) and powers a^t of an element in $\mathbb{Z}_p^*$ (by successive squaring). An element $a \in \mathbb{Z}_p^*$ is said to be a *quadratic residue* if it is a perfect square in $\mathbb{Z}_p^*$, i.e., if there exists $z \in \mathbb{Z}_p^*$ such that $z^2 = a$. The *Legendre symbol* (a/p) of an element $a \in \mathbb{Z}_p^*$ is an indicator of whether the element is a perfect square. It is defined as 1 if a is a quadratic residue and -1 otherwise. The Legendre symbol has the multiplicative property $(a_1 a_2/p) = (a_1/p)(a_2/p)$. It also follows from the fact that $\mathbb{Z}_p^*$

is a cyclic group that $(a/p) = a^{(p-1)/2}$. In particular, this shows that the Legendre symbol is easy to compute.

3.1. Square roots modulo p

One of the earliest randomized algorithms in number theory was for finding a square root of $a \in \mathbb{Z}_p^*$, given that a is a quadratic residue. There is an explicit formula for the square root when $p = 3 \pmod 4$. The following algorithm for the general case has been attributed to D.H. Lehmer. It also emerges as a special case of Berlekamp's algorithm [7] for factoring polynomials with coefficients in $\mathbb{Z}_p$.

Suppose we know that a is a quadratic residue in $\mathbb{Z}_p^*$ and we want to find its square roots. In other words, we want to factor the polynomial $x^2 - a$ over $\mathbb{Z}_p^*$. It is sufficient, instead of factoring $x^2 - a$, to obtain a factorization of the polynomial $(x-c)^2 - a$, for some $c \in \mathbb{Z}_p^*$, since this amounts to simply shifting the roots of $x^2 - a$. Thus, suppose $(x-c)^2 - a = (x-r)(x-s)$. Then $rs = c^2 - a$, and $(r/p)(s/p) = ((c^2 - a)/p)$. If, upon choosing c and computing $((c^2-a)/p)$, it turns out that $((c^2-a)/p)$ is not 1, then we know from the multiplicative property of the Legendre symbol that exactly one of r and s is a quadratic residue. On the other hand, the quadratic residues in $\mathbb{Z}_p^*$ are the roots of the polynomial $x^{(p-1)/2} - 1$; hence, the greatest common divisor of $(x-c)^2 - a$ and $x^{(p-1)/2} - 1$ is a first-degree polynomial whose root is that root of $(x-c)^2 - a$ which is a quadratic residue. So we choose c randomly, check whether $c^2 - a$ is a quadratic residue and, if it is not, then easy computations will yield a root of $(x-c)^2 - a$ from which we can obtain $\sqrt{a}$. These ideas lead to the following algorithm.

Algorithm 3.1 (Finding square roots in $\mathbb{Z}_p^*$).

- Choose c at random from $\mathbb{Z}_p^*$;
- **if** $((c^2-a)/p) = -1$ **then** compute $\gcd(x^{(p-1)/2} - 1, (x-c)^2 - a)$; the result is $\alpha x - \beta$ and a zero of $(x-c)^2 - a$ is $r = \alpha^{-1}\beta$; return $\sqrt{a} = \pm(c + r)$.

The fundamental question is: how abundant are those elements c such that $c^2 - a$ is a quadratic nonresidue? It can be proven that more than half the elements of $\mathbb{Z}_p^*$ have this property:

Theorem 3.2. *Given a quadratic residue $a \in \mathbb{Z}_p^*$, if c is chosen at random from $\mathbb{Z}_p^*$ then, with probability larger than $\frac{1}{2}$, we have $((c^2-a)/p) = -1$.*

This is an example of a randomized algorithm that depends on the abundance of witnesses. It is a *Las Vegas algorithm*; i.e., it provides a solution with probability larger than $\frac{1}{2}$ and never gives an incorrect solution. Often we have to settle for a weaker result: a *Monte Carlo algorithm*. The concept of a Monte Carlo algorithm applies in situations where the algorithm makes a decision or a classification, and its output is either *yes* or *no*. A Monte Carlo algorithm is a randomized algorithm such that, if the answer is *yes*, then it confirms it with probability larger than $\frac{1}{2}$, but if the answer is *no*, then it simply remains silent. Thus, on an input for which the answer is *no*, the algorithm will never give a definitive result; however, its failure to give a *yes* answer in a long series of trials gives strong circumstantial evidence that the correct answer is *no*. We will shortly see a Monte Carlo algorithm for testing whether an integer is composite. The class of decision problems for which polynomial-time Monte Carlo algorithms exist is called RP. The class for which polynomial-time Las Vegas algorithms exist is called ZPP. It is easy to see that $\mathrm{ZPP} = \mathrm{RP} \cap \text{co-RP}$, where a language is in co-RP if its complement is in RP.

We should point out that there are other kinds of randomized algorithms that make errors: they give an incorrect answer with no indication that the answer is wrong. Such an algorithm is said to be a *bounded-error randomized algorithm* if there exists a constant $\varepsilon > 0$ such that, on every input, the probability with which it gives a correct answer is at least $\frac{1}{2} + \varepsilon$. A bounded-error randomized algorithm is quite useful because if, say, $\varepsilon = .1$, then, every time the algorithm is run the answer provided is correct with probability at least .6, and this probability can be amplified at will by running the algorithm repeatedly and taking the most frequent answer. On the other hand, an *unbounded-error randomized algorithm* is one that gives the correct answer with probability which is greater than $\frac{1}{2}$, but is not bounded away from $\frac{1}{2}$ by any fixed amount. In this case there is no statistical method of using repeated trials in order to get high confidence in the answer. This makes unbounded-error algorithms rather impractical, but they have been the object of some nice theoretical studies.

3.2. *Monte Carlo test for compositeness*

The 1970s produced two famous polynomial-time Monte Carlo algorithms for testing whether a given integer is composite [45,37]. We shall present the one due to Solovay and Strassen. Given a positive integer n, not necessarily prime, let $\mathbb{Z}_n^* = \{a \mid a \in \{1, 2, \ldots, n-1\}$ and $\gcd(a, n) = 1\}$. This set forms a group under multiplication modulo n, and we have the *Jacobi symbol*, which generalizes the Legendre symbol: if n is prime, then the Jacobi and Legendre symbols agree and we have $(a/n) = a^{(n-1)/2}$; generally, if $p_1, p_2, \ldots, p_k$ are primes and $n = p_1 p_2 \cdots p_k$, then the Jacobi symbol is $(a/n) = (a/p_1)(a/p_2) \cdots (a/p_k)$. It turns out, somewhat surprisingly, that the Jacobi symbol (a/n) is easy to compute. Using Gauss' law of quadratic reciprocity, one obtains a fast algorithm which resembles the Euclidean algorithm and does not require that the prime factorization of n be known.

Solovay and Strassen discovered a way to use the Jacobi symbol to obtain an exceedingly simple Monte Carlo test for compositeness.

Algorithm 3.3 (Test if a positive integer n is composite).

- Choose a at random from $\{1, 2, \ldots, n-1\}$;

- **if** $\gcd(a,n)\neq 1$ **then** return *composite*
  else if $(a/n)\neq a^{(n-1)/2} \pmod n$ **then** report *composite*.

Let us say that a is a *witness* to the compositeness of n if, when the algorithm receives n as input and chooses a at random, it determines that n is composite. Then the effectiveness of the algorithm depends on the abundance of witnesses. If n is composite, then it is easy to see that the elements $a\in\mathbb{Z}_n^*$ that are not witnesses, i.e., those that satisfy $(a/n)=a^{(n-1)/2} \pmod n$, form a subgroup of $\mathbb{Z}_n^*$. Moreover, it can be shown that the nonwitnesses form a proper subgroup. Since the order of a subgroup divides the order of the group, the order of a proper subgroup is at most half the order of the group. Thus,

Theorem 3.4. *If n is composite, then witnesses to its compositeness are abundant.*

This Monte Carlo algorithm never gives an unambiguous report that n is prime. Rather, it keeps looking for witnesses to the compositeness of n, and if n is prime the algorithm remains perpetually silent. On the other hand, if n is composite, then the chance that the algorithm would fail to report compositeness within 100 trials is less than 2^{-100}. Perhaps in some contexts such a failure to report compositeness is a sufficiently convincing argument that n is prime, even if it does not provide a mathematically rigorous proof of the primality of n. So we see again that Monte Carlo algorithms provide a proof in one direction, but only circumstantial evidence in the other direction.

Within the past few years, the work of Adleman and Huang [2] and others has led to a fast Las Vegas algorithm for primality testing. This algorithm uses the Monte Carlo algorithm for compositeness, together with a rather complicated Monte Carlo test of primality. The Las Vegas algorithm alternates between running these two Monte Carlo algorithms. Whether n is prime or composite, one of the two Monte Carlo algorithms will (with probability 1) eventually produce a witness, thus determining the status of n; and the expected time to find a witness is polynomial bounded.

4. Randomized equality testing

4.1. Testing polynomial identities

A very important idea which is often attributed to Schwartz [40] or R. Zippel, but has been rediscovered many times, is the use of randomized algorithms for testing polynomial identities.

Here is an example: the formula for computing a Vandermonde determinant. Let $x_1,x_2,\ldots,x_n$ be variables, and let M be the $n\times n$ matrix whose $(i-j)$th element is x_i^{j-1}. Then the following is an identity:

$$\det(M)-\prod_{i>j}(x_i-x_j)=0.$$

If, instead of proving this identity, we just wanted to verify it for the case $n=6$, we might do the following: repeatedly plug in numerical values for the variables, and evaluate the left-hand side. If we ever obtained a nonzero value, we would be able to conclude that the identity is false. If we kept getting the value zero, we would not be able to conclude that the identity is correct, but it seems that our confidence in its correctness would increase with each evaluation.

In the context of randomized computation, there is some foundation for this viewpoint, namely

Theorem 4.1. *Let $f(x_1, x_2, \ldots, x_n)$ be a multivariate polynomial of degree d. If f is not identically zero and if values $a_1, a_2, \ldots, a_n$ are drawn independently from the uniform distribution over $\{0, \pm 1, \pm 2, \ldots, \pm d\}$, then* $\Pr[f(a_1, a_2, \ldots, a_n) = 0] < \frac{1}{2}$.

This theorem, which can be proved easily by induction on the number of variables, says that, if the polynomial is not identically zero, then, if we keep choosing independent random samples from a suitably large finite domain and substitute them for the variables, there is a very tiny chance that the polynomial will repeatedly take on the value zero. Thus we have an efficient Monte Carlo algorithm for testing the property that a polynomial is not identically zero. One of the uses of this technique occurs in graph theory in connection with the problem of determining whether a given graph has a perfect matching.

4.2. Testing whether a graph has a perfect matching

A *perfect matching* in a graph is a set of edges which covers each vertex exactly once. Figure 1 shows two graphs, one which has a perfect matching and another which does not have a perfect matching.

The following theorem due to Tutte [46] gives a necessary and sufficient determinantal condition for a graph to have a perfect matching.

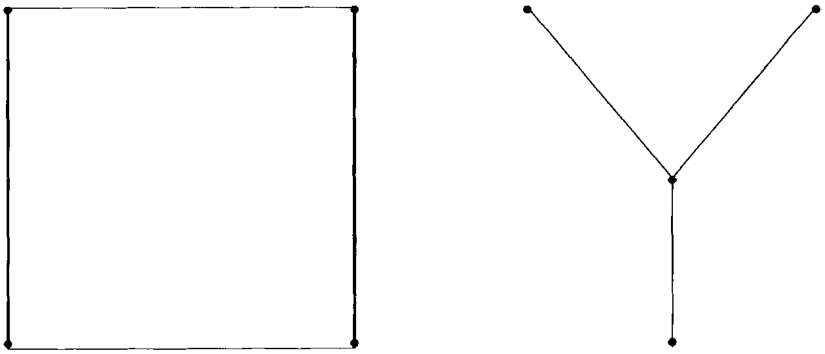

Fig. 1.

Theorem 4.2. *Let G be a graph with vertex set $\{1,2,\dots,n\}$, and let $A=(a_{ij})$ be the skew-symmetric $n\times n$ matrix defined as follows in terms of the indeterminates x_{ij}: $a_{ij}=0$ if i and j are not adjacent; $a_{ij}=x_{ij}$ if i and j are adjacent and $i<j$; $a_{ij}=-x_{ji}$ if i and j are adjacent and $i>j$. Then G has a perfect matching if and only if* $\det(A)\not\equiv 0$.

In view of our discussion of testing polynomial identities, Tutte's theorem is the basis of a randomized algorithm for testing whether a graph has a perfect matching. This was first observed by Lóvász.

Algorithm 4.3 (Testing for the existence of a perfect matching in a graph on n vertices).

- Form a matrix C by substituting $x_{ij}=c_{ij}$ in Tutte's symbolic matrix, where the c_{ij} are independent random variables uniformly distributed over the range $\{0,\pm 1,\pm 2,\dots,\pm n\}$;
- evaluate $\det(C)$;
- **if** $\det(C)\neq 0$ **then** report "*perfect matching exists*".

Since a numerical determinant can be computed in time $O(n^3)$ by Gaussian elimination, this algorithm is a polynomial-time Monte Carlo algorithm for testing whether a perfect matching exists. This is of somewhat academic interest, since Micali and Vazirani [32,47] have given a deterministic algorithm (much more complicated than the randomized algorithm given here) which tests for the existence of a perfect matching in time $O(m\sqrt{n})$, where m is the number of edges in the graph; the Micali–Vazirani algorithm is based on the augmenting path methods initiated by Berge [6] and Edmonds [18]. However, the above algorithm can also be implemented in parallel within the same resource bounds required for the evaluation of determinants: time $O(\log^2 n)$ using $O(n^{3.5})$ processors. This is noteworthy, since no deterministic parallel algorithm is known which tests for the existence of a perfect matching in polylog parallel time using a polynomial-bounded number of processors.

Instead of merely testing whether a perfect matching exists, suppose we would like to construct a perfect matching in parallel very rapidly. Is there some single randomized computation that will simultaneously identify all the edges in a perfect matching? This question was first resolved in [28]. We shall present the remarkably elegant solution to this problem by Mulmuley, Vazirani and Vazirani [36].

4.3. Fast parallel algorithm to find a perfect matching

The algorithm is based on the following probabilistic lemma which at first sight seems very surprising because of its great generality.

Lemma 4.4 (Isolation lemma). *Let $S_1,S_2,\dots,S_t$ be distinct subsets of*

$S=\{1,2,\ldots,N\}$. Let $w_1, w_2, \ldots, w_N$ be independent random weights assigned to the elements of S, drawn from the uniform distribution over $\{0, \pm 1, \pm 2, \ldots, \pm N\}$. By the weight of a subset of S we mean the sum of the weights of its elements. Then, with probability larger than $\frac{1}{2}$, there is a unique subset of minimum weight.

Proof. Suppose that two subsets, S_i and S_j, achieve the minimum weight. Then there is some element $x \in S$ that lies in one of these sets but not in the other. Thus, the minimum weight among the subsets containing x is equal to the minimum weight among the subsets not containing x. But the probability of this event is at most $1/(2N+1)$, since for each assignment of weights to the elements of $S-\{x]$ there is only one choice of a weight for x that will equate the minimum weight for the subsets not containing x with the minimum weight for the subsets containing x. Since x may be chosen in N ways, the probability of having two or more minimum-weight sets is at most $N/(2N+1)<\frac{1}{2}$. □

In order to apply this result to matchings, let G be a graph on the vertex set $\{1,2,\ldots,n\}$ with edge set S. We assign weights to the edges of G, the weights being independent random variables drawn from the uniform distribution over $\{0, \pm 1, \pm 2, \ldots, \pm[n^2/2]+1\}$. For the subsets S_i we take the sets of edges which are perfect matchings. Then the Isolation lemma guarantees that, with probability larger than $\frac{1}{2}$, there is a unique perfect matching of minimum weight.

The observation of Mulmuley, Vazirani and Vazirani is that, when a unique perfect matching M of minimum weight exists, a simple calculation based on determinants will identify in parallel all the edges of M. This is done as follows:

- Let w_{ij} be the weight of edge $i-j$;
- from the Tutte matrix, form the numerical matrix B by letting $x_{ij}=2^{w_{ij}}$;
- in parallel, for all edges $\{i,j\}$, perform the following test to determine whether $\{i,j\}$ lies in M: Compute

$$t_{ij}=\frac{\det(B_{ij})2^{w_{ij}}}{2^{2W}}$$

where B_{ij} is the ij-minor of the matrix B. The edge $\{i,j\}$ lies in M if and only if t_{ij} is an odd integer.

- Check that the edges determined to lie in M form a perfect matching. This check is necessary because of the possibility that the perfect matching of minimum weight is not unique.

The key observation is that each nonzero term in the expansion of $\det(B)$ corresponds to a perfect matching, and is equal to $\pm 2^{2\omega}$, where ω is the weight of the corresponding matching. It follows that, if W is the weight of the unique minimum-weight perfect matching, then 2^{2W} is the highest power of 2 that divides $\det(B)$. Thus the value of W can be determined easy from that of $\det(B)$.

The determinant of B and all the minors B_{ij} can be calculated in parallel in $O(\log^2 n)$ time using a polynomial-bounded number of processors. Thus we have a fast and reasonably efficient randomized parallel algorithm for constructing a perfect matching in a graph.

5. Testing equality of long strings by fingerprinting

We turn now to an application with a number-theoretic flavor which was first discussed by the Latvian mathematician Freivalds, who was one of the first to investigate the power of randomized algorithms. Suppose that two parties, Alice (A) and Bob (B) can communicate over a reliable but very costly communication channel. Suppose Alice has a very long string of bits x, Bob has a very long string of bits y, and they want to determine whether $x=y$. For example, we can think of Alice and Bob as having two versions of a manuscript and wanting to determine whether they are equal. An obvious way for them to test whether $x=y$ would be for Alice to send x across the channel to Bob; Bob could then compare x with y and let Alice know whether they are equal. But this brute-force solution would be extremely expensive, in view of the cost of using the channel. Another possibility would be for Alice to derive from x a much shorter string that could serve as a ''fingerprint'' of x; this short string might be obtained by applying some standard hashing or check sum technique. Alice could then send this fingerprint to Bob, who could determine whether the fingerprints of x and y were equal. If the fingerprints were unequal, then Bob would notify Alice that $x \neq y$. If the fingerprints were equal, then Bob would assume that $x=y$ and so notify Alice. This method requires the transmission of much less data across the channel, but permits the possibility of a *false match*, in which x and y have the same fingerprint even though they are not the same string.

In order to apply this idea Alice and Bob must agree on the fingerprinting function to be used. For example, they could choose a prime p and then use reduction modulo p as the fingerprinting function:

$$H_p(x) = H(x) \pmod{p}$$

where $H(x)$ is the integer represented by the bit string x. If p is not too large, then the fingerprint $H_p(x)$ can be transmitted as a short string of bits. This leads to the following algorithm.

Algorithm 5.1 (Testing equality of long strings by fingerprinting—prime p chosen in advance).

- A sends $H_p(x)$;
- B checks if $H_p(x) = H_p(y)$;
- B confirms that $x=y$ if $H_p(x) = H_p(y)$ and that $x \neq y$ otherwise.

The weakness of this method is that, if p is held fixed, then there are certain pairs of strings x and y on which the method will always fail, no matter how many times it is repeated. A more advantageous method, which avoids such bad pairs x and y, is to determine the prime p by randomization every time the equality of two strings is to be checked, rather than to agree on p in advance. This leads to the following algorithm, which is an improvement over the previous one.

Algorithm 5.2 (Testing equality of long strings by fingerprinting—randomized choice of the prime p).

- A draws p at random from the set of primes less than a certain value M;
- A sends p and $H_p(x)$;
- B checks whether $H_p(x) = H_p(y)$ and confirms the equality or inequality of the strings x and y.

The advantage of this second method is that if the prime p is chosen from a suitably large range, then for *any* strings x and y the probability that the algorithm will fail is extremely small; moreover, the probability of failure can be reduced even further by repeating the algorithm with several independent random choices of p. To make this precise, let $\pi(N)$ be the number of primes less than N. This function is a familiar and well-studied object in number theory and it is a fact that $\pi(N)$ is asymptotic to $N/\ln N$. It is also known that, if $A < 2^n$, then, except when n is very small, the number of primes that divide A is less than $\pi(n)$. Now let us compute the probability that the second algorithm will fail for two n-bit strings x and y. Failure can occur only in the case of a false match: i.e., $x \neq y$ but $H_p(x) = H_p(y)$. This is only possible if p divides $|H(x) - H(y)|$, an integer which is less than 2^n. Hence,

$$\Pr[\text{failure}] = \frac{|\{p \mid p<M,\ p \text{ is prime, } p \text{ divides } |H(x)-H(y)|\}|}{\pi(M)} \leq \frac{\pi(n)}{\pi(M)}.$$

In practice this second algorithm works very well. For example, if the strings x and y are 100,000 bits long and if Alice uses a fingerprint of at most 32 bits long, then, by substituting $n = 100{,}000$, $M = 2^{32}$ into the above formula, we find that, using Algorithm 5.2, the probability of a false match is less than 10^{-4}.

5.1. Pattern matching in strings

This is a classical problem in computer science to which fingerprinting can be applied. The problem is to determine whether or not a certain short pattern occurs in a long text. Any modern text processing system must provide the capability of performing such searches.

The most naive method for solving this problem is simply to move the short pattern across the entire text, and in every position make a brute-force comparison, character by character, between the symbols in the pattern and the corresponding symbols in the text. This is a quadratic method: its worst-case running time is proportional to nm, where n is the length of the pattern and m is the length of the text.

More complicated approaches using pointer structures lead to deterministic methods that run in time $O(n+m)$.

Here we will present a simple and efficient randomized method due to Karp and Rabin [27]. The method follows the brute-force approach of sliding the pattern $X=x_1x_2...x_n$ across the text $Y=y_1y_2...y_n$, but instead of comparing the pattern with each block $Y(i)=y_iy_{i+1}...y_{i+n-1}$ of the text, we will compare the fingerprint $H_p(X)$ of the pattern with the fingerprints $H_p(Y(i))$ of the blocks of text. These fingerprints are fortunately easy to compute. The key observation is that when we shift from one block of text to the next, the fingerprint of the new block $Y(i+1)$ can be computed easily from the fingerprint of $Y(i)$ using the following formula:

$$H_p(Y(i+1))=H_p(Y(i))+H_p(Y(i))-2^n y_i+y_{i+n} \pmod p.$$

Algorithm 5.3 (Pattern matching in strings).

```
• Choose $p$ at random from $\{q \mid 1\le q\le m^2n,\ q \text{ prime}\}$;
• MATCH $\leftarrow$ FALSE; $i\leftarrow 1$;
• while MATCH = FALSE and $1\le i\le m-n+1$ do
    if $H_p(X)=H_p(Y(i))$
    then MATCH $\leftarrow$ TRUE
    else $i\leftarrow i+1$; compute $Y(i+1)$.
```

Since the updating of the fingerprint requires a fixed number of arithmetic operations modulo p, the algorithm is essentially a real-time algorithm.

Now we need to analyze the frequency with which this algorithm will fail. A false match will be reported only if for some i we have $X\neq Y(i)$ but $H_p(X)=H_p(Y(i))$. This is only possible if p is a divisor of $\prod_{\{i \mid X\neq Y(i)\}} |H(X)-H(Y(i))|$. This product does not exceed 2^{mn}, and hence the number of primes that divide it does not exceed $\pi(mn)$. Consequently, the probability of a false match does not exceed $\pi(mn)/\pi(m^2n)\approx 2/m$. By way of a numerical example, if we have a text of length $m=4000$ bits and a pattern of length $n=250$ bits, we have $m^2n=4\times 10^9<2^{32}$. We can use a 32-bit fingerprint and the probability of a false match will be about 10^{-3}. So this randomized algorithm is an extremely practical method for performing pattern matching in a simple manner with a very small probability of error.

6. Selection, sorting and searching

We turn now to randomized algorithms in the core computer science areas of selection, searching and sorting. Many of the basic ideas of randomization were discovered and applied quite early in the context of these problems. We will see examples of how random sampling or random partitioning can be used effectively in algorithm design. Let us start with the classical problem of finding the median of a set of integers.

6.1. Finding the median

Let $\hat{x}$ be the nth smallest element of the set $X=\{x_1,x_2,\ldots,x_{2n-1}\}$. There are fairly complicated linear-time algorithms for finding $\hat{x}$. We will describe a simple randomized algorithm due to Floyd reported in [30]. It is based on the idea of taking a random sample of elements of X in order to determine an interval within which the median is likely to lie, and then discarding the elements that lie outside that interval, thereby reducing the size of the problem.

Using our coin-tossing capability, we pick from X a small random sample $\hat{X}$ (a good choice for the sample size is $2n^{2/3}$). Within this sample we pick two elements a and b which are not too far from the median of the sample, but far enough away that, with high probability, the median of the overall set will lie between a and b. Specifically, among the $2n^{2/3}$ elements that were chosen randomly, we pick the elements whose ranks in the ordering are $n^{2/3}-n^{1/3}\ln n$ and $n^{2/3}+n^{1/3}\ln n$. A straightforward argument shows that, with high probability, the proportion of elements from the overall set that lie in the interval $[a,b]$ will be quite similar to the proportion of elements in the random sample $\hat{X}$ which lie in this interval; specifically, it will be true with high probability that $|X\cap[a,b]|<2n^{2/3}\ln n$. Secondly, it will be true with high probability that $\hat{x}\in[a,b]$. Now the algorithm is quite obvious.

Algorithm 6.1 (Median-finding).

- Draw from X a random sample of size $2n^{2/3}$;
- sort $\hat{X}$ to determine the interval $[a,b]$;
- compare each element of X with a, and then, if necessary, with b, to determine whether the element lies in $[a,b]$; in this process, keep count of the number of elements less than a, in order to determine the rank of $\hat{x}$ in the set $X\cap[a,b]$ (for simplicity we neglect the extremely unlikely possibility that $\hat{x}$ does not lie in $[a,b]$);
- determine $\hat{x}$ by sorting the set $X\cap[a,b]$.

The execution time of the algorithm is dominated by the step in which each element is compared with a and possibly b. The expected number of elements compared with b is $n/2+\mathrm{o}(n)$, and thus the expected execution time of the entire algorithm is $3n/2+\mathrm{o}(n)$. This extremely simple randomized algorithm compares favorably with the deterministic median-finding algorithms, thus demonstrating in a simple context the power of random sampling.

6.2. Quicksort with random partitioning

Random partitioning is an important tool for the construction of randomized divide-and-conquer algorithms. The classic algorithm that uses random partitioning is a variant of the famous sorting algorithm *Quicksort*. Quicksort with random partitioning can be described very simply. To sort a set of elements $X=\{x_1,x_2,\ldots,x_n\}$ the algorithm proceeds as follows:

Algorithm 6.2 (Quicksort with random partitioning).

- Draw an element x^* at random from the set X; call x^* the *splitter*;
- compare each element with x^*, thus partitioning the remaining elements into two sets: SMALL $= \{x \in X \mid x < x^*\}$ and LARGE $= \{x \in X \mid x^* < x\}$;
- recursively, sort the sets SMALL and LARGE.

Of course, what is desired is a splitter that will divide X into two sets of approximately equal size. Although the random choice of a splitter does not guarantee such a division, it can be shown, using a straightforward analysis based on a recurrence relation, that the expected execution time of Quicksort when all splitters are chosen at random is $2n \ln n + O(n)$. This performance compares fairly well with the standard information-theoretic lower bound $n \log_2 n$ for the number of comparisons needed to sort n items.

6.3. Binary search trees

Similar ideas can be applied to problems concerning data structures. One of the most basic data structures is a *dictionary*. A dictionary is intended to include items from a linearly ordered set such as the integers or the words over an alphabet, and to support the operations of accessing, inserting or deleting an element. Often further operations are supported, such as joining two dictionaries together, splitting a dictionary in certain ways or finding the least element greater than a given element x.

One of the most common ways of maintaining a dictionary is through the use of a *binary search tree*. This is a binary tree whose internal nodes correspond to the items in the dictionary. It is based on the ordering principle that, at each node, the items in the left subtree precede, and the items in the right subtree follow, the item at the node. This ordering principle enables the search for an item or the insertion of an item to be accomplished by following a single path through the tree, starting at the root; the operation of deletion is only slightly more complicated.

There is an interesting parallel between the randomized quicksort algorithm described above and the behavior of binary search trees if the items are inserted in a random order. It can be shown quite easily that the number of comparisons needed in the randomized quicksort algorithm has the same probability distribution as the number of comparisons needed to insert n items into an initially empty binary search tree in random order. The reason for this is that we can view a binary search tree in two ways: as a tree resulting from insertions or as a depiction of the successive splittings used in Quicksort. This dual interpretation of binary search trees allows the transfer of the analysis of randomized Quicksort to yield the result that the expected insertion time or the expected time to access a random item is logarithmic if the items are inserted in random order.

Since we cannot assume that the insertions are made in random order, there is a real possibility that a binary search tree may perform in a catastrophic manner. The worst case occurs when the sequence of insertions produces a "linear" tree, in

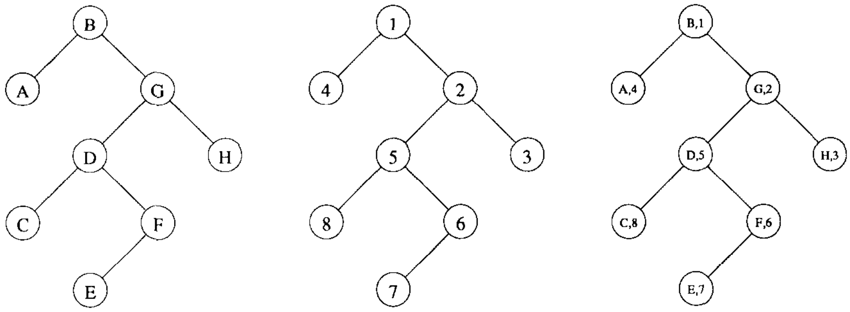

Fig. 2. Binary search tree, heap and treap.

which all the items lie along a single long path. In this case, the insertion and access times are linear in n, rather than logarithmic. There are various standard ways to restructure binary search trees in the process of performing operations on them so as to guarantee that all of the execution times are logarithmic. In particular, AVL trees [1] and splay trees [44] achieve this effect.

Here we will describe a very recent randomized approach to the maintenance of binary search trees using a new data structure due to Aragon and Seidel [4]. The structure is called a treap because it combines the ideas of binary search tree and heap. A *heap* is a binary tree with the following ordering principle: along any root-to-leaf path, the values of the items increase; thus, the value of any parent is less than the values of its children and, in particular, the smallest element appears at the root.

A *treap* is a binary tree in which each item x has two associated values, x. key and x. priority, and which is simultaneously a binary search tree with respect to the key values and a heap with respect to the priority values (see Fig. 2). Given n items with associated key and priority values, it is easy to show that there is a unique tree structure for the corresponding treap, namely the tree structure obtained by inserting the keys in increasing order of priorities.

The algorithms for maintaining a treap are slightly complicated, because, in order to maintain the binary search tree property and the heap property simultaneously, a certain amount of local restructuring is sometimes necessary. For example, the insertion of item (D,28) in the treap of Fig. 3(a) results in the tree shown in Fig. 3(b), which fails to be a heap with respect to the priority values. This requires a local rotation, and the treap properties are reestablished in Fig. 3(c).

Aragon and Seidel give a clever application of the concept of treap to the problem of maintaining binary search trees. The idea is to use a binary search tree to maintain a dictionary, but to use randomization in the following way: when an item is inserted, we draw a value from a continuous probability distribution and assign it to the item as its priority. This number provides a "random time stamp" for the item, and we require that the tree be a heap with respect to the random priorities. This has an interesting effect which facilitates the analysis of the expected time to

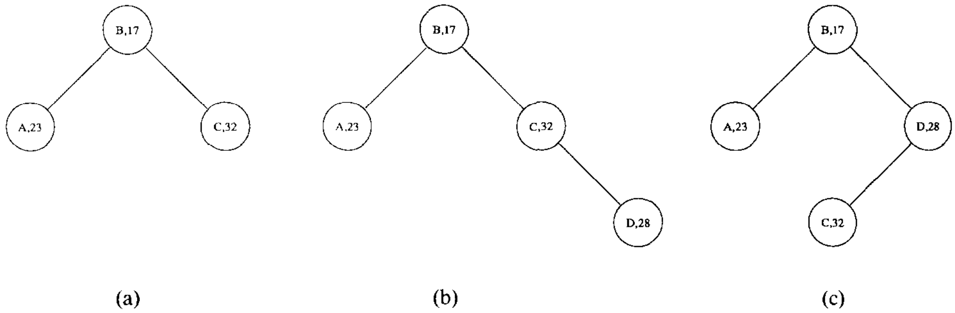

Fig. 3. Three binary trees.

perform data structure operations. We have mentioned that for a given set of items having keys and priorities, there is a unique treap containing them. Suppose that we have performed a long sequence of insertions, deletions and accesses. A snapshot of the treap structure at any one time is independent of any items that were inserted and later deleted and is uniquely determined by the two fields of the items currently present in the treap. Therefore, the snapshot at any point in time is exactly the tree that would have been obtained if those items had been inserted into an initially empty binary search tree in the order determined by their time-stamp priorities. Since the time stamps are completely independent and random, this means that at any fixed moment what we have statistically is a binary search tree obtained by random insertions. Thus, although the insertions were not performed in random order, the assignment of random priorities achieves an equivalent effect and we can apply the results regarding randomized Quicksort. To complete the analysis, it is also necessary to consider the cost of the rotations required to maintain the treap property; fortunately, Aragon and Seidel have shown that the expected number of rotations per insertion or deletion is less than 2. Thus, for a tree of size N occurring at any point in the history of a treap, the expected time for the next access, insertion or deletion is $O(\log N)$. Thus the approach based on random priorities is an elegant and efficient way to maintain binary search trees.

7. Computational geometry

About fifteen years ago computational geometry emerged as an important new area within the study of algorithms and their complexity. Since computational geometry deals in large part with data structures containing items with several keys (where each key typically corresponds to a coordinate in a d-dimensional Euclidean space) it is natural that many of the ideas developed in the context of sorting and searching carry over to computational geometry, where some of the combinatorial flavor is replaced by a geometric point of view. In particular, there has recently been high interest in randomized algorithms in computational geometry, and very

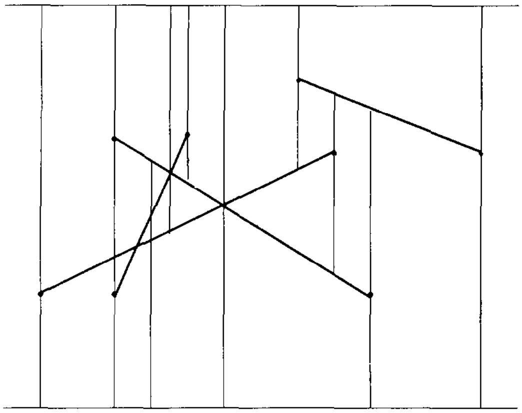

Fig. 4. A trapezoidal diagram.

beautiful and practical results have been obtained. We will describe some of these results.

7.1. Finding the planar partition determined by a set of line segments

Suppose we are given a collection of line segments within a two-dimensional box. The *planar partition* associated with these segments is obtained by running, from each endpoint of a segment, and from each point where two segments intersect, a vertical line that continues, both upward and downward, until it intersects the boundary of the box or another segment. The result is a dissection of the box into trapezoids, and thus the planar partition is sometimes called the *trapezoidal diagram* associated with the set of line segments. Figure 4 shows the trapezoidal diagram associated with a set of four line segments.

The problem of computing an explicit description of the trapezoidal diagram associated with a set of line segments is a classical one in computational geometry. The problem requires in particular the determination of all intersections of the given segments. Mulmuley [33] found a fast randomized algorithm for constructing the trapezoidal diagram. His algorithm is based on the introduction of randomization into a naive deterministic algorithm for the problem, and he proved that, through the use of randomization, the algorithm achieves a very favorable expected running time. The algorithm starts with no segments, and with a vertical line, extending from the top to the bottom of the bounding box, through each endpoint of the desired segments. It then adds segments, one at a time, in random order, updating the trapezoidal diagram after the addition of each segment. Each time a segment is added, the algorithm must trace along the length of that segment to find the vertical lines that insersect it and contract those vertical lines appropriately. It must also find any new segment intersections and create appropriate new vertical lines.

Figure 5 illustrates the process. It shows the diagram at the stage when the dotted

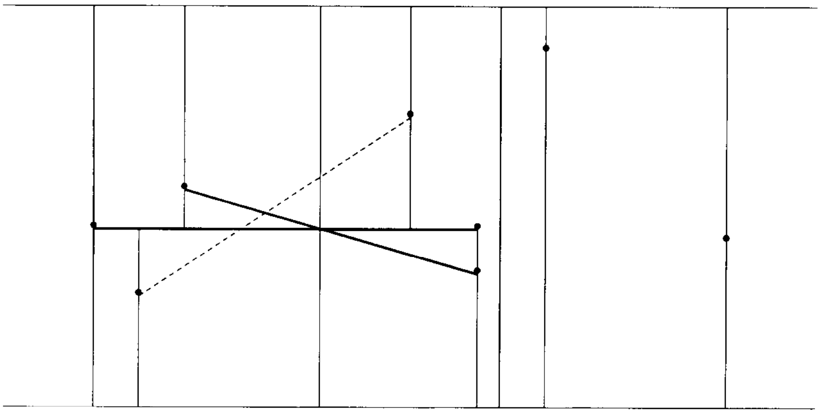

Fig. 5. Step in the construction of a trapezoidal diagram.

line segment is added. The segment is traced and, starting at its right endpoint, the algorithm encounters a vertical line whose upper part will have to be deleted from the structure, and two successive intersections with other segments through which new vertical lines must be added.

Mulmuley's insight is that if the segments are added in random order, then the solution tends to be obtained very rapidly. If the number of segments is n and the number of intersections is m, then the expected time to execute this algorithm is $O(m+n\log n)$. As usual, the expectation is over the random choices made by the algorithm, and the bound on expected time is valid for every instance of the problem. This bound matches the performance of the best deterministic algorithm for this problem [10], even though the deterministic algorithm is much more complicated.

In measuring the work performed by Mulmuley's algorithm, it is necessary to specify in some detail the data structures that will be maintained and to determine the expected contribution of various operations to the expected running time. For simplicity, we will concentrate on just one part of the analysis, the determination of the expected number of contractions of vertical segments. Consider an endpoint q and a vertical ray upward from q. Let U_q be the number of times this ray gets contracted. Note that U_q is a random variable, since it depends on the order in which the segments are added. If there are t segments crossing this ray, then the number of contractions may be as large as t if the segments are added in an unfortunate order. However, a segment L causes the line to be contracted only if L's intersection with the ray is lower than that of any segment processed before L. Thus, if L's intersection with the ray is the kth lowest, then the probability that L causes a contraction is $1/k$. It follows that the expected number of contractions is

$$E[U_q]=\sum_{i=1}^{t}\frac{1}{i}\sim\ln t.$$

7.2. Linear programming with a fixed number of variables

This is another elegant example of the use of randomization in computational geometry. It is due to Clarkson [12]. The linear programming problem is, of course, to minimize a linear objective function $c \cdot x$ subject to a system of linear inequalities $Ax \leq b$. The data of the problem consists of the d-dimensional vector c, the n-dimensional vector b and the $n \times d$ matrix A. The solution is a d-dimensional vector x.

Clarkson's randomized method is effective when d, the number of variables, is very small compared to n, the number of constraints. It exploits the basic fact that the optimal solution to a d-dimensional linear programming problem is determined by d of its n constraints; if the other $n-d$ constraints were deleted from the problem, the optimal solution would be unchanged. Clarkson's idea is to use random sampling to avoid considering irrelevant constraints.

Let S be the set of constraints of the problem. Let T be a subset of S and let $x^*(T)$ be the solution which is optimal when we consider only the constraints in T (for ease of exposition we ignore the possibility that a subset of constraints may fail to have a bounded optimal solution). As the algorithm proceeds, it accrues a set V^* of constraints that will be enforced at all times. The goal of the algorithm is to keep this set small while capturing all d of the constraints that determine the optimal solution of the problem.

Algorithm 7.1 (Linear programming with a fixed number of variables).

```
• (Initialization) $V^* \leftarrow \Phi$;
• Repeat
  - Choose at random a set $R \subset S$ of $d\sqrt{n}$ constraints (recall that $n$ is much larger
    than $d$);
  - solve, for example by the simplex method, the linear program with (small)
    constraint set $V^* \cup R$, obtaining a solution $x^*(V^* \cup R)$;
  - $x^* \leftarrow x^*(V^* \cup R)$;
  - inspect all constraints and determine the set $V$ of constraints that are violated
    by $x^*$;
  - if $V = \Phi$ then return $x^*$ and halt (all constraints are satisfied, and the answer
    to the problem is reported);
  - if $|V| \leq 2\sqrt{n}$ then $V^* \leftarrow V^* \cup V$ (the small set of currently violated constraints
    is added to the set of enforced constraints and the computation continues).
```

Let us examine the performance of this algorithm. The optimal solution of the problem is determined by some set S^* of d constraints. The algorithm will succeed as soon as S^* is captured in the set of enforced constraints. Each time we solve a linear program with some set of constraints and find that V, the set of violated constraints, is nonempty, at least one constraint from S^* is among the violated ones.

Thus, when we add V to the set of enforced constraints, we capture at least one constraint from S^*. It follows that the set of enforced constraints will be augmented at most d times. Also, the size of the set of enforced constraints does not grow too large, since we add only $2\sqrt{n}$ constraints at a time. However, at each iteration in which there are more than $2\sqrt{n}$ violated constraints the algorithm fails to augment the set of enforced constraints, and thus fails to make progress. The crux of the analysis of the algorithm is to show that such useless iterations will not be too frequent. A probabilistic argument shows that, at each iteration, the probability that the number of violated constraints will exceed $2\sqrt{n}$ is less than $\frac{1}{2}$. It follows that the expected number of iterations is less than $2d$. At each iteration a linear program with at most $3d\sqrt{n}$ constraints gets solved. It follows that the expected execution time of the entire algorithm is

$$\mathrm{O}(d^2n) + \lg\left(\frac{n}{d^2}\right)^{\lg d+2} \mathrm{O}(d^{d/2+\mathrm{O}(1)}).$$

If we think of d as fixed and n as growing, then the dominant term is $\mathrm{O}(d^2n)$. No deterministic algorithm for solving linear programs in a fixed number of dimensions is known to achieve as good a time bound.

The idea of using random sampling, random partitioning and random ordering in computational geometry has led to a large number of elegant and efficient randomized geometric algorithms. Among these are eminently practical algorithms for hidden surface removal, for the computation of convex hulls in three dimensions, and for the dual problem of computing the intersections of a set of hyperplanes [11-14,34,35,41].

8. Combinatorial enumeration problems

With every nondeterministic polynomial-time Turing machine M one may associate both a decision problem and an enumeration problem. The decision problem is to determine whether M, on a given input x, has an accepting computation. The enumeration problem is to determine the number of accepting computations of machine M on input x. The class of decision problems associated with nondeterministic polynomial-time Turing machines is called NP, and the class of enumeration problems associated with such machines is called #P. Thus, each problem in #P can be viewed as counting the witnesses to instances of a problem in NP.

Typical problems in #P include counting the perfect matchings, Hamiltonian circuits or spanning trees of a graph, counting the total orders compatible with a given partial order, and counting the truth-value assignments satisfying a propositional formula. Spanning trees can be counted in polynomial time, since the Kirchhoff Matrix-Tree Theorem tells us that the number of spanning trees is given by the determinant of a certain integer matrix associated with the given graph. The other problems appear to be much harder. In fact, the problems of counting perfect matchings, Hamiltonian circuits, compatible total orders, and satisfying assignments are #P-complete; this means that every problem in #P is polynomial-time reducible to each of the three. Thus, if any one of the three problems were solvable in polynomial time, then every problem in #P would be solvable in polynomial time; this is very unlikely to be true.

Recently there has been a wave of interest in polynomial-time randomized algorithms for the approximate solution of problems in #P. We will discuss some of the results that have been obtained.

8.1. Randomized approximation algorithms for combinatorial enumeration problems

The set-up is as follows: Let I denote an instance of a problem, and let COUNT(I) denote the number of solutions for instance I. For example, if the problem were to count perfect matchings, then I would be a graph and COUNT(I) would be the number of perfect matchings in that graph. Let $A(I)$ be the estimate of COUNT(I) produced by the randomized approximation algorithm I. For fixed positive constants ε and δ, algorithm A is called an *ε, δ-approximation algorithm* if, for every instance I, the probability that the relative error exceeds ε is less than δ; i.e.,

$$\Pr\left[\frac{|\mathrm{COUNT}(I)-A(I)|}{\mathrm{COUNT}(I)}>\varepsilon\right]<\delta.$$

Thus, ε can be viewed as an accuracy parameter and δ as a confidence parameter. Often, we are interested in a family $\{A_{\varepsilon,\delta}\}$ of related approximation algorithms where, for all $\varepsilon>0$ and $\delta>0$, $A_{\varepsilon,\delta}$ is an ε,δ-approximation algorithm. Such a family of approximation algorithms is called a *polynomial-time approximation scheme* if, for all ε and δ, the execution time of $A_{\varepsilon,\delta}$ is bounded by a polynomial in n (the size of the instance), ε^{-1} and $\ln(\delta^{-1})$. The logarithmic dependence on δ^{-1} is natural for the following reason: suppose we can achieve the desired performance for all ε when $\delta=\frac{1}{4}$; this means that every time we run the algorithm there is at most a 25% chance that our relative error will exceed ε. Now suppose we want to achieve a higher level of confidence, corresponding to a smaller value of δ. We can repeatedly run the algorithm that has a 25% chance of making a relative error larger than ε and then take, as our estimate of COUNT(I), the median of the estimates produced in the individual runs. A straightforward probabilistic calculation shows that the number of iterations of the algorithm that works for a confidence level of $\frac{1}{4}$ necessary for achieving a given confidence level δ grows as $\ln(\delta^{-1})$. Thus, for theoretical purposes, we may fix δ at $\frac{1}{4}$.

8.1.1. Estimating the cardinality of a union of sets

The problem of finding the cardinality of a union of sets is a classical combinatorial problem whose classical solution is given by the principle of inclusion and exclusion:

$$|S_1 \cup S_2 \cup \cdots \cup S_t| = \sum_i |S_i| - \sum_{i<j} |S_i \cap S_j| + \sum_{i<j<k} |S_i \cap S_j \cap S_k| - \cdots .$$

When t is large the direct evaluation of the inclusion-exclusion sum is not practical since the number of terms is $2^t - 1$. Furthermore, even though the inclusion-exclusion sum is bracketed between any two consecutive partial sums, the partial sums behave rather erratically and do not furnish a good approximation to the inclusion-exclusion sum.

Instead, let us consider a randomized method which produces an estimate of the cardinality of a union of sets. This method requires three assumptions: that we should be able to determine easily the cardinality of each set S_i, that we should be able to draw an element at random from the uniform distribution over any one of the sets, and that we should be able to test whether a given element lies in a given set. We will shortly see a concrete example where these conditions are fulfilled.

We will define the *coverage* of an element x as the number of sets that contain x: $\mathrm{cov}(x) = |\{i \mid x \in S_i\}|$. The algorithm produces an estimator X of the cardinality of $\bigcup S_i$ using a two-stage sampling process:

Algorithm 8.1 [25] (Estimating the cardinality of a union of sets).

- Draw a set at random from the distribution in which the probability of drawing S_i is proportional to its cardinality; i.e., $\Pr[S_i] = |S_i| / \sum_{j=1}^{t} |S_j|$;
- having drawn S_i, choose a random element x from S_i;
- by testing the membership of x in each S_j, determine $\mathrm{cov}(x)$;
- $X \leftarrow \sum_{i=1}^{t} |S_i| / \mathrm{cov}(x)$.

It is a simple exercise to show that X is an unbiased estimator of the cardinality of the union of sets: i.e., $E[X] = |\bigcup S_i|$. This suggests that we might estimate $|\bigcup S_i|$ by taking the average of N samples of the estimator X: $Y = (X_1 + X_2 + \cdots + X_N)/N$. We require that $\Pr[(|Y| - |\bigcup S_i|)/|\bigcup S_i| > \varepsilon] < \delta$. A routine calculation involving bounds on the tails of the binomial distribution shows that a sample size sufficient for this purpose is $N = t \ln(2/\delta) 4.5/\varepsilon^2$.

There are a number of concrete problems which can be expressed as computing the cardinality (or probability, or measure, or volume) of a union of sets, and are amenable to this approach. A number of these applications are in reliability theory, but the simplest example is estimating the number of truth assignments satisfying a Boolean formula in disjunctive normal form (DNF). In this case S_i is the set of truth assignments satisfying the ith term in the DNF formula. For example, if the formula is $x_1 x_2 \bar{x}_3 \vee \bar{x}_1 x_4 \vee x_3 x_4 \bar{x}_5 \vee \cdots$, then S_1 consists of all truth assignments in which x_1 is true, x_2 is true and x_3 is false. It is clear that the three assumptions required by the method are satisfied, and thus we get a polynomial-time approximation scheme for estimating the number of truth assignments satisfying a DNF Boolean formula. Even though the method is simple, this result is of interest because the problem of exactly counting the truth assignments satisfying a DNF formula is #P-complete. Note, however, that it is critical for the formula to be in disjunctive normal form; a moment's thought shows that, unless P = NP, there cannot exist a polynomial-time approximation scheme for the problem of counting the truth assignments satisfying a Boolean formula in conjunctive normal form.

8.1.2. Estimating the permanent of a 0–1 matrix

Another classical problem related to combinatorial enumeration is the computation of the permanent of a $n \times n$ matrix $A = (a_{ij})$. The *permanent* of A is defined as

$$\operatorname{per}(A) = \sum_{\sigma \in S_n} a_{1\sigma(1)} \cdot a_{2\sigma(2)} \cdot \cdots \cdot a_{n\sigma(n)}.$$

The problem of computing the permanent of a $n \times n$ 0–1 matrix is equivalent to the #P-complete problem of counting the perfect matchings in a (simple) bipartite graph with n vertices in each part.

Although the definition of the permanent resembles that of the determinant, the permanent seems to be much harder to evaluate; the best deterministic algorithm known is Ryser's algorithm [39] based on inclusion-exclusion, which runs in time $\Theta(n2^n)$. Many people have remarked that, since the determinant is easy to compute and has a definition resembling that of the permanent, there might be some way to use the ease of computing the determinant in a strategy for computing the permanent. We present here a randomized algorithm which exploits the following relation between permanent and determinant due to Godsil and Gutman:

> *Given a* 0–1 *matrix A, let B be the random matrix $(\pm a_{ij})$, where the plus and minus signs are chosen independently at random. Then* $E[\det^2(B)] = \operatorname{per}(A)$.

This suggests a Monte Carlo method in which one estimates $\operatorname{per}(A)$ as the mean of n independent samples, each of which is obtained by choosing random plus and minus signs to derive from A a random matrix B, and then computing $\det^2(B)$. This method will perform quite poorly on certain examples. For example, if A has 2×2 blocks of 1's on the main diagonal and 0's elsewhere, then the determinant of B will be zero whenever one of the 2×2 diagonal blocks has determinant zero. Each of the $n/2$ diagonal blocks independently has a 50% chance of having a zero determinant. Thus, $\Pr[\det^2(B) \neq 0] = 2^{-n/2}$. Therefore, a sample size around $2^{n/2}$ will be needed in order to have a reasonable chance of ever observing a nonzero determinant. Still, an analysis based on estimating the variance of $\det^2(B)$ and applying Chebyshev's inequality yields the following result, showing that the randomized method has some advantages in comparison with Ryser's deterministic algorithm, if one is willing to approximate the permanent rather than compute it exactly.

Theorem 8.2 [24]. *The number of trials needed for an ε, δ-approximation of the permanent of an $n \times n$* 0–1 *matrix is less than $C3^{n/2}\varepsilon^{-1}\ln(\delta^{-1})$, where C is a constant.*

The following refinement of the method reduces the variance of the estimator. Instead of multiplying the entries of A randomly by $+1$ and -1, use cube roots of unity: replace each entry a_{ij} independently and randomly by either a_{ij}, ωa_{ij} or $\omega^2 a_{ij}$, where ω is a principal cube root of 1. Let the resulting random complex matrix be C. Then $E[\det(C)\overline{\det}(C)] = \mathrm{per}(A)$, where $\bar{z}$ denotes the complex conjugate of the complex number z. Thus we can estimate $\mathrm{per}(A)$ by the mean of a number of samples, each of which is obtained by constructing from A the random complex matrix C, and then computing $\det(C)\overline{\det}(C)$.

Theorem 8.3 [24]. *The number of trials needed for an ε, δ-approximation to the permanent of an $n \times n$ 0–1 matrix is less than $C2^{n/2}\varepsilon^{-1}\ln(\delta^{-1})$.*

Although this Monte Carlo algorithm is an improvement over the best deterministic algorithm for computing the permanent of a 0–1 matrix, it requires exponential time. What is really wanted is a polynomial-time approximation scheme for the problem. This has not been achieved, but there exists a mathematically interesting approach which yields such an approximation scheme in certain special cases. In preparation for presenting this approach, we need to discuss random walks on multigraphs.

8.1.3. Random walk on a regular multigraph of degree d

We consider a random walk on a finite N-vertex multigraph which is regular of degree d. Loops and multiple edges are allowed; each loop at a vertex contributes 1 to its degree. The *random walk* associated with such a multigraph is defined as follows: when a vertex v_i is reached, the walk continues along a randomly chosen edge incident to v_i. This random walk determines a Markov chain whose states are the vertices of the graph. If there are a edges between v_i and v_j, then the transition probability associated with a transition from v_i to v_j is given by $p_{ij} = a/d$. In such a Markov chain, all states have the same stationary probability $1/N$.

We shall be interested in how rapidly the Markov chain associated with a regular multigraph approaches its stationary distribution. Let $p_{ij}^{(t)}$ be the probability that the state at time t is j, given that the state at time 0 is i. As a measure of how rapidly the Markov chain mixes, we introduce the quantity $\Delta(t) = \max_{i,j}[p_{ij}^{(t)}/(1/N)]$; this is a measure, in relative terms, of how close the distribution of states after t steps comes to the stationary distribution, when nothing is assumed about the initial state. A chain is said to be *rapidly mixing* if $\Delta(t)$ converges rapidly to 1. If a chain is rapidly mixing, then one can start in an arbitrary initial state and reach a nearly uniform distribution over the states after a small number of transitions.

We shall require sufficient conditions for a chain to be rapidly mixing. Jerrum and Sinclair [23] introduce the concept of the *conductance* of a Markov chain; the conductance of a chain (in the special case where the stationary distribution is uniform) is defined as the minimum, over all sets S of states containing at most half the states of the entire chain, of the conditional probability of escaping from S at the next step, given that the present state is uniformly distributed over S. Thus, the conductance is

$$\Phi = \min\left(\frac{1}{|S|}\sum_{\substack{i\in S\\ j\in \bar{S}}} p_{ij}\right),$$

where S ranges over all state sets of cardinality at most $n/2$.

If a chain has high conductance, then it will not tend to get trapped in small sets of states, and it will be rapidly mixing. Specifically, we obtain the following result due to Jerrum and Sinclair [23].

Theorem 8.4. $\Delta(t) \leq N(1-\Phi^2/2)^t$.

Jerrum and Sinclair [23] discovered the following useful technique for bounding the conductance of a Markov chain with uniform stationary distribution.

Theorem 8.5. *Suppose that one can specify, for each two-element set* $\{i,j\}$ *of states, a canonical simple path between i and j, such that no oriented edge appears in more than bN canonical simple paths. Then* $\Phi \geq 1/(2bd)$.

The application of random walks on multigraphs to combinatorial enumeration goes as follows. As a step towards approximately counting some set S of combinatorial objects, one often wishes to sample almost uniformly from some associated set T of combinatorial objects. To sample from T, one can set up a random walk on a regular multigraph with vertex set T; if the random walk is rapidly mixing, then the sampling problem can be solved by simulating the walk from an arbitrary initial state for a small number of steps, and then observing the state, which will be nearly uniformly distributed over T.

In the following sections we describe a very interesting application of the random walk method to the problem of estimating the number of perfect matchings in a bipartite graph. The approach was initiated by Broder [9] and the first rigorous proof of its efficiency in particular cases was given by Jerrum and Sinclair [23] using the theorems about conductance given above.

8.1.4. Estimating the number of perfect matchings in a bipartite graph

Given a bipartite graph with m edges and n vertices in each part, let M_k denote the number of matchings of size k. We are interested in estimating M_n, the number of perfect matchings. If all the ratios M_k/M_{k-1}, $k=2,3,\ldots,n$ were known, then M_n would be determined since M_1 is clearly equal to m. Broder's idea is to obtain sufficiently good estimates of these ratios that, by multiplying them together, we obtain a good estimate of M_n/M_1. Each ratio M_k/M_{k-1} can be estimated statistically by drawing a suitably large number of independent random samples from the uniform distribution (or from a nearly uniform distribution) over the set $M_k \cup M_{k-1}$, and for this purpose one defines a random walk on a regular multigraph with vertex set $M_k \cup M_{k-1}$. We illustrate the approach by defining a random walk over $M_n \cup M_{n-1}$, the set of perfect and "near-perfect" matchings in the graph.

In this case the state set is $M_n \cup M_{n-1}$ and there are m equally probable transitions out of each state, corresponding to the m edges of the graph. To complete the description, we need to describe the transition corresponding to edge e when the state is M. There are five cases:

- If M is a perfect matching and $e \notin M$, then the new state is M;
- if M is a perfect matching and $e \in M$, then the new state is $M - \{e\}$;
- if M is not a perfect matching and e is not incident with any vertex in M, then the new state is $M \cup \{e\}$;
- if M is not a perfect matching and e is incident with exactly one vertex covered by M, then there is a unique edge $f \in M$ such that $M \cup e - f$ is a matching, and the new state is $M \cup e - f$;
- if M is not a perfect matching and e is incident with two vertices covered by M, then the new matching is M.

This Markov chain corresponds to a random walk on a regular undirected graph of degree m with vertex set $M_n \cup M_{n-1}$. Jerrum and Sinclair [23] prove that, if the underlying n-vertex graph within which we are trying to count perfect matchings is *dense*, meaning that all degrees are greater than $n/2$, then the conductance of the Markov chain is at least $1/(12n^6)$. This establishes that one can sample from a distribution over $M_n \cup M_{n-1}$ which is exponentially close to the uniform distribution by simulating a polynomial-bounded number of steps of this Markov chain. Similar results hold for similarly defined Markov chains on $M_k \cup M_{k-1}$, for $k = 2, 3, \ldots, n-1$. It follows that, in the case of dense bipartite graphs, there is a polynomial-time randomized approximation scheme for the problem of counting perfect matchings. The paper [16] extends this result to a broader class of bipartite graphs. However, it remains an open question whether there exists a polynomial-time randomized approximation scheme for the problem of counting perfect matchings in an arbitrary bipartite graph.

8.1.5. Estimating the volume of a convex body

The idea of applying rapidly mixing Markov chains to combinatorial enumeration problems has been extremely influential, and has stimulated a great deal of research. One of the most striking recent results in this direction is due to Dyer, Frieze and Kannan [17], who gave a polynomial-time randomized approximation scheme for the problem of computing the volume of a convex body L. The principal input to the algorithm of Dyer, Frieze and Kannan is a *membership oracle*; i.e., a black box that will answer any query of the form "Does point x lie in L?". For technical reasons, two additional input items are required: a ball of positive volume contained in L and a ball containing L.

We shall not do more than sketch the approach taken by Dyer, Frieze and Kannan. The first step is to give a reduction showing that the problem of estimating the volume of L is polynomial-time reducible to the following problem: given a membership oracle for a convex body K, draw a point x from a "nearly uniform" distribution over K. In order to sample from K, they approximate K by a very slightly larger convex body which can be described as the union of a large number of congruent n-dimensional hypercubes. They then introduce a random walk on a regular graph of degree $2n$ whose vertices are the hypercubes. The edges of this graph incident with a given hypercube H are in one-to-one correspondence with the $2n$ $(n-1)$-dimensional facets of H. For a given facet F of H the transition is as follows: If H shares the facet F with another hypercube H', then the transition is to H', else the walk remains at H.

Let Φ be the conductance of this random walk. Using isoperimetric inequalities relating the volume and the surface area of smooth manifolds, it is possible to show that the reciprocal of Φ is bounded above by a polynomial in n. Hence the walk is rapidly mixing.

The existence of a polynomial-time randomized approximation scheme for the problem of computing the volume is surprising for several reasons:

- The problem of computing the volume exactly is #P-complete.
- According to a result in [5], any deterministic algorithm based on membership queries requires a number of queries exponential in n in order to guarantee an approximation to the volume with bounded relative error; thus randomization is essential for the result.
- The result implies a polynomial-time randomized approximation scheme for the seemingly difficult problem of computing the number of linear orderings compatible with a given partial ordering.

9. Randomization in parallel and distributed computation

In this section we briefly indicate some of the uses of randomization in parallel and distributed computing.

9.1. Dynamic task scheduling

The design of efficient parallel algorithms often entails decomposing a computation into smaller tasks and scheduling the execution of these tasks on individual processors. An ideal scheduling algorithm is one which keeps all the processors busy executing essential tasks, and minimizes the interprocessor communication required to determine the schedule and pass data between tasks. The scheduling problem is particularly challenging when the tasks are generated dynamically and unpredictably in the course of executing the algorithm. This is the case with many recursive divide-and-conquer algorithms, including backtrack search, game tree search and branch-and-bound computation.

Karp and Zhang [29,50] have shown that randomization is a powerful tool for solving dynamic task scheduling problems. We illustrate the approach with a simple model problem related to parallel backtrack search. In this case the set of tasks is a rooted binary tree whose shape is initially unknown to the parallel algorithm. Initially, only the root of the tree is given. The primitive unit-time computational step is called *node expansion*. The step of expanding a node x either determines that x is a leaf of the rooted tree or else creates the children of x. The goal is to expand every node of the tree, using p processors. In order to balance the workloads of the processors, it will also be necessary for nodes to be sent from one processor to another, and we assume that it takes one unit of time for a processor to send or receive a node.

If the tree contains n nodes, then n/p is clearly a lower bound on the time of any parallel algorithm with p processors. Also, if the maximum number of nodes on a root-leaf path is h, then h is a lower bound, since the tasks along a path must be executed sequentially. Thus, a randomized parallel algorithm that, with high probability, executes all tasks within time $O(n/p+h)$ may be considered optimal. Karp and Zhang have given an optimal algorithm which has the additional nice property that it is completely decentralized and requires no global control or global data structures. At any point in the execution of this algorithm, each node that has been created but not yet executed is assigned to a processor. A processor is called *idle* if no nodes are assigned to it, and *backlogged* if more than one node is assigned to it. Each step consists of three parts: first, each idle processor requests data from a randomly chosen processor; then, each backlogged processor which has received a request chooses a random requesting processor and donates its rightmost task (with respect to the left-to-right ordering implicit in the rooted tree); finally, each processor that is not idle executes one of its tasks. It can be proved that, with high probability, this simple randomized scheme yields an execution time bounded by a constant times the lower bound of $\max(n/p, h)$.

9.2. *Symmetry breaking in distributed computation*

Randomized algorithms have found many useful and elegant applications in the area of protocols for distributed systems. In distributed systems there are a large number of processes executing concurrently and asynchronously, each with only incomplete knowledge of what the other processes are doing. Many of the problems revolve around symmetry breaking; that is, the use of randomization to make a choice between two or more alternatives that look identical.

We will use a metaphor to describe one typical example of a distributed system problem that can be attacked using randomization. Suppose that a tour group is to gather at one of the two entrances to a railroad station. However, the station is so free of landmarks that there is no canonical way to distinguish between the two entrances: they cannot be referred to as ‘‘the entrance by the newsstand’’ or ‘‘the north entrance’’. However, near each entrance is a small bulletin board on which messages can be left. How can the tourists arrange to convene at a common entrance, when they may only communicate via the bulletin boards? This problem is called the *choice coordination problem.*

The problem arises in computer systems in various ways, for example, when a large group of users must reach agreement on which of two versions of a distributed data structure to use, even though they have no agreed-upon way of naming the two versions or distinguishing one from the other.

Rabin [38] has given an elegant and efficient randomized solution to this problem. The problem is defined abstractly as follows. There are n indistinguishable processes that must coordinate their choices. All processes are to execute the same algorithm. The computation is asynchronous; thus, at any point in the computation, any process that is ready to execute a step may be the next one to do so. There are two memory cells (corresponding to the bulletin boards associated with the two entrances to the station). The algorithm executed by each process consists of a sequence of *indivisible actions.* An indivisible action has two parts: reading one of the two memory cells and (optionally) writing a value back into the same cell. When the computation terminates, exactly one of the two memory cells is to contain the special value $\square$; the result of the choice coordination process is the selection of the cell that ultimately contains $\square$.

In Rabin's randomized protocol the contents of a cell is either the special symbol $\square$ or an ordered pair $[n, b]$, where n is a nonnegative integer and b is either 0 or 1. Initially, each cell contains the ordered pair $[0, 0]$. Each process executes the following brief but intricate algorithm, in which the variable m denotes a positive integer and r and t denote binary digits.

```
• $m \leftarrow 0$; $r \leftarrow$ random bit; $t \leftarrow 0$
• **repeat** the following primitive action **forever**
• $r \leftarrow 1 - r$; $x \leftarrow$ contents of cell $r$
  - *Case* 1: $x = \square$: halt;
  - *Case* 2: $x = [n, b]$
      (a) $m < n$ or ($m = n$ and $t < b$): $[m, t] \leftarrow [n, b]$;
      (b) $[m, t] = [n, b]$: $[m, t] \leftarrow$ contents of cell $r \leftarrow [m + 1$, random bit];
      (c) $m > n$ or ($m = n$ and $t > b$): contents of cell $r \leftarrow \square$; halt.
```

The reader is invited to prove that Rabin's protocol is correct, and that the following is true for every positive integer k: the probability that some process will execute more than k primitive actions before the protocol halts is not greater than 2^{-k}. Since the protocol is correct and tends to terminate rapidly, it constitutes a highly effective randomized solution to the choice coordination problem.

10. Interactive proofs

During the past few years complexity theorists have been intensely investigating a radical new concept of mathematical proof. In an *interactive proof* one demonstrates that a statement is true not by deriving it within a formal system of axioms and rules of inference, but by performing some feat that would not be possible unless the statement were true. The concept of interactive proof was first defined in [20].

An interactive proof is a dialogue conducted by two randomized algorithms, the *prover* P and the *verifier* V. The dialogue begins when both P and V receive the input x, and the object of the dialogue is to enable V to decide whether x has a certain property Π. In the course of conducting the protocol the two parties send messages back and forth. It is required that the total length of these messages be bounded by a polynomial in the length of the input x. It is also required that V be a polynomial-time randomized algorithm; P, on the other hand, is allowed to have unlimited computational power. The dialogue always ends with a decision by V as to whether x lies in L. Informally, the role of P is to persuade V that x has property Π, and the role of V is to put questions to P that P will be able to answer in a satisfactory way if and only if x actually does have property Π.

In order for the pair P, V to qualify as an interactive proof of membership in L, the following two properties must be satisfied:

- *completeness*: if x has property Π, then, with high probability, the dialogue between P and V will end in the acceptance of x;
- *soundness*: if x does not have property Π, then, even if P is replaced by some other randomized algorithm P^* (i.e., even if the prover cheats), the probability that the dialogue will end in acceptance of x is very small.

It is clear that every problem in NP has an extremely simple interactive proof in which both the prover and the verifier are deterministic. In the case where the input x has property Π, P simply sends a witness to this fact, and V verifies the validity of the witness. The ability of P and V to randomize, combined with the possibility of a multi-stage dialogue, permits the construction of interactive protocols for problems that appear not to lie in NP. In fact, Shamir [42] has shown that the collection of problems for which interactive protocols exist is precisely the class of problems solvable by Turing machines within polynomial space; this class is believed to be far more extensive than NP.

As an example, let us give an interactive protocol for the *graph nonisomorphism problem*, in which the input string represents a pair G, H of graphs, and Π is the property that G and H are not isomorphic. The dialogue consists of a sequence of rounds. In each round, V tosses a fair coin and, depending on the outcome of the coin toss, selects either G or H. Then, using randomization, V constructs a graph K that is isomorphic to the selected graph, but differs from the selected graph by having the names of the vertices randomly permuted. Then V sends K to P and challenges P to declare whether it is G or H that was selected and scrambled in order to produce K. In the case where G and H are not isomorphic, P gives the unique correct reply; otherwise he arbitrarily chooses G or H. The verifier V declares that G and H are nonisomorphic if and only if P gives the correct answer in each of a long series of rounds. If G and H are not isomorphic, then P, by virtue of his unlimited computing power, will always be able to determine whether K came from G or from H, and thus will be able to persuade V to accept G, H; this is the basis for the completeness of the protocol. If G and H are isomorphic, then P, being unaware of V's coin tosses, will be making a pure guess each time as to whether K comes from G or from H, and thus will be very unlikely to answer correctly every time; this is the basis for the soundness of the protocol.

The protocol for graph nonisomorphism has the further property of being a *zero-knowledge protocol*; this means that, in the course of demonstrating that G and H are nonisomorphic, P reveals no information about the two graphs except the fact that they are not isomorphic. This property is useful in the application of interactive protocols to various kinds of business transactions between mutually distrustful parties. For example, if the purpose of the protocol were for P to identify himself to V by proving that he possessed a certain piece of secret information (P's digital signature), it would be undesirable if the dialogue were to leak anything about the signature except its correctness, since V might be able to use the leaked information to guess the secret information and thereby impersonate P.

11. Randomness as a computational resource

We have seen that randomized algorithms are often simple, beautiful and efficient, but *is* there such a thing as a randomized algorithm? Do computers really have available a source of random bits? One approach, which has often been proposed but seldom put into practice, is to build into a computer a physical source of randomness based on shot noise or some other process that is random at the quantum-mechanical level. The more common approach in practice is to use a "random number generator"; i.e., an algorithm which starts with a short "seed" which is presumed to be random and produces from it a long bit string which has some of the mathematical properties that random strings are expected to have.

It is clear that random bits, if they can be produced at all, will be slow and costly to generate. For this reason, there has been considerable interest in reducing the number of truly random bits that algorithms require, or else showing that imperfect sources of randomness are adequate. We shall briefly describe some possible approaches.

11.1. Techniques for finding a witness

As we discussed earlier in the paper, a Monte Carlo algorithm receives, in addition to its input data x, a string y of random bits of some length n determined by the length of x. The string x is to be accepted if and only if at least one string y is a "witness" to the acceptance of x. The salient property of a Monte Carlo algorithm is that witnesses are either nonexistent or abundant; more precisely, if x is to be accepted, then at least half of the n-bit strings are witnesses.

We would like an algorithm that doesn't use too many random bits but has an extremely high probability of finding a witness whenever witnesses exist. One way is to keep generating random n-bit strings; at each trial, the probability of failing to find a witness is at most $\frac{1}{2}$. A more efficient approach, first proposed in [26,43], is based on a special type of bipartite graph called a disperser. A *(d, n, t)-disperser* is regular of degree d, has a vertex set consisting of two copies, A and B, of $\{0,1\}^n$, and has the property that, for every t-element set $T \subset A$, at least half the vertices in B are adjacent to at least one vertex in T. In order to find a witness, we can expend n bits to generate a random vertex $a \in A$ and then test for witnesshood all d of the n-bit strings in B that are adjacent to a. The probability that we will fail to find a witness in this way is at most $t2^{-n}$. It is possible to efficiently construct families of dispersers in which d is a polynomial in n and $t = 2^{cn}$, where c is a constant less than 1. This leads to randomized algorithms that use n random bits, for which the probability of failing to find a witness when one exists is exponentially small in n. The paper [15] is a recent survey covering the construction and application of dispersers. Other approaches to the efficient generation of witnesses involve random walks on expanders [3] and the use of universal families of hash functions [22]. The first of these approaches allows the error probability of a Monte Carlo algorithm to be reduced to 2^{-k} using $O(n+k)$ bits.

11.2. k-wise independent random variables

Certain randomized algorithms do not require a source of completely independent random bits; instead, it is sufficient that each individual bit shall have an equal chance of being 0 or 1, and that the stream of bits shall be *k-wise independent*. This latter property means that any k of the bits in the sequence are mutually independent. Because k-wise independence is often sufficient, considerable attention has been devoted to the generation of long k-wise independent bit strings from short strings of completely independent bits. One powerful approach is as follows. Let A be a $m \times n$ matrix over GF[2], the field with two elements, such that any k rows are linearly independent; the construction of such matrices is a well-studied central problem in the theory of error-correcting codes. Let b be a n-vector of completely independent random bits, each of which is equally likely to be 0 or 1. Then the m-vector Ab has the property that its components are k-wise independent, and that each component is equally likely to be 0 or 1. A related construction works over $\mathbb{Z}_p$, the integers modulo a prime p. In order to generate a sequence $b_0, b_1, \ldots, b_{p-1}$ of p k-wise independent elements, each of which is uniformly distributed over $\mathbb{Z}_p$, we start with a seed $a_0, a_1, \ldots, a_{k-1}$ consisting of k mutually independent elements, each of which has the uniform distribution over $\mathbb{Z}_p$, and generate the desired sequence according to the formula

$$b_i = a_0 + a_1 i + \cdots + a_{k-1} i^{k-1}.$$

Thus, the elements of the seed are used as the coefficients of a polynomial, and the long sequence is formed by the values of the polynomial at $i=0,1,\ldots,p-1$.

In cases where pairwise independence is sufficient, it is often possible to eliminate randomization altogether. For example, Luby [31] has given a randomized parallel algorithm for constructing a maximal independent set of vertices in a graph. The algorithm requires a sequence of p pairwise independent elements, each drawn from the uniform distribution over $\mathbb{Z}_p$. Such a sequence can be formed from a seed consisting of two elements of $\mathbb{Z}_p$. An alternate interpretation is that we are working with a very small probability space whose points are the p^2 possible seeds, rather than the very large probability space that would be required if we were generating completely independent random elements of $\mathbb{Z}_p$. Therefore, we can simply enumerate all the possible seeds (a_0, a_1) and run the algorithm for each one. By the same argument that justifies the correctness of the original randomized algorithm, most of these choices have to work. Hence, at the cost of some additional computation, randomness can be eliminated entirely.

11.3. Imperfect sources of randomness

One difficulty with the available physical sources of randomness is that they generate correlated sequences of bits, rather than the completely independent random bits that one would ideally want. This difficulty has motivated Vazirani and Vazirani [48] to investigate the power of algorithms based on imperfect sources of randomness. They define a *slightly random source* as one which satisfies the following very weak requirement: at any step, the conditional probability that the next bit will be 0, given the entire past sequence of bits, lies between a and $1-a$, where a is some fixed positive constant. Vazirani and Vazirani show that, if a problem can be solved by a polynomial-time Monte Carlo algorithm using an ideal source of random bits, then the problem can also be solved using an arbitrary slightly random source.

11.4. Pseudo-random number generators

A *pseudo-random number generator* [8,49] is defined as a parametrized sequence of functions $g=\{g_n\}$, such that each function $g_n:\{0,1\}^n \rightarrow \{0,1\}^{t(n)}$ takes a seed consisting of n bits and "stretches" that seed, by a deterministic process, to a longer string of length $t(n)$. For example, we may have $t(n)=n^2$. We say that such a sequence of functions is a pseudo-random number generator if no test that can be implemented in polynomial time, even with the help of true randomness, can distinguish the outputs of the generator from a random sequence. This property has the consequence that, in any polynomial-time randomized algorithm, the output of a pseudo-random number generator can safely be used in place of an ideal source of random bits.

It turns out, somewhat surprising, that there is a profound connection between the concept of a pseudo-random number generator and the concept of a one-way function, which is of central importance in cryptography. A *one-way function* is, roughly speaking, a function that is easy to compute but hard to invert. More specifically, a *one-way permutation* is a sequence $f=\{f_n\}$, where each f_n is a one-to-one function from $\{0,1\}^n$ onto itself, such that $f(x)$ can be computed in polynomial time, but no randomized polynomial-time algorithm has a significant chance of computing the preimage $f^{-1}(y)$ for a randomly chosen element y.

It is not known whether one-way functions exist. If $\mathrm{P}=\mathrm{NP}$, then they definitely do not exist. There are, however, a number of seemingly intractable problems in number theory which have been conjectured to give rise to one-way permutations. One example is the *discrete logarithm problem*. Let p be a prime and let g be a generator of $\mathbb{Z}_p^*$. Let $f(x)=g^x$. Then f is easy to compute but its inverse function, the discrete logarithm, seems intractable.

It has recently been proven [21] that every one-way function can be used to construct a pseudo-random number generator. In the case where the one-way function is a permutation f, the construction of the generator is particularly simple. The seed consists of two strings, $x\in\{0,1\}^n$ and $y\in\{0,1\}^n$, and the output of the generator is obtained by iterating f on x and computing the scalar product (mod 2) of each iterate with y: $x\cdot y, f(x)\cdot y, f(f(x))\cdot y, \ldots, f^{[k]}(x)\cdot y, \ldots$.

References

[1] G.M. Adelson-Velskiĭ and E.M. Landis, Dokl. Akad. Nauk SSSR 146 (1962) 263-266; also: Soviet Math. 3 (1962) 1259-1263 (English translation).

[2] Adleman and M.A. Huang, Recognizing primes in random polynomial time, Tech. Rep., Department of Computer Science, University of Southern California, Los Angeles, CA (1988).

[3] M. Ajtai, J. Komlós and E. Szemerédi, Deterministic simulation in LOGSPACE, in: Proceedings of the Nineteenth ACM Symposium on Theory of Computing (STOC) (1987) 132.

[4] C. Aragon and R. Seidel, Randomized search trees, in: Proceedings of the Thirtieth Symposium on Foundations of Computer Science (FOCS) (1989) 540-545.

[5] I. Bárány and Z. Füredi, Computing the volume is difficult, in: Proceedings of the Eighteenth ACM Symposium on Theory of Computing (STOC) (1986) 442-447.

[6] C. Berge, Two theorems in graph theory, Proc. Nat. Acad. Sci. 43 (1957) 842-844.

[7] E.R. Berlekamp, Factoring polynomials over large finite fields, Math. Comp. 24 (1970) 713-735.

[8] M. Blum and S. Micali, How to generate cryptographically strong sequences of pseudo-random bits, SIAM J. Comput. 13 (1984) 850-864.

[9] A.Z. Broder, How hard is it to marry at random? (On the approximation of the permanent), in: Proceedings of the Eighteenth ACM Symposium on Theory of Computing (STOC) (1986) 50-58.

[10] B. Chazelle and H. Edelsbrunner, An optimal algorithm for intersecting line segments in the plane, in: Proceedings of the Twenty-Ninth Symposium on Foundations of Computer Science (FOCS) (1988) 590-600.

[11] K.L. Clarkson, New applications of random sampling in computational geometry, Discrete Comput. Geom. 2 (1987) 195-222.

[12] K.L. Clarkson, A Las Vegas algorithm for linear programming when the dimension is small, in: Proceedings of the Twenty-Ninth Symposium on Foundations of Computer Science (FOCS) (1988) 452-457.

[13] K.L. Clarkson and P. Shor, Applications of random sampling in computational geometry II, Discrete Comput. Geom. 4 (1989) 387–421.

[14] K.L. Clarkson, R.E. Tarjan and C.J. Van Wyk, A fast Las Vegas algorithm for triangulating a simple polygon, Discrete Comput. Geom. 4 (1989) 423–432.

[15] A. Cohen and A. Wigderson, Manuscript, Hebrew University (1989).

[16] P. Dagum, M. Luby, M. Mihail and U. Vazirani, Polytopes, permanents and graphs with large factors, in: Proceedings of the Twenty-Ninth Symposium on Foundations of Computer Science (FOCS) (1988) 412–421.

[17] M. Dyer, A. Frieze and R. Kannan, A random polynomial time algorithm for approximating the volume of convex bodies, in: Proceedings of the Twenty-First ACM Symposium on Theory of Computing (STOC) (1989) 375–381.

[18] J. Edmonds, Paths, trees and flowers, J. Res. Nat. Bur. Standards 17 (1965) 449–467.

[19] J. Gill, Computational complexity of probabilistic Turing machines, SIAM J. Comput. 6 (1977) 675–695.

[20] S. Goldwasser, S. Micali and C. Rackoff, The knowledge complexity of interactive proof systems, SIAM J. Comput. 18 (1989) 186–208.

[21] R. Impagliazzo, L.A. Levin and M. Luby, Pseudorandom generation from one-way functions, in: Proceedings of the Twenty-First ACM Symposium on Theory of Computing (STOC) (1989) 12–24.

[22] R. Impagliazzo and D. Zuckerman, How to recycle random bits, in: Proceedings of the Thirtieth Symposium on Foundations of Computer Science (FOCS) (1989) 248–253.

[23] M. Jerrum and A. Sinclair, Conductance and the rapid mixing property for Markov chains: The approximation of the permanent resolved, in: Proceedings of the Eighteenth ACM Symposium on Theory of Computing (STOC) (1986) 235–243.

[24] N. Karmarkar, R. Karp, R. Lipton, L. Lovász and M. Luby, A Monte Carlo algorithm for estimating the permanent, SIAM J. Comput., to appear.

[25] R.M. Karp and M. Luby, Monte Carlo algorithms for the planar multiterminal network reliability problem, J. Complexity 1 (1985) 45–64.

[26] R.M. Karp, N. Pippenger and M. Sipser, A time-randomness trade-off, AMS Conference on Probabilistic Computational Complexity, Durham, NH (1985).

[27] R.M. Karp and M. Rabin, Efficient randomized pattern-matching algorithms, IBM J. Res. Develop. 31 (1987) 249–260.

[28] R. Karp, E. Upfal and A. Wigderson, Constructing a perfect matching in random NC, Combinatorica 6 (1986) 35–48.

[29] R.M. Karp and Y. Zhang, A randomized parallel branch-and-bound procedure, in: Proceedings of the Twentieth ACM Symposium on Theory of Computing (STOC) (1988) 290–300.

[30] D.E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching (Addison-Wesley, Menlo Park, CA, 1973) 217–220.

[31] M. Luby, A simple parallel algorithm for the maximal independent set problem, SIAM J. Comput. 15 (1986) 1036–1053.

[32] S. Micali and V.V. Vazirani, An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs, in: Proceedings of the Twenty-First Symposium on Foundations of Computer Science (FOCS) (1980) 17–27.

[33] K. Mulmuley, A fast planar partition algorithm, I, in: Proceedings of the Twenty-Ninth Symposium on Foundations of Computer Science (FOCS) (1988) 580–589.

[34] K. Mulmuley, An efficient algorithm for hidden surface removal, in: Proceedings ACM SIGGRAPH (1989) 379–388.

[35] K. Mulmuley, A fast planar partition algorithm, II, in: Proceedings of the Fifth ACM Symposium on Computational Geometry (1989) 33–43.

[36] K. Mulmuley, U.V. Vazirani and V.V. Vazirani, Matching is as easy as matrix inversion, Combinatorica 7 (1987) 105–113.

[37] M.O. Rabin, Probabilistic algorithms, in: J. Traub, ed., Algorithms and Complexity (Academic Press, New York, 1976).

[38] M.O. Rabin, The choice coordination problem, Acta Inform. 17 (1982) 121–134.

[39] H. Ryser, Combinatorial mathematics, in: Carus Mathematical Monographs 14 (Math. Assoc. America, Washington, DC, 1963) 26–28.

[40] J.T. Schwartz, Fast probabilistic algorithms for verification of polynomial identities, J. ACM 27 (1980) 701–717.

[41] R. Seidel, Linear programming and convex hulls made easy, in: Proceedings of the Sixth ACM Symposium on Computational Geometry (1990).

[42] A. Shamir, IP = PSPACE, in: Proceedings of the Twenty-Second ACM Symposium on the Theory of Computing (STOC) (1990) 11–15.

[43] M. Sipser, Expanders, randomness or time vs. space, in: Structure in Complexity Theory (Springer, Berlin, 1986) 325.

[44] D.D. Sleator and R.E. Tarjan, Self-adjusting binary search trees, J. ACM 26 (1985) 652–686.

[45] R. Solovay and V. Strassen, A fast Monte-Carlo test for primality, SIAM J. Comput. 6 (1977) 84–85.

[46] W.T. Tutte, The factorization of linear graphs, J. London Math. Soc. 22 (1947) 107–111.

[47] V.V. Vazirani, A theory of alternating paths and blossoms for proving correctness of the $O(\sqrt{VE})$ general graph matching algorithm, #89-1035, Department of Computer Science, Cornell University, Ithaca, NY (1989).

[48] V.V. Vazirani and U.V. Vazirani, Random polynomial time is equal to semi-random polynomial time, in: Proceedings of the Twenty-Sixth Symposium on Foundations of Computer Science (FOCS) (1985) 417–428.

[49] A.C. Yao, Theory and applications of trapdoor functions, in: Proceedings of the Twenty-Third Symposium on Foundations of Computer Science (FOCS) (1982) 80–91.

[50] Y. Zhang, Parallel algorithms for combinatorial search problems, Ph.D. Thesis, University of California, Berkeley, CA (1989).