performance. The randomized sorting algorithm described above is an example. This book presents many other randomized algorithms that enjoy these advantages.

In the next few sections, we will illustrate some basic ideas from probability theory using simple applications to randomized algorithms. The reader wishing to review some of the background material on the analysis of algorithms or on elementary probability theory is referred to the Appendices.

1.1. A Min-Cut Algorithm

Two events $\mathcal{E}_1$ and $\mathcal{E}_2$ are said to be *independent* if the probability that they both occur is given by

$$\mathbf{Pr}[\mathcal{E}_1 \cap \mathcal{E}_2] = \mathbf{Pr}[\mathcal{E}_1] \times \mathbf{Pr}[\mathcal{E}_2] \tag{1.4}$$

(see Appendix C). In the more general case where $\mathcal{E}_1$ and $\mathcal{E}_2$ are not necessarily independent,

$$\mathbf{Pr}[\mathcal{E}_1 \cap \mathcal{E}_2] = \mathbf{Pr}[\mathcal{E}_1 \mid \mathcal{E}_2] \times \mathbf{Pr}[\mathcal{E}_2] = \mathbf{Pr}[\mathcal{E}_2 \mid \mathcal{E}_1] \times \mathbf{Pr}[\mathcal{E}_1], \tag{1.5}$$

where $\mathbf{Pr}[\mathcal{E}_1 \mid \mathcal{E}_2]$ denotes the *conditional probability* of $\mathcal{E}_1$ given $\mathcal{E}_2$. Sometimes, when a collection of events is not independent, a convenient method for computing the probability of their intersection is to use the following generalization of (1.5).

$$\mathbf{Pr}[\cap_{i=1}^{k} \mathcal{E}_i] = \mathbf{Pr}[\mathcal{E}_1] \times \mathbf{Pr}[\mathcal{E}_2 \mid \mathcal{E}_1] \times \mathbf{Pr}[\mathcal{E}_3 \mid \mathcal{E}_1 \cap \mathcal{E}_2] \cdots \mathbf{Pr}[\mathcal{E}_k \mid \cap_{i=1}^{k-1} \mathcal{E}_i]. \tag{1.6}$$

Consider a graph-theoretic example. Let G be a connected, undirected multigraph with n vertices. A *multigraph* may contain multiple edges between any pair of vertices. A *cut* in G is a set of edges whose removal results in G being broken into two or more components. A *min-cut* is a cut of minimum cardinality. We now study a simple algorithm for finding a min-cut of a graph.

We repeat the following step: pick an edge uniformly at random and merge the two vertices at its end-points (Figure 1.1). If as a result there are several edges between some pairs of (newly formed) vertices, retain them all. Edges between vertices that are merged are removed, so that there are never any self-loops. We refer to this process of merging the two end-points of an edge into a single vertex as the *contraction* of that edge. With each contraction, the number of vertices of G decreases by one. The crucial observation is that an edge contraction does not reduce the min-cut size in G. This is because every cut in the graph at any intermediate stage is a cut in the original graph. The algorithm continues the contraction process until only two vertices remain; at this point, the set of edges between these two vertices is a cut in G and is output as a candidate min-cut.

Does this algorithm always find a min-cut? Let us analyze its behavior after first reviewing some elementary definitions from graph theory.

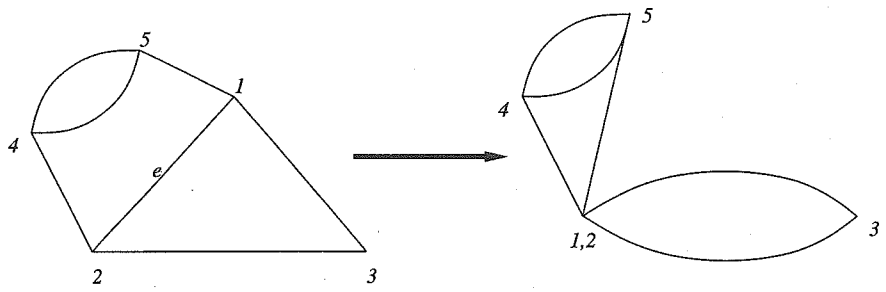

Figure 1.1: A step in the min-cut algorithm; the effect of contracting edge $e = (1,2)$ is shown.

► **Definition 1.1:** For any vertex v in a multigraph G, the *neighborhood* of v, denoted $\Gamma(v)$, is the set of vertices of G that are adjacent to v. The *degree* of v, denoted $d(v)$, is the number of edges incident on v. For a set S of vertices of G, the neighborhood of S, denoted $\Gamma(S)$, is the union of the neighborhoods of the constituent vertices.

Note that $d(v)$ is the same as the cardinality of $\Gamma(v)$ when there are no self-loops or multiple edges between v and any of its neighbors.

Let k be the min-cut size. We fix our attention on a particular min-cut C with k edges. Clearly G has at least $kn/2$ edges; otherwise there would be a vertex of degree less than k, and its incident edges would be a min-cut of size less than k. We will bound from below the probability that no edge of C is ever contracted during an execution of the algorithm, so that the edges surviving till the end are exactly the edges in C.

Let $\mathcal{E}_i$ denote the event of *not* picking an edge of C at the ith step, for $1 \leq i \leq n-2$. The probability that the edge randomly chosen in the first step is in C is at most $k/(nk/2) = 2/n$, so that $\mathbf{Pr}[\mathcal{E}_1] \geq 1-2/n$. Assuming that $\mathcal{E}_1$ occurs, during the second step there are at least $k(n-1)/2$ edges, so the probability of picking an edge in C is at most $2/(n-1)$, so that $\mathbf{Pr}[\mathcal{E}_2 \mid \mathcal{E}_1] \geq 1-2/(n-1)$. At the ith step, the number of remaining vertices is $n-i+1$. The size of the min-cut is still at least k, so the graph has at least $k(n-i+1)/2$ edges remaining at this step. Thus, $\mathbf{Pr}[\mathcal{E}_i \mid \cap_{j=1}^{i-1}\mathcal{E}_j] \geq 1-2/(n-i+1)$. What is the probability that no edge of C is ever picked in the process? We invoke (1.6) to obtain

$$\mathbf{Pr}[\cap_{i=1}^{n-2}\mathcal{E}_i] \geq \prod_{i=1}^{n-2}\left(1-\frac{2}{n-i+1}\right) = \frac{2}{n(n-1)}.$$

The probability of discovering a particular min-cut (which may in fact be the unique min-cut in G) is larger than $2/n^2$. Thus our algorithm may err in declaring the cut it outputs to be a min-cut. Suppose we were to repeat the above algorithm $n^2/2$ times, making independent random choices each time. By (1.4), the probability that a min-cut is not found in any of the $n^2/2$

attempts is at most

$$\left(1-\frac{2}{n^2}\right)^{n^2/2} < 1/e.$$

By this process of repetition, we have managed to reduce the probability of failure from $1-2/n^2$ to a more respectable $1/e$. Further executions of the algorithm will make the failure probability arbitrarily small – the only consideration being that repetitions increase the running time.

Note the extreme simplicity of the randomized algorithm we have just studied. In contrast, most deterministic algorithms for this problem are based on network flows and are considerably more complicated. In Section 10.2 we will return to the min-cut problem and fill in some implementation details that have been glossed over in the above presentation; in fact, it will be shown that a variant of this algorithm has an expected running time that is significantly smaller than that of the best known algorithms based on network flow.

Exercise 1.2: Suppose that at each step of our min-cut algorithm, instead of choosing a random edge for contraction we choose two vertices at random and coalesce them into a single vertex. Show that there are inputs on which the probability that this modified algorithm finds a min-cut is exponentially small.

1.2. Las Vegas and Monte Carlo

The randomized sorting algorithm and the min-cut algorithm exemplify two different types of randomized algorithms. The sorting algorithm *always* gives the correct solution. The only variation from one run to another is its running time, whose distribution we study. We call such an algorithm a *Las Vegas algorithm.*

In contrast, the min-cut algorithm may sometimes produce a solution that is incorrect. However, we are able to bound the probability of such an incorrect solution. We call such an algorithm a *Monte Carlo algorithm.* In Section 1.1 we observed a useful property of a Monte Carlo algorithm: if the algorithm is run repeatedly with independent random choices each time, the failure probability can be made arbitrarily small, at the expense of running time. Later, we will see examples of algorithms in which both the running time and the quality of the solution are random variables; sometimes these are also referred to as Monte Carlo algorithms. For decision problems (problems for which the answer to an instance is YES or NO), there are two kinds of Monte Carlo algorithms: those with *one-sided error*, and those with *two-sided error.* A Monte Carlo algorithm is said to have two-sided error if there is a non-zero probability that it errs when it outputs either YES or NO. It is said to have one-sided error if the probability that it errs is zero for at least one of the possible outputs (YES/NO) that it produces.

CHAPTER 7

Algebraic Techniques

SOME of the most notable results in theoretical computer science, particularly in complexity theory, have involved a non-trivial use of algebraic techniques combined with randomization. In this chapter we describe some basic randomization techniques with an underlying algebraic flavor. We begin by describing Freivalds' technique for the verification of identities involving matrices, polynomials, and integers. We describe how this generalizes to the Schwartz-Zippel technique for identities involving multivariate polynomials, and we illustrate this technique by applying it to the problem of detecting the existence of perfect matchings in graphs. Then we present a related technique that leads to an efficient randomized algorithm for pattern matching in strings. We conclude with some complexity-theoretic applications of the techniques introduced here. In particular, we define interactive proof systems and demonstrate such systems for the graph non-isomorphism problem and the problem of counting the number of satisfying truth assignments for a Boolean formula. We then refine this concept into that of an efficiently verifiable proof and demonstrate such proofs for the satisfiability problem. We indicate how these concepts have led to a completely different view of classical complexity classes, as well as the new results obtained via the resulting insight into the structure of these classes.

Most of these techniques and their applications involve (sometimes indirectly) a *fingerprinting* mechanism, which can be described as follows. Consider the problem of deciding the equality of two elements x and y drawn from a large universe U. Under any "reasonable" model of computation, testing the equality of x and y then has a deterministic complexity of at least $\log |U|$. An alternative approach is to pick a random mapping from U into a significantly smaller universe V in such a way that there is a good chance that x and y are identical if and only if their images are identical. The images of x and y are their *fingerprints*, and their equality can be verified in $\log |V|$ time by comparing the fingerprints.

Throughout this chapter we will be working over some unspecified field $\mathbb{F}$. Part of the reason we do not explicitly specify the underlying field is that

typically the randomization will involve uniform sampling from a finite subset of the field; in such cases, we do not have to worry about whether the field is finite or not. The reader may find it helpful to think of $\mathbb{F}$ as the field $\mathbb{Q}$ of the rational numbers; when we restrict ourselves to finite fields, it may be useful to assume that $\mathbb{F}$ is $\mathbb{Z}_p$, the field of integers modulo some prime number p. We will use the unit-cost RAM model from Section 1.5.1 to measure the running time of an algorithm over the field $\mathbb{F}$. In this model each field operation (addition, subtraction, multiplication, division, comparison, or choosing a random element) takes unit time, provided the operand magnitude is polynomially related to the input size. For example, over the field of rationals we will assume that operations involving $O(\log n)$-bit numbers take unit time. This is not completely realistic as arithmetic operations are significantly more expensive in practice. However, in most applications described below this small additional factor in the running time is inconsequential, and we would get essentially the same result in the more expensive model.

7.1. Fingerprinting and Freivalds' Technique

We illustrate fingerprinting by describing a technique for verifying matrix multiplication. The fastest known algorithm for matrix multiplication runs in time $O(n^{2.376})$, which improves significantly on the obvious $O(n^3)$ time algorithm but has the disadvantage of being extremely complicated. Suppose we are given an implementation of this algorithm and would like to verify its correctness. Since program verification is a difficult task, a reasonable goal might be to verify the correctness of the output produced on specific executions of the algorithm. (Such verification on specific inputs has been studied in the theory of *program checking*.) In other words, given $n \times n$ matrices $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ over the field $\mathbb{F}$, we would like to verify that $\boldsymbol{AB} = \boldsymbol{C}$. We cannot afford to use a simpler but slower algorithm for matrix multiplication to verify the output $\boldsymbol{C}$, as this would defeat the purpose of using the fast matrix multiplication algorithm. Moreover, we would like to use the fact that we do not have to compute $\boldsymbol{C}$; rather, our task is to verify that this product is indeed $\boldsymbol{C}$. The following technique, known as *Freivalds' technique*, provides an elegant solution. It gives an $O(n^2)$ time randomized algorithm with a bounded error probability.

The randomized algorithm first chooses a random vector $\boldsymbol{r} \in \{0,1\}^n$; each component of $\boldsymbol{r}$ is chosen independently and uniformly at random from 0 and 1, the additive and multiplicative identities of the field $\mathbb{F}$. We can compute $\boldsymbol{x} = \boldsymbol{Br}$, $\boldsymbol{y} = \boldsymbol{Ax} = \boldsymbol{ABr}$, and $\boldsymbol{z} = \boldsymbol{Cr}$ in $O(n^2)$ time; clearly, if $\boldsymbol{AB} = \boldsymbol{C}$ then $\boldsymbol{y} = \boldsymbol{z}$. We now show that for $\boldsymbol{AB} \neq \boldsymbol{C}$, the probability that $\boldsymbol{y} \neq \boldsymbol{z}$ is at least $1/2$. The algorithm errs only if $\boldsymbol{AB} \neq \boldsymbol{C}$ but $\boldsymbol{y}$ and $\boldsymbol{z}$ turn out to be equal.

Theorem 7.1: *Let $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ be $n \times n$ matrices over $\mathbb{F}$ such that $\boldsymbol{AB} \neq \boldsymbol{C}$. Then for $\boldsymbol{r}$ chosen uniformly at random from $\{0,1\}^n$, $\mathbf{Pr}[\boldsymbol{ABr} = \boldsymbol{Cr}] \leq 1/2$.*

PROOF: Let $\boldsymbol{D} = \boldsymbol{AB} - \boldsymbol{C}$; we know that $\boldsymbol{D}$ is not the all-zeroes matrix. We wish to bound the probability that $\boldsymbol{y} = \boldsymbol{z}$, or, equivalently, the probability that $\boldsymbol{Dr} = 0$. Without loss of generality, we may assume that the first row in $\boldsymbol{D}$ has a non-zero entry, and that all the non-zero entries in that row precede the zero entries. Let $\boldsymbol{d}$ be the vector consisting of the entries from the first row in $\boldsymbol{D}$, and assume that the first $k > 0$ entries in $\boldsymbol{d}$ are non-zero. We concentrate on the probability that the inner product of $\boldsymbol{d}$ and $\boldsymbol{r}$ is non-zero; since the first entry in $\boldsymbol{Dr}$ is exactly $\boldsymbol{d}^T\boldsymbol{r}$, this yields a lower bound on the probability that $\boldsymbol{y} \neq \boldsymbol{z}$.

Now, the inner product $\boldsymbol{d}^T\boldsymbol{r} = 0$ if and only if

$$r_1 = \frac{-\sum_{i=2}^{k} d_i r_i}{d_1}. \tag{7.1}$$

We invoke the Principle of Deferred Decisions (Section 3.5) and assume that all the other random entries in $\boldsymbol{r}$ are chosen before r_1. Then the right-hand side of (7.1) is fixed at some value $v \in \mathbb{F}$. Since r_1 is uniformly distributed over a set of size 2, the probability that it equals v cannot exceed $1/2$. □

Exercise 7.1: Verify that there is nothing magical about choosing $\boldsymbol{r}$ to have only entries drawn from $\{0, 1\}$. In fact, any two elements of $\mathbb{F}$ may be used instead.

Thus, in $O(n^2)$ time we have reduced the matrix product verification problem to that of verifying the equality of two vectors, and the latter can be done in $O(n)$ time. This gives an overall running time of $O(n^2)$ for this Monte Carlo procedure. The probability of error can be reduced to $1/2^k$ by performing k independent iterations. The following exercise gives an alternative approach to reducing the probability of error.

Exercise 7.2: Suppose that each component of $\boldsymbol{r}$ is chosen uniformly and independently from some subset $\mathbb{S} \subseteq \mathbb{F}$. Show that the probability of error in the verification procedure is no more than $1/|\mathbb{S}|$. Compare the usefulness of the two different methods for reducing the error probability.

Freivalds' technique is applicable to verifying any matrix identity $\boldsymbol{X} = \boldsymbol{Y}$. Of course, if $\boldsymbol{X}$ and $\boldsymbol{Y}$ are explicitly provided, just comparing their entries takes only $O(n^2)$ time. But as in the case of matrix multiplication, there are situations where computing $\boldsymbol{X}$ explicitly is expensive (or even infeasible, as we will see in Section 7.8), whereas computing $\boldsymbol{Xr}$ is easy.

7.2. Verifying Polynomial Identities

Freivalds' technique is fairly general in that it can be applied to the verification of several different kinds of identities. In this section we show that it also applies

to the verification of identities involving polynomials. Two polynomials $P(x)$ and $Q(x)$ are said to be equal if they have the same coefficients for corresponding powers of x. Verifying identities of integers, or, in general, strings over any fixed alphabet, is a special case since we can represent any string of length n as a polynomial of degree n. This is achieved by treating the kth element in the string as the coefficient of the kth power of a symbolic variable.

We first consider the *polynomial product verification* problem: given polynomials $P_1(x)$, $P_2(x)$, $P_3(x) \in \mathbb{F}[x]$, verify that $P_1(x) \times P_2(x) = P_3(x)$. Assume that the polynomials $P_1(x)$ and $P_2(x)$ are of degree at most n; then $P_3(x)$ cannot have degree exceeding $2n$. Polynomials of degree n can be multiplied in $O(n \log n)$ time using Fast Fourier Transforms, whereas the evaluation of a polynomial at a fixed point requires $O(n)$ time.

The basic idea underlying the randomized algorithm for polynomial product verification is similar in spirit to the algorithm for matrices. Let $\mathbb{S} \subseteq \mathbb{F}$ be a set of size at least $2n+1$. Pick $r \in \mathbb{S}$ uniformly at random and evaluate $P_1(r)$, $P_2(r)$, and $P_3(r)$ in $O(n)$ time. The polynomial identity $P_1(x)P_2(x) = P_3(x)$ is declared correct unless $P_1(r)P_2(r) \neq P_3(r)$. This algorithm errs only when the polynomial identity is false but the evaluation of the polynomials at r fails to detect this.

Define the polynomial $Q(x) = P_1(x)P_2(x) - P_3(x)$ of degree $2n$. We say that a polynomial P is *identically zero*, or $P \equiv 0$, if all of its coefficients are zero. Clearly, $Q(x)$ is identically zero if and only if the polynomial product is correct. We complete the analysis of the randomized verification algorithm by showing that if $Q(x) \not\equiv 0$, then with high probability $Q(r) = P_1(r)P_2(r) - P_3(r) \neq 0$. Elementary algebra tells us that Q can have at most $2n$ distinct roots. Hence, unless $Q \equiv 0$, not more that $2n$ different choices of $r \in \mathbb{S}$ will have $Q(r) = 0$. Thus, the probability of error is at most $2n/|\mathbb{S}|$. This probability can be reduced by either using independent iterations of the entire algorithm or by choosing a sufficiently large set $\mathbb{S}$.

In the case where $\mathbb{F}$ is an infinite field (such as the reals), the error probability can be reduced to 0 by choosing r uniformly from the entire field $\mathbb{F}$. Unfortunately, this requires an infinite number of random bits! We could also use a deterministic version of this algorithm where each choice of $r \in \mathbb{S}$ is tried once. But this requires $2n+1$ different evaluations of each polynomial, and the best algorithm for this requires $\Theta(n \log^2 n)$ time, which is more than the time required to actually multiply $P_1(x)$ and $P_2(x)$.

This verification procedure is not restricted to polynomial product verification. It is a generic procedure for testing any polynomial identity of the form $P_1(x) = P_2(x)$, by transforming it into the identity $Q(x) = P_1(x) - P_2(x) \equiv 0$. Obviously, if the polynomials P_1 and P_2 are explicitly provided, we can perform this task deterministically in $O(n)$ time by comparing corresponding coefficients. The randomized algorithm will take as long to just evaluate the polynomials at a random point. However, the verification procedure pays off in situations where the polynomials are provided implicitly, such as when we have only a "black box" for computing the polynomial, with no means of accessing its coefficients. There are also situations where the polynomials are provided in

a form where computing the actual coefficients is exceedingly expensive. One example is provided by the following problem concerning the determinant of a symbolic matrix; in fact, this problem will turn out to be the same as that of verifying a polynomial identity involving *multivariate* polynomials, necessitating a generalization of the idea used for univariate polynomials.

Let M be an $n \times n$ matrix. The determinant of M is defined by

$$\det(M) = \sum_{\pi \in \mathbb{S}_n} \operatorname{sgn}(\pi) \prod_{i=1}^{n} M_{i,\pi(i)}, \tag{7.2}$$

where $\mathbb{S}_n$ is the symmetric group of permutations of size n, and $\operatorname{sgn}(\pi)$ is the sign of the permutation π. Recall that $\operatorname{sgn}(\pi) = (-1)^t$, where t is the number of pairwise element exchanges required to transform the identity permutation into π. Although the determinant has $n!$ terms, it can be evaluated in polynomial time given explicit values for the matrix entries M_{ij}.

► **Definition 7.1:** The *Vandermonde matrix* $M(x_1, \ldots, x_n)$ is defined in terms of the indeterminates $x_1, \ldots, x_n$ such that $M_{ij} = x_i^{j-1}$, that is

$$M = \begin{pmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{n-1} \\ & & & \cdot & \\ & & & \cdot & \\ & & & \cdot & \\ 1 & x_n & x_n^2 & \ldots & x_n^{n-1} \end{pmatrix}.$$

Vandermonde's identity states that for this matrix M, $\det(M) = \prod_{j<i}(x_i - x_j)$. Suppose that we did not have a proof of this identity and would like to verify it efficiently. Computing the determinant of this symbolic matrix is prohibitively expensive since it has $n!$ terms. Instead, we will formulate this as the problem of verifying that the polynomial $Q(x_1, \ldots, x_n) = \det(M) - \prod_{j<i}(x_i - x_j)$ is identically zero. Drawing upon our experience with Freivalds' technique, it seems natural to substitute random values for each x_i and check whether $Q \equiv 0$. The polynomial Q is easy to evaluate at a specific point since the determinant can be computed in polynomial time for specified values of the variables $x_1, \ldots, x_n$.

We formalize this intuition by extending the analysis of Freivalds' technique for univariate polynomial identity verification to the multivariate case. In a multivariate polynomial $Q(x_1, \ldots, x_n)$, the degree of any term is the sum of the exponents of the variables, and the *total degree* of Q is the maximum of the degrees of its terms.

Theorem 7.2 (Schwartz-Zippel Theorem): *Let $Q(x_1, \ldots, x_n) \in \mathbb{F}[x_1, \ldots, x_n]$ be a multivariate polynomial of total degree d. Fix any finite set $\mathbb{S} \subseteq \mathbb{F}$, and let $r_1, \ldots, r_n$ be chosen independently and uniformly at random from $\mathbb{S}$. Then*

$$\mathbf{Pr}[Q(r_1, \ldots, r_n) = 0 \mid Q(x_1, \ldots, x_n) \not\equiv 0] \leq \frac{d}{|\mathbb{S}|}.$$

PROOF: The proof is by induction on the number of variables n. The base case $n = 1$ involves a univariate polynomial $Q(x_1)$ of degree d, and by the preceding discussion we already know that for $Q(x_1) \not\equiv 0$, the probability that $Q(r_1) = 0$ is at most $d/|\mathbb{S}|$. Assume now that the induction hypothesis is true for a multivariate polynomial with up to $n-1$ variables, for $n > 1$.

Consider the polynomial $Q(x_1, \ldots, x_n)$, and factor out the variable x_1 to obtain

$$Q(x_1, \ldots, x_n) = \sum_{i=0}^{k} x_1^i Q_i(x_2, \ldots, x_n),$$

where $k \leq d$ is the largest exponent of x_1 in Q. (Assume that x_1 affects Q, so that $k > 0$). The coefficient of x_1^k, $Q_k(x_2, \ldots, x_n)$, is not identically zero by our choice of k. Since the total degree of Q_k is at most $d-k$, the induction hypothesis implies that the probability that $Q_k(r_2, \ldots, r_n) = 0$ is at most $(d-k)/|\mathbb{S}|$.

Suppose that $Q_k(r_2, \ldots, r_n) \neq 0$. Consider the following univariate polynomial:

$$q(x_1) = Q(x_1, r_2, r_3, \ldots, r_n) = \sum_{i=0}^{k} x_1^i Q_i(r_2, \ldots, r_n).$$

The polynomial $q(x_1)$ has degree k, and it is not identically zero since the coefficient of x_1^k is $Q_k(r_2, \ldots, r_n)$. The base case now implies that the probability that $q(r_1) = Q(r_1, r_2, \ldots, r_n)$ evaluates to 0 is at most $k/|\mathbb{S}|$.

Thus, we have shown the following two inequalities.

$$\begin{aligned} \mathbf{Pr}[Q_k(r_2, \ldots, r_n) = 0] &\leq \frac{d-k}{|\mathbb{S}|}; \\ \mathbf{Pr}[Q(r_1, r_2, \ldots, r_n) = 0 \mid Q_k(r_2, \ldots, r_n) \neq 0] &\leq \frac{k}{|\mathbb{S}|}. \end{aligned}$$

Invoking the result in Exercise 7.3, we find that the probability that $Q(r_1, r_2, \ldots, r_n) = 0$ is no more than the sum of these two probabilities, which is $d/|\mathbb{S}|$. This completes the induction. □

Exercise 7.3: Show that for any two events $\mathcal{E}_1$ and $\mathcal{E}_2$,

$$\mathbf{Pr}[\mathcal{E}_1] \leq \mathbf{Pr}[\mathcal{E}_1 \mid \overline{\mathcal{E}}_2] + \mathbf{Pr}[\mathcal{E}_2].$$

The randomized verification procedure for polynomials has one potential problem. In the case of infinite fields, the intermediate results in the evaluation of the polynomial could involve enormous values. This problem can be avoided in the case of integers by performing all the computations modulo a small random prime number, without adversely affecting the error probability. We will return to this issue in Example 7.1.

As suggested in Problem 7.1, Theorem 7.2 can be viewed as a generalization of Freivalds' technique from Section 7.1. A generalized version of this theorem is described in Problem 7.6.

7.3. Perfect Matchings in Graphs

We illustrate the power of the techniques of Section 7.2 by giving a fascinating application. Consider a bipartite graph $G(U, V, E)$ with the independent sets of vertices $U = \{u_1, \ldots, u_n\}$ and $V = \{v_1, \ldots, v_n\}$. A *matching* is a collection of edges $M \subseteq E$ such that each vertex occurs at most once in M. A *perfect matching* is a matching of size n. Each perfect matching M in G can be viewed as a permutation from U into V. More precisely, the perfect matchings in G can be put into a one-to-one correspondence with the permutations in $\mathbb{S}_n$, where the matching corresponding to a permutation $\pi \in \mathbb{S}_n$ is given by the pairs $(u_i, v_{\pi(i)})$, for $1 \leq i \leq n$. The following theorem draws a connection between determinants and matchings.

Theorem 7.3 (Edmonds' Theorem): *Let A be the $n \times n$ matrix obtained from $G(U, V, E)$ as follows:*

$$A_{ij} = \begin{cases} x_{ij} & (u_i, v_j) \in E \\ 0 & (u_i, v_j) \notin E \end{cases}.$$

Define the multivariate polynomial $Q(x_{11}, x_{12}, \ldots, x_{nn})$ as being equal to $\det(A)$. *Then, G has a perfect matching if and only if $Q \not\equiv 0$.*

Remark: The matrix of indeterminates is sometimes referred to as the *Edmonds matrix* of a bipartite graph. We do not explicitly specify the underlying field because any field will do for the purposes of this theorem.

PROOF: The determinant of A is given by

$$\det(A) = \sum_{\pi \in \mathbb{S}_n} \operatorname{sgn}(\pi) A_{1,\pi(1)} A_{2,\pi(2)} \ldots A_{n,\pi(n)}.$$

Since each indeterminate x_{ij} occurs at most once in A, there can be no cancellation of the terms in the summation. Therefore the determinant is not identically zero if and only if there is a permutation π for which the corresponding term in the summation is non-zero. The latter happens if and only if each of the entries $A_{i,\pi(i)}$, for $1 \leq i \leq n$, is non-zero. This is equivalent to having a perfect matching (the one corresponding to π) in G. □

We can now construct a simple randomized test for the existence of perfect matchings. Using the algorithm from Section 7.2, we can determine whether the determinant is identically zero or not. The time required is dominated by the cost of computing a determinant, which is essentially that of multiplying two matrices. As it turns out, there are algorithms for *constructing* a maximum matching in a graph in time $O(m\sqrt{n})$, where $m = |E|$. Since the time to compute the determinant exceeds $m\sqrt{n}$ for small m, the payoff in using this randomized decision procedure is marginal at best. However, we will see later (in Section 12.4) that this decision procedure is essential for devising a fast *parallel* algorithm for computing a maximum matching in a graph. In Problem 7.8 we will see that this technique also applies to the case of non-bipartite graphs.