Even more surprising, the lattice reduction algorithm can prove that no linear integer relation with integers smaller than a chosen parameter exists among the real or complex numbers. There is an efficient alternative to the lattice reduction algorithm, originally due to Ferguson and Forcade [1982] and recently improved by Ferguson and Bailey.

The complexity of factoring an integer polynomial of degree n with coefficients of no more than l bits is thus a polynomial in n and l. From a theoretical point of view, an algorithm with a low estimate is by Miller [1992] and has a running time of $O(n^{5+o(1)}l^{1+o(1)} + n^{4+o(1)}l^{2+o(1)})$ bit operations. It is expected that the relation-finding methods will become usable in practice on hard-to-factor polynomials in the near future. If the hard-to-factor input polynomial is irreducible, an alternate approach can be used to prove its irreducibility. One finds an integer evaluation point at which the integral value of the polynomial has a large prime factor, and the irreducibility follows by mathematical theorems. Monagan [1992] has proven large hard-to-factor polynomials irreducible in this way, which would be hopeless by the lifting algorithm.

Coefficient fields other than finite fields and the rational numbers are of interest. Computing the factorizations of univariate polynomials over the complex numbers is the root finding problem described in the earlier section Approximating Polynomial Zeros. When the coefficient field has an extra variable, such as the field of fractions of polynomials (rational functions) the problem reduces, by an old theorem of Gauss, to factoring multivariate polynomials, which we discuss subsequently. When the coefficient field is the field of Laurent series in t with a finite segment of negative powers,

$$\frac{c_{-k}}{t^k} + \frac{c_{-k+1}}{t^{k-1}} + \cdots + \frac{c_{-1}}{t} + c_0 + c_1 t + c_2 t^2 + \cdots, \quad \text{where } k \geq 0$$

fast methods appeal to the theory of Puiseux series, which constitute the domain of algebraic functions [Walsh 1993].

8.4.3 Polynomials in Two Variables

Factoring bivariate polynomials by reduction to univariate factorization via homomorphic projection and subsequent lifting can be done similarly to the univariate algorithm [Musser 1975]. The second variable y takes the role of the prime integer p and $f(x, y) \bmod y = f(x, 0)$. Lifting is possible only if $f(x, 0)$ had no multiple root. Provided that $f(x, y)$ has no multiple factor, which can be ensured by a simple GCD computation, the *squarefreeness* of $f(x, 0)$ can be obtained by variable translation $\hat{y} = y + a$, where a is an easy-to find constant in the coefficient field. For certain domains, such as the rational numbers, any irreducible multivariate polynomial $h(x, y)$ can be mapped to an irreducible univariate polynomial $h(x, b)$ for some constant b. This is the important *Hilbert irreducibility theorem*, whose consequence is that the combinatorial explosion observed in the univariate lifting algorithm is, in practice, unlikely. However, the magnitude and probabilistic distribution of good points b is not completely analyzed.

For so-called non-Hilbertian coefficient fields good reduction is not possible. An important such field is the complex number. Clearly, all $f(x, b)$ completely split into linear factors, while $f(x, y)$ may be irreducible over the complex numbers. An example of an irreducible polynomial is $f(x, y) = x^2 - y^3$. Polynomials that remain irreducible over the complex numbers are called absolutely irreducible. An additional problem is the determination of the algebraic extension of the ground field in which the absolutely irreducible factors can be expressed. In the example

$$x^6 - 2x^3y^2 + y^4 - 2x^3 = (x^3 - \sqrt{2}x - y^2) \cdot (x^3 + \sqrt{2}x - y^2)$$

the needed extension field is $\mathbb{Q}(\sqrt{2})$. The relation-finding approach proves successful for this problem. The root is computed as a Taylor series in y, and the integrality of the linear relation for the powers of the series means that the multipliers are polynomials in y of bounded degree. Several algorithms of polynomial-time complexity and pointers to the literature are found in Kaltofen [1995].

Bivariate polynomials constitute implicit representations of algebraic curves. It is an important operation in geometric modeling to convert from implicit to parametric representation. For example, the circle

$$x^2 + y^2 - 1 = 0$$

has the rational parameterization

$$x = \frac{2t}{1+t^2}, \qquad y = \frac{1-t^2}{1+t^2}, \quad \text{where } -\infty \le t \le \infty$$

Algorithms are known that can find such rational parameterizations provided that they exist [Sendra and Winkler 1991]. It is crucial that the inputs to these algorithms are absolutely irreducible polynomials.

8.4.4 Polynomials in Many Variables

Polynomials in many variables, such as the symmetric Toeplitz determinant previously exhibited, are rarely given explicitly, due to the fact that the number of possible terms grows exponentially in the number of variables: there can be as many as $\binom{n+v}{n} \ge 2^{\min\{n,v\}}$ terms in a polynomial of degree n with v variables. Even the factors may be dense in canonical representation, but could be sparse in another basis: for instance, the polynomial

$$(x_1 - 1)(x_2 - 2) \cdots (x_v - v) + 1$$

has only two terms in the shifted basis, whereas it has 2^v terms in the power basis, i.e., in expanded format.

Randomized algorithms are available that can efficiently compute a factor of an implicitly given polynomial, say, a matrix determinant, and even can find a shifted basis with respect to which a factor would be sparse, provided, of course, that such a shift exists. The approach is by manipulating polynomials in so-called black box representations [Kaltofen and Trager 1990]: a black box is an object that takes as input a value for each variable, and then produces the value of the polynomial it represents at the specified point. In the Toeplitz example the representation of the determinant could be the Gaussian elimination program which computes it. We note that the size of the polynomial in this case would be nearly constant, only the variable names and the dimension need to be stored. The factorization algorithm then outputs procedures which will evaluate all irreducible factors at an arbitrary point (supplied as the input). These procedures make calls to the black box given as input to the factorization algorithm in order to evaluate them at certain points, which are derived from the point at which the procedures computing the values of the factors are probed. It is, of course, assumed that subsequent calls evaluate one and the same factor and not associates that are scalar multiples of one another. The algorithm by Kaltofen and Trager [1990] finds procedures that with a controllably high probability evaluate the factors correctly. Randomization is needed to avoid parasitic factorizations of homomorphic images which provide some static data for the factor boxes and cannot be avoided without mathematical conjecture. The procedures that evaluate the individual factors are deterministic.

Factors constructed as black box programs are much more space efficient than those represented in other formats, for example, the straight-line program format [Kaltofen 1989]. More importantly, once the black box representation for the factors is found, sparse representations can be rapidly computed by any of the new sparse interpolation algorithms. See Grigoriev and Lakshman [1995] for the latest method allowing shifted bases and pointers to the literature of other methods, including those for the standard power bases.

The black box representation of polynomials is normally not supported by commercial computer algebra systems such as Axiom, Maple, or Mathematica. Díaz is currently developing the FoxBox system in C++ that makes black box methodology available to users of such systems. It is anticipated that factorizations as those of large symmetric Toeplitz determinants will be possible on computers. Earlier implementations based on the straight-line program model [Freeman et al. 1988] could factor 16 × 16 group determinants, which represent polynomials of over 300 million terms.

Acknowledgment

This material is based on work supported in part by the National Science Foundation under Grants CCR-9319776 (first and second author) and CCR-9020690 (third author), by GTE under a Graduate Computer Science Fellowship (first author), and by PSC CUNY Awards 665301 and 666327 (third author). Part of this work was done while the second author was at the Department of Computer Science at Rensselaer Polytechnic Insititute in Troy, New York.

Defining Terms

Characteristic polynomial: A polynomial associated with a square matrix, the determinant of the matrix when a single variable is subtracted to its diagonal entries. The roots of the characteristic polynomial are the eigenvalues of the matrix.

Condition number: A scalar derived from a matrix that measures its relative nearness to a singular matrix. Very close to singular means a large condition number, in which case numeric inversion becomes an unstable process.

Degree order: An order of the terms in a multivariate polynomial; for two variables x and y with $x \prec y$ the ascending chain of terms is $1 \prec x \prec y \prec x^2 \prec xy \prec y^2 \cdots$.

Determinant: A polynomial in the entries of a square matrix with the property that its value is nonzero if and only if the matrix is invertible.

Lexicographic order: An order of the terms in a multivariate polynomial; for two variables x and y with $x \prec y$ the ascending chain of terms is $1 \prec x \prec x^2 \prec \cdots \prec y \prec xy \prec x^2y \cdots \prec y^2 \prec xy^2 \cdots$.

Ops: Arithmetic operations, i.e., additions, subtractions, multiplications, or divisions; as in floating point operations (*flops*).

Singularity: A square matrix is singular if there is a nonzero second matrix such that the product of the two is the zero matrix. Singular matrices do not have inverses.

Sparse matrix: A matrix where many of the entries are zero.

Structured matrix: A matrix where each entry can be derived by a formula depending on few parameters. For instance, the Hilbert matrix has $1/(i + j - 1)$ as the entry in row i and column j.

References

Anderson, E. et al. 1992. *LAPACK Users' Guide*. SIAM Pub., Philadelphia, PA.

Aho, A., Hopcroft, J., and Ullman, J. 1974. *The Design and Analysis of Algorithms*. Addison–Wesley, Reading, MA.

Bareiss, E. H. 1968. Sylvester's identity and multistep integers preserving Gaussian elimination. *Math. Comp.* 22:565–578.

Becker, T. and Weispfenning, V. 1993. *Gröbner Bases: A Computational Approach to Commutative Algebra*. Springer–Verlag, New York.

Berlekamp, E. R. 1967. Factoring polynomials over finite fields. *Bell Systems Tech. J.* 46:1853–1859; rev. 1968. *Algebraic Coding Theory*. Chap. 6, McGraw–Hill, New York.

Berlekamp, E. R. 1970. Factoring polynomials over large finite fields. *Math. Comp.* 24:713–735.

Bini, D. and Pan, V. Y. 1991. Parallel complexity of tridiagonal symmetric eigenvalue problem. In *Proc. 2nd Annu. ACM-SIAM Symp. on Discrete Algorithms*, pp. 384–393. ACM Press, New York, SIAM Pub., 1994. Philadelphia, PA.

Bini, D. and Pan, V. Y. 1994. *Polynomial and Matrix Computations Vol. 1, Fundamental Algorithms*. Birkhäuser, Boston, MA.

Bini, D. and Pan, V. Y. 1996. *Polynomial and Matrix Computations, Vol. 2*. Birkhäuser, Boston, MA.

Borodin, A. and Munro, I. 1975. *Computational Complexity of Algebraic and Numeric Problems*. American Elsevier, New York.

Brown, W. S. 1971. On Euclid's algorithm and the computation of polynomial greatest common divisors. *J. ACM* 18:478–504.

Brown, W. S. and Traub, J. F. 1971. On Euclid's algorithm and the theory of subresultants. *J. ACM* 18:505–514.

Buchberger, B. 1965. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal.* Ph.D. dissertation. University of Innsbruck, Austria.

Buchberger, B. 1976. A theoretical basis for the reduction of polynomials to canonical form. *ACM SIGSAM Bull.* 10(3):19–29.

Buchberger, B. 1983. A note on the complexity of constructing Gröbner-bases. In *Proc. EUROCAL '83*, J. A. van Hulzen, ed. *Lecture Notes in Computer Science*, pp. 137–145. Springer.

Buchberger, B. 1985. Gröbner bases: an algorithmic method in polynomial ideal theory. In *Recent Trends in Multidimensional Systems Theory*, N. K. Bose, ed., pp. 184–232. D. Reidel, Dordrecht, Holland.

Cantor, D. G. 1989. On arithmetical algorithms over finite fields. *J. Combinatorial Theory, Serol. A* 50:285–300.

Canny, J. 1990. Generalized characteristic polynomials. *J. Symbolic Comput.* 9(3):241–250.

Canny, J. and Emiris, I. 1993a. An efficient algorithm for the sparse mixed resultant. In *Proc. AAECC-10*, G. Cohen, T. Mora, and O. Moreno, ed. Vol. 673, *Lecture Notes in Computer Science*, pp. 89–104. Springer.

Canny, J. and Emiris, I. 1993b. A practical method for the sparse resultant. In *ISSAC '93*, *Proc. Internat. Symp. Symbolic Algebraic Comput.*, M. Bronstein, ed., pp. 183–192. ACM Press, New York.

Canny, J. and Emiris, I. 1994. Efficient incremental algorithms for the sparse resultant and the mixed volume. Tech. Rep., Univ. California-Berkeley, CA.

Canny, J., Kaltofen, E., and Lakshman, Y. 1989. Solving systems of non-linear polynomial equations faster. In *Proc. ACM-SIGSAM Internat. Symp. Symbolic Algebraic Comput.*, pp. 121–128.

Canny, J. and Manocha, D. 1991. Efficient techniques for multipolynomial resultant algorithms. In *ISSAC '91, Proc. Internat. Symp. Symbolic Algebraic Comput.*, S. M. Watt, ed., pp. 85–95, ACM Press, New York.

Cantor, D. G. and Zassenhaus, H. 1981. A new algorithm for factoring polynomials over finite fields. *Math. Comp.* 36:587–592.

Cayley, A. 1865. On the theory of eliminaton. *Cambridge and Dublin Math. J.* 3:210–270.

Chionh, E. 1990. *Base Points, Resultants and Implicit Representation of Rational Surfaces.* Ph.D. dissertation. Department of Computer Science, University of Waterloo, Waterloo, Canada.

Collins, G. E. 1967. Subresultants and reduced polynomial remainder sequences. *J. ACM* 14:128–142.

Collins, G. E. 1971. The calculation of multivariate polynomial resultants. *J. ACM* 18:515–532.

Cuppen, J. J. M. 1981. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numer. Math.* 36:177–195.

Davenport, J. H., Tournier, E., and Siret, Y. 1988. *Computer Algebra Systems and Algorithms for Algebraic Computation*. Academic Press, London.

Díaz, A. and Kaltofen, E. 1995. On computing greatest common divisors with polynomials given by black boxes for their evaluation. In *ISSAC '95 Proc. 1995 Internat. Symp. Symbolic Algebraic Comput.*, A. H. M. Levelt, ed., pp. 232–239, ACM Press, New York.

Dixon, A. L. 1908. The elimination of three quantics in two independent variables. In *Proc. London Math. Soc.* Vol. 6, pp. 468–478.

Dongarra, J. et al. 1978. *LAPACK Users' Guide*. SIAM Pub., Philadelphia, PA.

Faugère, J. C., Gianni, P., Lazard, D., and Mora, T. 1993. Efficient computation of zero-dimensional Gröbner bases by change of ordering. *J. Symbolic Comput.* 16(4):329–344.

Ferguson, H. R. P. and Forcade, R. W. 1982. Multidimensional Euclidean algorithms. *J. Reine Angew. Math.* 334:171–181.

Finch, S. 1995. The miraculous Bailey–Borwein–Plouffe pi algorithm. Internet document, Mathsoft Inc., http://www.mathsoft.com/asolve/plouffe/plouffe.html, Oct.

Foster, L. V. 1981. Generalizations of Laguerre's method: higher order methods. *SIAM J. Numer. Anal.* 18:1004–1018.

Freeman, T. S., Imirzian, G., Kaltofen, E., and Lakshman, Y. 1988. Dagwood: a system for manipulating polynomials given by straight-line programs. *ACM Trans. Math. Software* 14(3):218–240.

Garbow, B. S. et al. 1972. *Matrix Eigensystem Routines: EISPACK Guide Extension*. Springer, New York.

Geddes, K. O., Czapor, S. R., and Labahn, G. 1992. *Algorithms for Computer Algebra*. Kluwer Academic.

Gelfand, I. M., Kapranov, M. M., and Zelevinsky, A. V. 1994. *Discriminants, Resultants and Multidimensional Determinants*. Birkhäuser Verlag, Boston, MA.

George, A. and Liu, J. W.-H. 1981. *Computer Solution of Large Sparse Positive Definite Linear Systems.* Prentice–Hall, Englewood Cliffs, NJ.

Gianni, P. and Trager, B. 1985. GCD's and factoring polynomials using Gröbner bases. *Proc. EUROCAL '85*, Vol. 2, *Lecture Notes in Computer Science*, 204, pp. 409–410.

Giesbrecht, M. 1995. Nearly optimal algorithms for canonical matrix forms. *SIAM J. Comput.* 24(5):948–969.

Gilbert, J. R. and Tarjan, R. E. 1987. The analysis of a nested dissection algorithm. *Numer. Math.* 50:377–404.

Golub, G. H. and Van Loan, C. F. 1989. *Matrix Computations.* Johns Hopkins Univ. Press, Baltimore, MD.

Gondran, M. and Minoux, M. 1984. *Graphs and Algorithms.* Wiley–Interscience, New York.

Grigoriev, D. Y. and Lakshman, Y. N. 1995. Algorithms for computing sparse shifts for multivariate polynomials. In *ISSAC '95 Proc. 1995 Internat. Symp. Symbolic Algebraic Comput.*, A. H. M. Levelt, ed., pp. 96–103, ACM Press, New York.

Hansen, E., Patrick, M., and Rusnack, J. 1977. Some modifications of Laguerre's method. *BIT* 17:409–417.

Heath, M. T., Ng, E., and Peyton, B. W. 1991. Parallel algorithms for sparse linear systems. *SIAM Rev.* 33:420–460.

Jenkins, M. A., and Traub, J. F. 1970. A three-stage variable-shift iteration for polynomial zeros and its relation to generalized Rayleigh iteration. *Numer. Math.* 14:252–263.

Kaltofen, E. 1982. Polynomial factorization. In 2nd ed. *Computer Algebra*, B. Buchberger, G. Collins, and R. Loos, eds., pp. 95–113. Springer–Verlag, Vienna.

Kaltofen, E. 1988. Greatest common divisors of polynomials given by straight-line programs. *J. ACM* 35(1):231–264.

Kaltofen, E. 1989. Factorization of polynomials given by straight-line programs. In *Randomness and Computation*, S. Micali, ed. Vol. 5 of Advances in computing research, pp. 375–412. JAI Press, Greenwhich, CT.

Kaltofen, E. 1990. Polynomial factorization 1982–1986. 1990. In *Computers in Mathematics*, D. V. Chudnovsky and R. D. Jenks, eds. Vol. 125, *Lecture Notes in Pure and Applied Mathematics,* pp. 285–309. Marcel Dekker, New York.

Kaltofen, E. 1992. Polynomial factorization 1987–1991. In *Proc. LATIN '92*, I. Simon, ed. Vol. 583, *Lecture Notes in Computer Science*, pp. 294–313.

Kaltofen, E. 1995. Effective Noether irreducibility forms and applications. *J. Comput. Syst. Sci.* 50(2):274–295.

Kaltofen, E., Krishnamoorthy, M. S., and Saunders, B. D. 1990. Parallel algorithms for matrix normal forms. *Linear Algebra Appl.* 136:189–208.

Kaltofen, E. and Lakshman, Y. 1988. Improved sparse multivariate polynomial interpolation algorithms. *Proc. ISSAC '88,* Vol. 358, *Lecture Notes in Computer Science,* pp. 467–474.

Kaltofen, E. and Lobo, A. 1994. Factoring high-degree polynomials by the black box Berlekamp algorithm. In *ISSAC '94, Proc. Internat. Symp. Symbolic Algebraic Comput.*, J. von zur Gathen and M. Giesbrecht, eds., pp. 90–98, ACM Press, New York.

Kaltofen, E., Musser, D. R., and Saunders, B. D. 1983. A generalized class of polynomials that are hard to factor. *SIAM J. Comp.* 12(3):473–485.

Kaltofen, E. and Pan, V. 1991. Processor efficient parallel solution of linear systems over an abstract field. In *Proc. 3rd Ann. ACM Symp. Parallel Algor. Architecture*, pp. 180–191, ACM Press, New York.

Kaltofen, E. and Pan, V. 1992. Processor-efficient parallel solution of linear systems II: the positive characteristic and singular cases. In *Proc. 33rd Annual Symp. Foundations of Comp. Sci.*, pp. 714–723, Los Alamitos, CA. IEEE Computer Society Press.

Kaltofen, E. and Shoup, V. 1995. Subquadratic-time factoring of polynomials over finite fields. In *Proc. 27th Annual ACM Symp. Theory Comp.*, pp. 398–406, ACM Press, New York.

Kaltofen, E. and Trager, B. 1990. Computing with polynomials given by black boxes for their evaluations: greatest common divisors, factorization, separation of numerators and denominators. *J. Symbolic Comput.* 9(3):301–320.

Kapur, D. 1986. Geometry theorem proving using Hilbert's nullstellensatz. *J. Symbolic Comp.* 2:399–408.

Kapur, D. and Lakshman, Y. N. 1992. Elimination methods: an introduction. In *Symbolic and Numerical Computation for Artificial Intelligence*. B. Donald, D. Kapur, and J. Mundy, eds. Academic Press.

Kapur, D. and Saxena, T. 1995. Comparison of various multivariate resultant formulations. In *Proc. Internat. Symp. Symbolic Algebraic Comput. ISSAC '95*, A. H. M. Levelt, ed., pp. 187–195, ACM Press, New York.

Kapur, D., Saxena, T., and Yang, L. 1994. Algebraic and geometric reasoning using Dixon resultants. In *ISSAC '94, Proc. Internat. Symp. Symbolic Algebraic Comput.* J. von zur Gathen and M. Giesbrecht, ed., pp. 99–107, ACM Press, New York.

Knuth, D. E. 1981. *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms,* 2nd ed. Addison–Wesley, Reading, MA.

Lakshman, Y. N. 1990. *On the complexity of computing Gröbner bases for zero dimensional polynomia.* Ph.D. thesis, Dept. Comput. Sci., Rensselaer Polytechnic Inst. Troy, NY, Dec.

Lakshman, Y. N. and Saunders, B. D. 1994. On computing sparse shifts for univariate polynomials. In *ISSAC '94, Proc. Internat. Symp. Symbolic Algebraic Comput.*, J. von zur Gathen and M. Giesbrecht, eds., pp. 108–113, ACM Press, New York.

Lakshman, Y. N. and Saunders, B. D. 1995. Sparse polynomial interpolation in non-standard bases. *SIAM J. Comput.* 24(2):387–397.

Lazard, D. 1981. Résolution des systèmes d'équation algébriques. *Theoretical Comput. Sci.* 15:77–110. (In French).

Lenstra, A. K., Lenstra, H. W., and Lovász, L. 1982. Factoring polynomials with rational coefficients. *Math. Ann.* 261:515–534.

Leyland, P. 1995. Cunningham project data. Internet document, Oxford Univ., ftp://sable.ox.ac.uk/pub/math/cunningham/, November.

Lipton, R. J., Rose, D., and Tarjan, R. E. 1979. Generalized nested dissection. *SIAM J. on Numer. Analysis* 16(2):346–358.

Macaulay, F. S. 1916. Algebraic theory of modular systems. *Cambridge Tracts* 19, Cambridge.

MacWilliams, F. J. and Sloan, N. J. A. 1977. *The Theory of Error-Correcting Codes*. North–Holland, New York.

Madsen, K. 1973. A root-finding algorithm based on Newton's method. *BIT* 13:71–75.

McCormick, S., ed. 1987. *Multigrid Methods*. SIAM Pub., Philadelphia, PA.

McNamee, J. M. 1993. A bibliography on roots of polynomials. *J. Comput. Appl. Math.* 47(3):391–394.

Miller, V. 1992. Factoring polynomials via relation-finding. In *Proc. ISTCS '92*, D. Dolev, Z. Galil, and M. Rodeh, eds. Vol. 601, *Lecture Notes in Computer Science*, pp. 115–121.

Monagan, M. B. 1992. A heuristic irreducibility test for univariate polynomials. *J. Symbolic Comput.* 13(1):47–57.

Musser, D. R. 1975. Multivariate polynomial factorization. *J. ACM* 22:291–308.

Niederreiter, H. 1994. New deterministic factorization algorithms for polynomials over finite fields. In *Finite Fields: Theory, Applications and Algorithms*, L. Mullen and P. J.-S. Shiue, eds. Vol. 168, Contemporary mathematics, pp. 251–268, Amer. Math. Soc., Providence, RI.

Ortega, J. M., and Voight, R. G. 1985. Solution of partial differential equations on vector and parallel computers. *SIAM Rev.* 27(2):149–240.

Pan, V. Y. 1984a. How can we speed up matrix multiplication? *SIAM Rev.* 26(3):393–415.

Pan, V. Y. 1984b. How to multiply matrices faster. *Lecture Notes in Computer Science,* 179.

Pan, V. Y. 1987. Sequential and parallel complexity of approximate evaluation of polynomial zeros. *Comput. Math. (with Appls.),* 14(8):591–622.

Pan, V. Y. 1991. Complexity of algorithms for linear systems of equations. In *Computer Algorithms for Solving Linear Algebraic Equations (State of the Art)*, E. Spedicato, ed. Vol. 77 of *NATO ASI Series*, Series F: computer and systems sciences, pp. 27–56, Springer–Verlag, Berlin.

Pan, V. Y. 1992a. Complexity of computations with matrices and polynomials. *SIAM Rev.* 34(2):225–262.

Pan, V. Y. 1992b. Linear systems of algebraic equations. In *Encyclopedia of Physical Sciences and Technology*, 2nd ed. Marvin Yelles, ed. Vol. 8, pp. 779–804, 1987. 1st ed. Vol. 7, pp. 304–329.

Pan, V. Y. 1993. Parallel solution of sparse linear and path systems. In *Synthesis of Parallel Algorithms*, J. H. Reif, ed. Ch. 14, pp. 621–678. Morgan Kaufmann, San Mateo, CA.

Pan, V. Y. 1994a. Improved parallel solution of a triangular linear system. *Comput. Math. (with Appl.)*, 27(11):41–43.

Pan, V. Y. 1994b. *On approximating polynomial zeros: modified quadtree construction and improved Newton's iteration.* Manuscript, Lehman College, CUNY, Bronx, New York.

Pan, V. Y. 1995a. Parallel computation of a Krylov matrix for a sparse and structured input. *Math. Comput. Modelling* 21(11):97–99.

Pan, V. Y. 1995b. *Solving a polynomial equation: some history and recent progress.* Manuscript, Lehman College, CUNY, Bronx, New York.

Pan, V. Y. 1996. Optimal and nearly optimal algorithms for approximating polynomial zeros. *Comput. Math. (with Appl.).*

Pan, V. Y. and Preparata, F. P. 1995. Work-preserving speed-up of parallel matrix computations. *SIAM J. Comput.* 24(4):811–821.

Pan, V. Y. and Reif, J. H. 1992. Compact multigrid. *SIAM J. Sci. Stat. Comput.* 13(1):119–127.

Pan, V. Y. and Reif, J. H. 1993. Fast and efficient parallel solution of sparse linear systems. *SIAM J. Comp.*, 22(6):1227–1250.

Pan, V. Y., Sobze, I. and Atinkpahoun, A. 1995. On parallel computations with band matrices. *Inf. and Comput.* 120(2):227–250.

Parlett, B. 1980. *Symmetric Eigenvalue Problem*. Prentice–Hall, Englewood Cliffs, NJ.

Quinn, M. J. 1994. *Parallel Computing: Theory and Practice*. McGraw–Hill, New York.

Rabin, M. O. 1980. Probabilistic algorithms in finite fields. *SIAM J. Comp.* 9:273–280.

Renegar, J. 1989. On the worst case arithmetic complexity of approximating zeros of systems of polynomials. *SIAM J. Comput.* 18(2):350–370.

Ritt, J. F. 1950. *Differential Algebra*. AMS, New York.

Saad, Y. 1992. *Numerical Methods for Large Eigenvalue Problems: Theory and Algorithms*. Manchester Univ. Press, U.K., Wiley, New York. 1992.

Saad, Y. 1995. *Iterative Methods for Sparse Linear Systems*. PWS Kent, Boston, MA.

Sendra, J. R. and Winkler, F. 1991. Symbolic parameterization of curves. *J. Symbolic Comput.* 12(6): 607–631.

Shoup, V. 1996. A new polynomial factorization algorithm and its implementation. *J. Symbolic Comput.*

Smith, B. T. et al. 1970. *Matrix Eigensystem Routines: EISPACK Guide,* 2nd ed. Springer, New York.

St. Schwarz, 1956. On the reducibility of polynomials over a finite field. *Quart. J. Math. Oxford Ser. (2)*, 7:110–124.

Stederberg, T. and Goldman, R. 1986. Algebraic geometry for computer-aided design. *IEEE Comput. Graphics Appl.* 6(6):52–59.

Sturmfels, B. 1991. Sparse elimination theory. In *Proc. Computat. Algebraic Geom. and Commut. Algebra*, D. Eisenbud and L. Robbiano, eds. Cortona, Italy, June.

Sturmfels, B. and Zelevinsky, A. 1992. Multigraded resultants of the Sylvester type. *J. Algebra*.

von zur Gathen, J. and Shoup, V. 1992. Computing Frobenius maps and factoring polynomials. *Comput. Complexity* 2:187–224.

Walsh, P. G. 1993. *The computation of Puiseux expansions and a quantitative version of Runge's theorem on diophantine equations.* Ph.D. dissertation. University of Waterloo, Waterloo, Canada.

Watkins, D. S. 1982. Understanding the QR algorithm. *SIAM Rev.* 24:427–440.

Watkins, D. S. 1991. Some perspectives on the eigenvalue problem. *SIAM Rev.* 35(3):430–471.

Wilkinson, J. H. 1965. *The Algebraic Eigenvalue Problem*. Clarendon Press, Oxford, England.

Winkler, F. 1996. *Introduction to Computer Algebra*. Springer–Verlag, Heidelberg, Germany.

Wu, W. 1984. Basis principles of mechanical theorem proving in elementary geometries. *J. Syst. Sci. Math Sci.* 4(3):207–235.

Wu, W. 1986. Basis principles of mechanical theorem proving in elementary geometries. *J. Automated Reasoning* 2:219–252.

Zassenhaus, H. 1969. On Hensel factorization I. *J. Number Theory* 1:291–311.

Zippel, R. 1993. *Effective Polynomial Computations*, p. 384. Kluwer Academic, Boston, MA.

Further Information

The books by Knuth [1981], Davenport et al. [1988], Geddes et al. [1992], and Zippel [1993] provide a much broader introduction to the general subject. There are well-known libraries and packages of subroutines for the most popular numerical matrix computations, in particular, Dongarra et al. [1978] for solving linear systems of equations, Smith et al. [1970] and Garbow et al. [1972] approximating matrix eigenvalues, and Anderson et al. [1992] for both of the two latter computational problems. There is a comprehensive treatment of numerical matrix computations [Golub and Van Loan 1989], with extensive bibliography, and there are several more specialized books on them [George and Liu 1981, Wilkinson 1965, Parlett 1980, Saad 1992, 1995], as well as many survey articles [Heath et al. 1991, Watkins 1991, Ortega and Voight 1985, Pan 1992b] and thousands of research articles.

Special (more efficient) parallel algorithms have been devised for special classes of matrices, such as sparse [Pan and Reif 1993, Pan 1993], banded [Pan et al. 1995], and dense structured [Bini and Pan (cf. [1994])]. We also refer to Pan and Preparata [1995] on a simple but surprisingly effective extension of Brent's principle for improving the processor and work efficiency of parallel matrix algorithms and to Golub and Van Loan [1989], Ortega and Voight [1985], and Heath et al. [1991] on practical parallel algorithms for matrix computations.

9 Cryptography

Jonathan Katz
University of Maryland

9.1 Introduction

Cryptography is a vast subject, and we cannot hope to give a comprehensive account of the field here. Instead, we have chosen to narrow our focus to those areas of cryptography having the most practical relevance to the problem of *secure communication*. Broadly speaking, secure communication encompasses two complementary goals: the **secrecy** (sometimes called "privacy") and **integrity** of communicated data. These terms can be illustrated using the simple example of a user A sending a message m to a user B over a public channel. In the simplest sense, techniques for data secrecy ensure that an eavesdropping adversary (i.e., an adversary who sees all communication occurring on the channel) cannot get any information about m and, in particular, cannot determine m. Viewed in this way, such techniques protect against a *passive* adversary who listens to — but does not otherwise interfere with — the parties' communication. Techniques for data integrity, on the other hand, protect against an *active* adversary who may arbitrarily modify the data sent over the channel or may interject messages of his own. Here, secrecy is not necessarily an issue; instead, security in this setting requires only that any modifications performed by the adversary to the transmitted data will be detected by the receiving party.

In the cases of both secrecy and integrity, two different assumptions regarding the initial setup of the communicating parties can be considered. In the **private-key** setting (also known as the "shared-key," "secret-key," or "symmetric-key" setting), the assumption is that parties A and B have securely shared a random key s in advance. This key, which is completely hidden from the adversary, is used to secure their future communication. (We do not comment further on how such a key might be securely generated and shared; for our purposes, it is simply an assumption of the model.) Techniques for secrecy in this setting are called **private-key encryption** schemes, and those for data integrity are termed **message authentication codes (MACs)**.

In the **public-key** setting, the assumption is that one (or both) of the parties has generated a pair of keys: a *public key* that is widely disseminated throughout the network and an associated *secret key* that is kept private. The parties generating these keys may now use them to ensure secret communication using a **public-key encryption** scheme; they can also use these keys to provide data integrity (for messages they send) using a **digital signature scheme**.

We stress that, in the public-key setting, widespread distribution of the public key is assumed to occur before any communication over the public channel and without any interference from the adversary. In particular, if A generates a public/secret key, then B (for example) knows the correct public key and can use this key when communicating with A. On the flip side, the fact that the public key is widely disseminated implies that the adversary also knows the public key, and can attempt to use this knowledge when attacking the parties' communication.

We examine each of the above topics in turn. In Section 9.2 we introduce the information-theoretic approach to cryptography, describe some information-theoretic solutions for the above tasks, and discuss the severe limitations of this approach. We then describe the modern, computational (or complexity-theoretic) approach to cryptography that will be used in the remaining sections. This approach requires computational "hardness" assumptions of some sort; we formalize these assumptions in Section 9.3 and thus provide cryptographic building blocks for subsequent constructions. These building blocks are used to construct some basic cryptographic primitives in Section 9.4.

With these primitives in place, we proceed in the remainder of the chapter to give solutions for the tasks previously mentioned. Sections 9.5 and 9.6 discuss private-key encryption and message authentication, respectively, thereby completing our discussion of the private-key setting. Public-key encryption and digital signature schemes are described in Sections 9.7 and 9.8. We conclude with some suggestions for further reading.

9.2 Cryptographic Notions of Security

Two central features distinguish modern cryptography from "classical" (i.e., pre-1970s) cryptography: precise definitions and rigorous proofs of security. Without a precise definition of security for a stated goal, it is meaningless to call a particular protocol "secure." The importance of rigorous proofs of security (based on a set of well-defined assumptions) should also be clear: if a given protocol is not proven secure, there is always the risk that the protocol can be "broken." That protocol designers have not been able to find an attack does not preclude a more clever adversary from doing so. A proof that a given protocol is secure (with respect to some precise definition and using clearly stated assumptions) provides much more confidence in the protocol.

9.2.1 Information-Theoretic Notions of Security

With this in mind, we present one possible definition of security for private-key encryption and explore what can be achieved with respect to this definition. Recall the setting: two parties A and B share a random secret key s; this key will be used to secure their future communication and is completely hidden from the adversary. The data that A wants to communicate to B is called the **plaintext**, or simply the *message*. To transmit this message, A will **encrypt** the message using s and an encryption algorithm $\mathcal{E}$, resulting in **ciphertext** C. We write this as $C = \mathcal{E}_s(m)$. This ciphertext is sent over the public channel to B. Upon receiving the ciphertext, B recovers the original message by **decrypting** it using s and decryption algorithm $\mathcal{D}$; we write this as $m = \mathcal{D}_s(C)$.

We stress that the adversary is assumed to know the encryption and decryption algorithms; the only information hidden from the adversary is the secret key s. It is a mistake to require that the details of the encryption scheme be hidden in order for it to be secure, and modern cryptosystems are designed to be secure even when the full details of all algorithms are publicly available.

A plausible definition of security is to require that an adversary who sees ciphertext C (recall that C is sent over a public channel) — but does not know s — learns no information about the message m. In particular, even if the message m is known to be one of two possible messages m_1, m_2 (each being chosen with probability 1/2), the adversary should not learn which of these two messages was actually sent. If we abstract this by requiring the adversary to, say, output "1" when he believes that m_1 was sent, this requirement can be formalized as:

> For all possible m_1, m_2 and for any adversary A, the probability that A guesses "1" when C is an encryption of m_1 is equal to the probability that A guesses "1" when C is an encryption of m_2.

That is, the adversary is no more likely to guess that m_1 was sent when m_1 is the actual message than when m_2 is the actual message. An encryption scheme satisfying this definition is said to be *information-theoretically secure* or to achieve *perfect secrecy*.

Perfect secrecy can be achieved by the **one-time pad** encryption scheme, which works as follows. Let ℓ be the length of the message m, where m is viewed as a binary string. The parties share in advance a secret key s that is uniformly distributed over strings of length ℓ (i.e., s is an ℓ-bit string chosen uniformly at random). To encrypt message m, the sender computes $C = m \oplus s$ where $\oplus$ represents binary exclusive-or and is computed bit-by-bit. Decryption is performed by setting $m = C \oplus s$. Clearly, decryption always recovers the original message. To see that the scheme is perfectly secret, let M, C, K be random variables denoting the message, ciphertext, and key, respectively, and note that for *any* message m and observed ciphertext c, we have:

$$\begin{aligned}\Pr[M = m|C = c] &= \frac{\Pr[C = c|M = m]\Pr[M = m]}{\Pr[C = c]} \\ &= \frac{\Pr[K = c \oplus m]\Pr[M = m]}{\Pr[C = c]} = \frac{2^{-\ell}\Pr[M = m]}{\Pr[C = c]}\end{aligned}$$

Thus, if m_1, m_2 have equal *a priori* probability, then $\Pr[M = m_1|C = c] = \Pr[M = m_2|C = c]$ and the ciphertext gives no further information about the actual message sent.

While this scheme is provably secure, it has limited value for most common applications. For one, *the length of the shared key is equal to the length of the message*. Thus, the scheme is simply impractical when long messages are sent. Second, it is easy to see that the scheme is secure *only when it is used to send a single message* (hence the name "one-time pad"). This will not do for applications is which multiple messages must be sent. Unfortunately, it can be shown that the one-time pad is optimal if perfect secrecy is desired. More formally, any scheme achieving perfect secrecy requires the key to be at least as long as the (total) length of all messages sent.

Can information-theoretic security be obtained for other cryptographic goals? It is known that perfectly-secure message authentication is possible (see, e.g., [51, Section 4.5]), although constructions achieving perfect security are similarly inefficient and require impractically long keys to authenticate multiple messages. In the public-key setting, the situation is even worse: perfectly secure public-key encryption or digital signature schemes are simply unachievable.

In summary, it is impossible to design perfectly secure yet practical protocols achieving the basic goals outlined in Section 9.1. To obtain reasonable solutions for our original goals, it will be necessary to (slightly) relax our definition of security.

9.2.2 Toward a Computational Notion of Security

The observation noted at the end of the previous section has motivated a shift in modern cryptography toward *computational* notions of security. Informally, whereas information-theoretic security guarantees that a scheme is absolutely secure against all (even arbitrarily powerful) adversaries, computational security ensures that a scheme is secure except with "negligible" probability against all "efficient" adversaries (we formally define these terms below). Although information-theoretic security is a strictly stronger notion, computational security suffices in practice and allows the possibility of more efficient schemes. However, it should be noted that computational security ultimately relies on currently unproven assumptions regarding the computational "hardness" of certain problems; that is, the security guarantee provided in the computational setting is not as iron-clad as the guarantee given by information-theoretic security.

In moving to the computational setting, we introduce a *security parameter* $k \in \mathbb{N}$ that will be used to precisely define the terms "efficient" and "negligible." An *efficient* algorithm is defined as a probabilistic algorithm that runs in time polynomial in k; we also call such an algorithm "probabilistic, polynomial-time (PPT)." A *negligible* function is defined as one asymptotically smaller than any inverse polynomial; that is, a function $\varepsilon : \mathbb{N} \to \mathbb{R}^+$ is negligible if, for all $c \geq 0$ and for all n large enough, $\varepsilon(n) < 1/n^c$.

A cryptographic construction will be indexed by the security parameter k, where this value is given as input (in unary) to the relevant algorithms. Of course, we will require that these algorithms are all efficient and run in time polynomial in k. A typical definition of security in the computational setting requires that some condition hold for all PPT adversaries with all but negligible probability or, equivalently, that a PPT adversary will succeed in "breaking" the scheme with at most negligible probability. Note that the security parameter can be viewed as corresponding to a higher level of security (in some sense) because, as the security parameter increases, the adversary may run for a longer amount of time but has even lower probability of success.

Computational definitions of this sort will be used throughout the remainder of this chapter, and we explicitly contrast this type of definition with an information-theoretic one in Section 9.5 (for the case of private-key encryption).

9.2.3 Notation

Before continuing, we introduce some mathematical notation (following [30]) that will provide some useful shorthand. If A is a deterministic algorithm, then $y = A(x)$ means that we set y equal to the output of A on input x. If A is a probabilistic algorithm, the notation $y \leftarrow A(x_1, x_2, \ldots)$ denotes running A on inputs $x_1, x_2, \ldots$ and setting y equal to the output of A. Here, the "$\leftarrow$" is an explicit reminder that the process is probabilistic, and thus running A twice on the same inputs, for example, may not necessarily give the same value for y. If S represents a finite set, then $b \leftarrow S$ denotes assigning b an element chosen uniformly at random from S. If $p(x_1, x_2, \ldots)$ is a predicate that is either true or false, the notation

$$\Pr\left[x_1 \leftarrow S; x_2 \leftarrow A(x_1, y_2, \ldots); \cdots : p(x_1, x_2, \ldots)\right]$$

denotes the probability that $p(x_1, x_2, \ldots)$ is true after ordered execution of the listed experiment. The key features of this notation are that everything to the left of the colon represents the experiment itself (whose components are executed in order, from left to right, and are separated by semicolons) and the predicate is written to the right of the colon. To give a concrete example: $\Pr[b \leftarrow \{0, 1, 2\} : b = 2]$ denotes the probability that b is equal to 2 following the experiment in which b is chosen at random from $\{0, 1, 2\}$; this probability is, of course, $1/3$.

The notation $\{0, 1\}^\ell$ denotes the set of binary strings of length ℓ, while $\{0, 1\}^{\leq \ell}$ denotes the set of binary strings of length at most ℓ. We let $\{0, 1\}^*$ denote the set of finite-length binary strings. 1^k represents k repetitions of the digit "1", and has the value k in unary notation.

We assume familiarity with basic algebra and number theory on the level of [11]. We let $\mathbb{Z}_N = \{0, \ldots, N-1\}$ denote the set of integers modulo N; also, $\mathbb{Z}_N^* \subset \mathbb{Z}_N$ is the set of integers between 0 and N that are relatively prime to N. The Euler totient function is defined as $\varphi(N) \stackrel{\text{def}}{=} |\mathbb{Z}_N^*|$; of importance here is that $\varphi(p) = p - 1$ for p prime, and $\varphi(pq) = (p-1)(q-1)$ if p, q are distinct primes. For any N, the set $\mathbb{Z}_N^*$ forms a group under multiplication modulo N [11].

9.3 Building Blocks

As hinted at previously, cryptography seeks to exploit the presumed existence of computationally "hard" problems. Unfortunately, the mere existence of computationally hard problems does not appear to be sufficient for modern cryptography as we know it. Indeed, it is not currently known whether it is possible to have, say, secure private-key encryption (in the sense defined in Section 9.5) based only on the conjecture that $P \neq NP$ (where P refers to those problems solvable in polynomial time and NP [informally] refers to those problems whose solutions can be verified in polynomial time; cf. [50] and Chapter 6). Seemingly stronger assumptions are currently necessary in order for cryptosystems to be built. On the other hand — fortunately for cryptographers — such assumptions currently seem very reasonable.

9.3.1 One-Way Functions

The most basic building block in cryptography is a **one-way function**. Informally, a one-way function f is a function that is "easy" to compute but "hard" to invert. Care must be taken, however, in interpreting this informal characterization. In particular, the formal definition of one-wayness requires that f be hard to invert *on average* and not merely hard to invert *in the worst case*. This is in direct contrast to the situation in complexity theory, where a problem falls in a particular class based on the worst-case complexity of solving it (and this is one reason why $P \neq NP$ does not seem to be sufficient for much of modern cryptography).

A number of equivalent definitions of one-way functions are possible; we present one such definition here. Note that the security parameter is explicitly given as input (in unary) to all algorithms.

Definition 9.1 Let $F = \{f_k : \mathcal{D}_k \rightarrow \mathcal{R}_k\}_{k \geq 1}$ be an infinite collection of functions where $\mathcal{D}_k \subseteq \{0,1\}^{\leq \ell(k)}$ for some fixed polynomial $\ell(\cdot)$. Then F is one-way (more formally, F is a one-way function family) if the following conditions hold:

"Easy" to compute There is a deterministic, polynomial-time algorithm A such that for all k and for all $x \in \mathcal{D}_k$ we have $A(1^k, x) = f_k(x)$.

"Hard" to invert For all PPT algorithms B, the following is negligible (in k):

$$\Pr[x \leftarrow \mathcal{D}_k; y = f_k(x); x' \leftarrow B(1^k, y) : f_k(x') = y].$$

Efficiently sampleable There is a PPT algorithm S such that $S(1^k)$ outputs a uniformly distributed element of $\mathcal{D}_k$.

It is not hard to see that the existence of a one-way function family implies $P \neq NP$. Thus, we have no hope of proving the unequivocal existence of a one-way function family given our current knowledge of complexity theory. Yet, certain number-theoretic problems appear to be one-way (and have thus far resisted all attempts at proving otherwise); we mention three popular candidates:

1. **Factoring.** Let $\mathcal{D}_k$ consist of pairs of k-bit primes, and define f_k such that $f_k(p, q) = pq$. Clearly, this function is easy to compute. It is also true that the domain $\mathcal{D}_k$ is efficiently sampleable because efficient algorithms for generating random primes are known (see, e.g., Appendix A.7 in [14]). Finally, f_k is hard to invert — and thus the above construction is a one-way function family — under the conjecture that factoring is hard (we refer to this simply as "the factoring assumption"). Of course, we have no proof for this conjecture; rather, evidence favoring the conjecture comes from the fact that no polynomial-time algorithm for factoring has been discovered in roughly 300 years of research related to this problem.
2. **Computing discrete logarithms.** Let $\mathcal{D}_k$ consist of tuples (p, g, x) in which p is a k-bit prime, g is a generator of the multiplicative group $\mathbb{Z}_p^*$, and $x \in \mathbb{Z}_{p-1}$. Furthermore, define f_k such that $f_k(p, g, x) = (p, g, g^x \bmod p)$. Given p, g as above and for any $y \in \mathbb{Z}_p^*$, define $\log_g y$ as the unique value $x \in \mathbb{Z}_{p-1}$ such that $g^x = y \bmod p$ (that a unique such x exists follows from the fact that $\mathbb{Z}_p^*$ is a cyclic group for p prime). Although exponentiation modulo p can be done in time polynomial in the lengths of p and the exponent x, it is not known how to efficiently compute $\log_g y$ given p, g, y. This suggests that this function family is indeed one-way (we note that there exist algorithms to efficiently sample from $\mathcal{D}_k$; see e.g., Chapter 6 in [14]).

   It should be clear that the above construction generalizes to other collections of finite, cyclic groups in which exponentiation can be done in polynomial time. Of course, the function family thus defined is one-way only if the discrete logarithm problem in the relevant group is hard. Other popular examples in which this is believed to be the case include the group of points on certain elliptic curves (see Chapter 6 in [34]) and the subgroup of quadratic residues in $\mathbb{Z}_p^*$ when p and $\frac{p-1}{2}$ are both prime.

3. **RSA [45].** Let $\mathcal{D}_k$ consist of tuples (N, e, x), where N is a product of two distinct k-bit primes, $e < N$ is relatively prime to $\varphi(N)$, and $x \in \mathbb{Z}_N^*$. Furthermore, define f_k such that $f_k(N, e, x) = (N, e, x^e \bmod N)$. Following the previous examples, it should be clear that this function is easy to compute and has an efficiently sampleable domain (note that $\varphi(N)$ can be efficiently computed if p, q are known), It is conjectured that this function is hard to invert [45] and thus constitutes a one-way function family; we refer to this assumption simply as "the RSA assumption." For reasons of efficiency, the RSA function family is sometimes restricted by considering only $e = 3$ (and choosing N such that $\varphi(N)$ is not divisible by 3), and this is also believed to give a one-way function family.

   It is known that if RSA is a one-way function family, then factoring is hard (see the discussion of RSA as a trapdoor permutation, below). The converse is not believed to hold, and thus the RSA assumption appears to be strictly stronger than the factoring assumption (of course, all other things being equal, the weaker assumption is preferable).

9.3.2 Trapdoor Permutations

One-way functions are sufficient for many cryptographic applications. Sometimes, however, an "asymmetry" of sorts — whereby one party can efficiently accomplish some task which is infeasible for anyone else — must be introduced. **Trapdoor permutations** represent one way of formalizing this asymmetry. Recall that a one-way function has the property (informally) that it is "easy" to compute but "hard" to invert. Trapdoor permutations are also "easy" to compute and "hard" to invert *in general*; however, there is some trapdoor information that makes the permutation "easy" to invert. We give a formal definition now, and follow with some examples.

Definition 9.2 Let $\mathcal{K}$ be a PPT algorithm which, on input 1^k (for any $k \geq 1$), outputs a pair $(\mathsf{key}, \mathsf{td})$ such that key defines a permutation f_{key} over some domain $\mathcal{D}_{\mathsf{key}}$. We say $\mathcal{K}$ is a trapdoor permutation generator if the following conditions hold:

"Easy" to compute There is a deterministic, polynomial-time algorithm A such that for all k, all $(\mathsf{key}, \mathsf{td})$ output by $\mathcal{K}(1^k)$, and all $x \in \mathcal{D}_{\mathsf{key}}$ we have $A(1^k, \mathsf{key}, x) = f_{\mathsf{key}}(x)$.

"Hard" to invert For all PPT algorithms B, the following is negligible (in k):

$$\Pr[(\mathsf{key}, \mathsf{td}) \leftarrow \mathcal{K}(1^k); x \leftarrow \mathcal{D}_{\mathsf{key}}; y = f_{\mathsf{key}}(x); x' \leftarrow B(1^k, \mathsf{key}, y) : f_{\mathsf{key}}(x') = y].$$

Efficiently sampleable There is a PPT algorithm S such that for all $(\mathsf{key}, \mathsf{td})$ output by $\mathcal{K}(1^k)$, $S(1^k, \mathsf{key})$ outputs a uniformly distributed element of $\mathcal{D}_{\mathsf{key}}$.

"Easy" to invert with trapdoor There is a deterministic, polynomial-time algorithm I such that for all $(\mathsf{key}, \mathsf{td})$ output by $\mathcal{K}(1^k)$ and all $y \in \mathcal{D}_{\mathsf{key}}$ we have $I(1^k, \mathsf{td}, y) = f_{\mathsf{key}}^{-1}(y)$.

It should be clear that the existence of a trapdoor permutation generator immediately implies the existence of a one-way function family. Note that one could also define the completely analogous notion of trapdoor *function* generators; however, these have (thus far) had much more limited applications to cryptography.

It seems that the existence of a trapdoor permutation generator is a strictly stronger assumption than the existence of a one-way function family. Yet, number theory again provides examples of (conjectured) candidates:

9.3.2.1 RSA

We have seen in the previous section that RSA gives a one-way function family. It can also be used to give a trapdoor permutation generator. Here, we let $\mathcal{K}$ be an algorithm which, on input 1^k, chooses two distinct

k-bit primes p, q at random, sets $N = pq$, and chooses $e < N$ such that e and $\varphi(N)$ are relatively prime (note that $\varphi(N) = (p-1)(q-1)$ is efficiently computable because the factorization of N is known to $\mathcal{K}$). Then, $\mathcal{K}$ computes d such that $ed = 1 \bmod \varphi(N)$. The output is $((N, e), d)$, where (N, e) defines the permutation $f_{N,e} : \mathbb{Z}_N^* \to \mathbb{Z}_N^*$ given by $f_{N,e}(x) = x^e \bmod N$. It is not hard to verify that this is indeed a permutation. That this permutation satisfies the first three requirements of the definition above follows from the fact that RSA is a one-way function family. To verify the last condition ("easiness" of inversion given the trapdoor d), note that

$$f_{N,d}(x^e \bmod N) = (x^e)^d \bmod N = x^{ed \bmod \varphi(N)} \bmod N = x,$$

and thus $f_{N,d} = f_{N,e}^{-1}$. So, the permutation $f_{N,e}$ can be efficiently inverted given d.

9.3.2.2 A Trapdoor Permutation Based on Factoring [42]

Let $\mathcal{K}$ be an algorithm which, on input 1^k, chooses two distinct k-bit primes p, q at random such that $p = q = 3 \bmod 4$, and sets $N = pq$. The output is $(N, (p, q))$, where N defines the permutation $f_N : \mathcal{QR}_N \to \mathcal{QR}_N$ given by $f_N(x) = x^2 \bmod N$; here, $\mathcal{QR}_N$ denotes the set of quadratic residues modulo N (i.e., the set of $x \in \mathbb{Z}_N^*$ such that x is a square modulo N). It can be shown that f_N is a permutation, and it is immediate that f_N is easy to compute. $\mathcal{QR}_N$ is also efficiently sampleable: to choose a random element in $\mathcal{QR}_N$, simply pick a random $x \in \mathbb{Z}_N^*$ and square it. It can also be shown that the trapdoor information p, q (i.e., the factorization of N) is sufficient to enable efficient inversion of f_N (see Section 3.6 in [14]). We now prove that this permutation is hard to invert as long as factoring is hard.

Lemma 9.1 *Assuming the hardness of factoring N of the form generated by $\mathcal{K}$, algorithm $\mathcal{K}$ described above is a trapdoor permutation family.*

Proof The lemma follows by showing that the squaring permutation described above is hard to invert (without the trapdoor). For any PPT algorithm B, define

$$\delta(k) \stackrel{\text{def}}{=} \Pr[(N, (p, q)) \leftarrow \mathcal{K}(1^k); y \leftarrow \mathcal{QR}_N; z \leftarrow B(1^k, N, y) : z^2 = y \bmod N]$$

(this is exactly the probability that B inverts a randomly-generated f_N). We use B to construct another PPT algorithm B' which factors the N output by $\mathcal{K}$. Algorithm B' operates as follows: on input $(1^k, N)$, it chooses a random $\tilde{x} \in \mathbb{Z}_N^*$ and sets $y = \tilde{x}^2 \bmod N$. It then runs $B(1^k, N, y)$ to obtain output z. If $z^2 = y \bmod N$ and $z \neq \pm\tilde{x}$, we claim that $\gcd(z - \tilde{x}, N)$ is a nontrivial factor of N. Indeed, $z^2 - \tilde{x}^2 = 0 \bmod N$, and thus

$$(z - \tilde{x})(z + \tilde{x}) = 0 \bmod N.$$

Since $z \neq \pm\tilde{x}$, it must be the case that $\gcd(z - \tilde{x}, N)$ gives a nontrivial factor of N, as claimed.

Now, conditioned on the fact that $z^2 = y \bmod N$ (which is true with probability $\delta(k)$), the probability that $z \neq \pm\tilde{x}$ is exactly 1/2; this follows from the fact that y has exactly four square roots, two of which are $\tilde{x}$ and $-\tilde{x}$. Thus, the probability that B' factors N is exactly $\delta(k)/2$. Because this quantity is negligible under the factoring assumption, $\delta(k)$ must be negligible as well. □

9.4 Cryptographic Primitives

The building blocks of the previous section can be used to construct a variety of primitives, which in turn have a wide range of applications. We explore some of these primitives here.

9.4.1 Pseudorandom Generators

Informally, a **pseudorandom generator (PRG)** is a deterministic function that takes a short, random string as input and returns a longer, "random-looking" (i.e., pseudorandom) string as output. But to properly understand this, we must first ask: what does it mean for a string to "look random"? Of course, it is meaningless (in the present context) to talk about the "randomness" of any particular string — once a string is fixed, it is no longer random! Instead, we must talk about the randomness — or pseudorandomness — of a *distribution* of strings. Thus, to evaluate $G : \{0,1\}^k \to \{0,1\}^{k+1}$ as a PRG, we must compare the uniform distribution on strings of length $k+1$ with the distribution $\{G(x)\}$ for x chosen uniformly at random from $\{0,1\}^k$.

It is rather interesting that although the design and analysis of PRGs has a long history [33], it was not until the work of [9, 54] that a definition of PRGs appeared which was satisfactory for cryptographic applications. Prior to this work, the quality of a PRG was determined largely by ad hoc techniques; in particular, a PRG was deemed "good" if it passed a specific battery of statistical tests (for example, the probability of a "1" in the final bit of the output should be roughly 1/2). In contrast, the approach advocated by [9, 54] is that a PRG is good if it passes *all possible* (efficient) statistical tests! We give essentially this definition here.

Definition 9.3 Let $G : \{0,1\}^* \to \{0,1\}^*$ be an efficiently computable function for which $|G(x)| = \ell(|x|)$ for some fixed polynomial $\ell(k) > k$ (i.e., fixed-length inputs to G result in fixed-length outputs, and the output of G is always longer than its input). We say G is a pseudorandom generator (PRG) with expansion factor $\ell(k)$ if the following is negligible (in k) for all PPT statistical tests T:

$$\left|\Pr[x \leftarrow \{0,1\}^k : T(G(x)) = 1] - \Pr[y \leftarrow \{0,1\}^{\ell(k)} : T(y) = 1]\right|.$$

Namely, no PPT algorithm can distinguish between the output of G (on uniformly selected input) and the uniform distribution on strings of the appropriate length.

Given this strong definition, it is somewhat surprising that PRGs can be constructed at all; yet, they can be constructed from any one-way function (see below). As a step toward the construction of PRGs based on general assumptions, we first define and state the existence of a *hard-core bit* for any one-way function. Next, we show how this hard-core bit can be used to construct a PRG from any one-way *permutation*. (The construction of a PRG from arbitrary one-way *functions* is more complicated and is not given here.) This immediately extends to give explicit constructions of PRGs based on some specific assumptions.

Definition 9.4 Let $F = \{f_k : \mathcal{D}_k \to \mathcal{R}_k\}_{k\geq 1}$ be a one-way function family, and let $H = \{h_k : \mathcal{D}_k \to \{0,1\}\}_{k\geq 1}$ be an efficiently computable function family. We say that H is a hard-core bit for F if $h_k(x)$ is hard to predict with probability significantly better than 1/2 given $f_k(x)$. More formally, H is a hard-core bit for F the following is negligible (in k) for all PPT algorithms A:

$$\left|\Pr[x \leftarrow \mathcal{D}_k; y = f_k(x) : A(1^k, y) = h_k(x)] - 1/2\right|.$$

(Note that this is the "best" one could hope for in a definition of this sort, since an algorithm that simply outputs a random bit will guess $h_k(x)$ correctly half the time.)

We stress that *not* every H is a hard-core bit for a given one-way function family F. To give a trivial example: for the one-way function family based on factoring (in which $f_k(p,q) = pq$), it is easy to predict the last bit of p (and also q), which is always 1! On the other hand, a one-way function family with a hard-core bit can be constructed from any one-way function family; we state the following result to that effect without proof.

Theorem 9.2 ([27]) *If there exists a one-way function family F, then there exists (constructively) a one-way function family F' and an H which is a hard-core bit for F'.*

Hard-core bits for specific functions are known without recourse to the general theorem above [1, 9, 21, 32, 36]. We discuss a representative result for the case of RSA (this function family was introduced in Section 9.3, and we assume the reader is familiar with the notation used there). Let $H = \{h_k\}$ be a function family such that $h_k(N, e, x)$ returns the least significant bit of x mod N. Then H is a hard-core bit for RSA [1, 21]. Reiterating the definition above and assuming that RSA is a one-way function family, this means that given N, e, and x^e mod N (for randomly chosen N, e, and x from the appropriate domains), it is hard for any PPT algorithm to compute the least significant bit of x mod N with probability better than 1/2.

We show now a construction of a PRG with expansion factor $k + 1$ based on any one-way *permutation* family $F = \{f_k\}$ with hard-core bit $H = \{h_k\}$. For simplicity, assume that the domain of f_k is $\{0, 1\}^k$; furthermore, for convenience, let $f(x), h(x)$ denote $f_{|x|}(x), h_{|x|}(x)$, respectively. Define:

$$G(x) = f(x) \circ h(x).$$

We claim that G is a PRG. As some intuition toward this claim, let $|x| = k$ and note that the first k bits of $G(x)$ are indeed uniformly distributed if x is uniformly distributed; this follows from the fact that f is a permutation over $\{0, 1\}^k$. Now, because H is a hard-core bit of F, $h(x)$ cannot be predicted by any efficient algorithm with probability better than 1/2 even when the algorithm is given $f(x)$. Informally, then, $h(x)$ "looks random" to a PPT algorithm even conditioned on the observation of $f(x)$; hence, the entire string $f(x) \circ h(x)$ is pseudorandom.

It is known that given any PRG with expansion factor $k + 1$, it is possible to construct a PRG with expansion factor $\ell(k)$ for any polynomial $\ell(\cdot)$. The above construction, then, may be extended to yield a PRG that expands its input by an essentially arbitrary amount. Finally, although the preceding discussion focused only on the case of one-way *permutations*, it can be generalized (with much difficulty!) for the more general case of one-way *functions*. Putting these known results together, we obtain:

Theorem 9.3 ([31]) *If there exists a one-way function family, then for any polynomial $\ell(\cdot)$, there exists a PRG with stretching factor $\ell(k)$.*

9.4.2 Pseudorandom Functions and Block Ciphers

A pseudorandom generator G takes a short random string x and yields a polynomially-longer pseudorandom string $G(x)$. This in turn is useful in many contexts; see Section 9.5 for an example. However, a PRG has the following "limitations." First, for $G(x)$ to be pseudorandom, it is necessary that (1) x be chosen uniformly at random and also that (2) x be unknown to the distinguishing algorithm (clearly, once x is known, $G(x)$ is determined and hence no longer looks random). Furthermore, a PRG generates pseudorandom output whose length must be polynomially related to that of the input string x. For some applications, it would be nice to circumvent these limitations in some way.

These considerations have led to the definition and development of a more powerful primitive: a (family of) **pseudorandom functions (PRFs)**. Informally, a PRF $F : \{0, 1\}^k \times \{0, 1\}^m \rightarrow \{0, 1\}^n$ is a keyed function, so that fixing a particular key $s \in \{0, 1\}^k$ may be viewed as defining a function $F_s : \{0, 1\}^m \rightarrow \{0, 1\}^n$. (For simplicity in the rest of this and the following paragraph, we let $m = n = k$ although in general $m, n = \text{poly}(k)$.) Informally, a PRF F acts like a random function in the following sense: no efficient algorithm can distinguish the input/output behavior of F (with a randomly chosen key which is fixed for the duration of the experiment) from the input/output behavior of a truly random function. We stress that this holds even when the algorithm is allowed to interact with the function in an arbitrary way. It may be helpful to picture the following imaginary experiment: an algorithm is given access to a box that implements a function over $\{0, 1\}^k$. The algorithm can send inputs of its choice to the box and observe the corresponding outputs, but may not experiment with the box in any other way. Then F is a PRF if no efficient algorithm can distinguish whether the box implements a truly random function over $\{0, 1\}^k$ (i.e., a function chosen uniformly at random from the space of all 2^{k2^k} functions over $\{0, 1\}^k$) or whether it implements an instance of F_s (for uniformly chosen key $s \in \{0, 1\}^k$).

Note that this primitive is much stronger than a PRG. For one, the key s can be viewed as encoding an *exponential* amount of pseudorandomness because, roughly speaking, $F_s(x)$ is an independent pseudorandom value for each $x \in \{0,1\}^k$. Second, note that $F_s(x)$ is pseudorandom even if x is known, and even if x was not chosen at random. Of course, it must be the case that the key s is unknown and is chosen uniformly at random. We now give a formal definition of a PRF.

Definition 9.5 Let $\mathcal{F} = \{F_s : \{0,1\}^{m(k)} \to \{0,1\}^{n(k)}\}_{k \geq 1; s \in \{0,1\}^k}$ be an efficiently computable function family where $m, n = \mathsf{poly}(k)$, and let $\mathsf{Rand}_{m(k)}^{n(k)}$ denote the set of all functions from $\{0,1\}^{m(k)}$ to $\{0,1\}^{n(k)}$. We say $\mathcal{F}$ is a pseudorandom function family (PRF) if the following is negligible in k for all PPT algorithms A:

$$\left| \Pr[s \leftarrow \{0,1\}^k : A^{F_s(\cdot)}(1^k) = 1] - \Pr[f \leftarrow \mathsf{Rand}_{m(k)}^{n(k)} : A^{f(\cdot)}(1^k) = 1] \right|,$$

where the notation $A^{f(\cdot)}$ denotes that A has oracle access to function f; that is, A can send (as often as it likes) inputs of its choice to f and receive the corresponding outputs.

We do not present any details about the construction of a PRF based on general assumptions, beyond noting that they can be constructed from any one-way function family.

Theorem 9.4 ([25]) *If there exists a one-way function family F, then there exists (constructively) a PRF $\mathcal{F}$.*

An efficiently computable permutation family $\mathcal{P} = \{P_s : \{0,1\}^{m(k)} \to \{0,1\}^{m(k)}\}_{k \geq 1; s \in \{0,1\}^k}$ is an efficiently computable function family for which P_s is a permutation over $\{0,1\}^{m(k)}$ for each $s \in \{0,1\}^k$; and furthermore P_s^{-1} is efficiently computable (given s). By analogy with the case of a PRF, we say that $\mathcal{P}$ is a pseudorandom permutation (PRP) if P_s (with s randomly chosen in $\{0,1\}^k$) is indistinguishable from a truly random *permutation* over $\{0,1\}^{m(k)}$. A pseudorandom permutation can be constructed from any pseudorandom function [37].

What makes PRFs and PRPs especially useful in practice (especially as compared to PRGs) is that very efficient implementations of (conjectured) PRFs are available in the form of **block ciphers**. A block cipher is an efficiently computable permutation family $\mathcal{P} = \{P_s : \{0,1\}^m \to \{0,1\}^m\}_{s \in \{0,1\}^k}$ *for which keys have a fixed length k*. Because keys have a fixed length, we can no longer speak of a "negligible function" or a "polynomial-time algorithm" and consequently there is no notion of asymptotic security for block ciphers; instead, concrete security definitions are used. For example, a block cipher is said to be a (t, ε)-secure PRP, say, if no adversary running in time t can distinguish P_s (for randomly chosen s) from a random permutation over $\{0,1\}^m$ with probability better than ε. See [3] for further details.

Block ciphers are particularly efficient because they are *not* based on number-theoretic or algebraic one-way function families but are instead constructed directly, with efficiency in mind from the outset. One popular block cipher is DES (the Data Encryption Standard) [17, 38], which has 56-bit keys and is a permutation on $\{0,1\}^{64}$. DES dates to the mid-1970s, and recent concerns about its security — particularly its relatively short key length — have prompted the development* of a new block cipher termed AES (the Advanced Encryption Standard). This cipher supports 128-, 192-, and 256-bit keys, and is a permutation over $\{0,1\}^{128}$. Details of the AES cipher and the rationale for its construction are available [13].

9.4.3 Cryptographic Hash Functions

Although hash functions play an important role in cryptography, our discussion will be brief and informal because they are used sparingly in the remainder of this survey.

Hash functions — functions that compress long, often variable-length strings to much shorter strings — are widely used in many areas of computer science. For many applications, constructions of hash functions

*See http://csrc.nist.gov/CryptoToolkit/aes/ for a history and discussion of the design competition resulting in the selection of a cipher for AES.

with the necessary properties are known to exist without any computational assumptions. For cryptography, however, hash functions with very strong properties are often needed; furthermore, it can be shown that the existence of a hash function with these properties would imply the existence of a one-way function family (and therefore any such construction must be based on a computational assumption of some sort). We discuss one such property here.

The security property that arises most often in practice is that of collision resistance. Informally, H is said to be a **collision-resistant hash function** if an adversary is unable to find a "collision" in H; namely, two inputs x, x' with $x \neq x'$ but $H(x) = H(x')$. As in the case of PRFs and block ciphers (see the previous section), we can look at either the asymptotic security of a function family $\mathcal{H} = \{H_s : \{0,1\}^* \rightarrow \{0,1\}^k\}_{k \geq 1; s \in \{0,1\}^k}$ or the concrete security of a fixed hash function $H : \{0,1\}^* \rightarrow \{0,1\}^m$. The former are constructed based on specific computational assumptions, while the latter (as in the case of block ciphers) are constructed directly and are therefore much more efficient.

It is not hard to show that a collision-resistant hash function family mapping arbitrary-length inputs to fixed-length outputs is itself a one-way function family. Interestingly, however, collision-resistant hash function families are believed to be impossible to construct based on (general) one-way function families or trapdoor permutation generators [49]. On the other hand, constructions of collision-resistant hash function families based on specific computational assumptions (e.g., the hardness of factoring) are known; see Section 10.2 in [14].

In practice, customized hash functions — designed with efficiency in mind and not derived from number-theoretic problems — are used. One well-known example is MD5 [44], which hashes arbitrary-length inputs to 128-bit outputs. Because collisions in *any* hash function with output length k can be found in expected time (roughly) $2^{k/2}$ via a "birthday attack" (see, for example, Section 3.4.2 in [14]) and because computations on the order of 2^{64} are currently considered just barely outside the range of feasibility, hash functions with output lengths longer than 128 bits are frequently used. A popular example is SHA-1 [19], which hashes arbitrary-length inputs to 160-bit outputs. SHA-1 is considered collision-resistant for practical purposes, given current techniques and computational ability.

Hash functions used in cryptographic protocols sometimes require properties stronger than collision resistance in order for the resulting protocol to be provably secure [5]. It is fair to say that, in many cases, the exact properties needed by the hash function are not yet fully understood.

9.5 Private-Key Encryption

As discussed in Section 9.2.1, perfectly secret private-key encryption is achievable using the one-time pad encryption scheme; however, perfectly secret encryption requires that the shared key be at least as long as the communicated message. Our goal was to beat this bound by considering computational notions of security instead. We show here that this is indeed possible.

Let us first see what a definition of computational secrecy might involve. In the case of perfect secrecy, we required that for all messages m_0, m_1 of the same length ℓ, *no possible* algorithm could distinguish *at all* whether a given ciphertext is an encryption of m_0 or m_1. In the notation we have been using, this is equivalent to requiring that for all adversaries A,

$$\left|\Pr[s \leftarrow \{0,1\}^\ell : A(\mathcal{E}_s(m_0)) = 1] - \Pr[s \leftarrow \{0,1\}^\ell : A(\mathcal{E}_s(m_1)) = 1]\right| = 0.$$

To obtain a computational definition of security, we make two modifications: (1) we require the above to hold only for *efficient* (i.e., PPT) algorithms A; and (2) we only require the "distinguishing advantage" of the algorithm to be *negligible*, and not necessarily 0. The resulting definition of computational secrecy is that for all PPT adversaries A, the following is negligible:

$$\left|\Pr[s \leftarrow \{0,1\}^k : A(1^k, \mathcal{E}_s(m_0)) = 1] - \Pr[s \leftarrow \{0,1\}^k : A(1^k, \mathcal{E}_s(m_1)) = 1]\right|. \tag{9.1}$$

The one-time pad encryption scheme, together with the notion of a PRG as defined in Section 9.4.1, suggest a computationally secret encryption scheme in which the shared key is shorter than the message (we reiterate that this is simply not possible if perfect secrecy is required). Specifically, let G be a PRG with expansion factor $\ell(k)$ (recall $\ell(k)$ is a polynomial with $\ell(k) > k$). To encrypt a message of length $\ell(k)$, the parties share a key s of length k; message m is then encrypted by computing $C = m \oplus G(s)$. Decryption is done by simply computing $m = C \oplus G(s)$.

For some intuition as to why this is secure, note that the scheme can be viewed as implementing a "pseudo"-one-time pad in which the parties share the pseudorandom string $G(s)$ instead of a uniformly random string of the same length. (Of course, to minimize the secret key length, the parties actually share s and regenerate $G(s)$ when needed.) But because the pseudorandom string $G(s)$ "looks random" to a PPT algorithm, the pseudo-one-time pad scheme "looks like" the one-time pad scheme to any PPT adversary. Because the one-time pad scheme is secure, so is the pseudo-one-time pad. (This is not meant to serve as a rigorous proof, but can easily be adapted to give one.)

We re-cap the discussion thus far in the following lemma.

Lemma 9.5 *Perfectly secret encryption is possible if and only if the shared key is at least as long as the message. However, if there exists a PRG, then there exists a computationally secret encryption scheme in which the message is (polynomially) longer than the shared key.*

Let us examine the pseudo-one-time pad encryption scheme a little more critically. Although the scheme allows encrypting messages longer than the secret key, the scheme is secure only when it is used *once* (as in the case of the one-time pad). Indeed, if an adversary views ciphertexts $C_1 = m_1 \oplus G(s)$ and $C_2 = m_2 \oplus G(s)$ (where m_1 and m_2 are unknown), the adversary can compute $m_1 \oplus m_2 = C_1 \oplus C_2$ and hence learn something about the relation between the two messages. Even worse, if the adversary somehow learns (or later determines), say, m_1, then the adversary can compute $G(s) = C_1 \oplus m_1$ and can thus decrypt any ciphertexts subsequently transmitted. We stress that such attacks (called *known-plaintext attacks*) are not merely of academic concern, because there are often messages sent whose values are uniquely determined, or known to lie in a small range. Can we obtain secure encryption even in the face of such attacks?

Before giving a scheme that prevents such attacks, let us precisely formulate a definition of security. First, the scheme should be "secure" even when used to encrypt multiple messages; in particular, an adversary who views the ciphertexts corresponding to multiple messages should not learn any information about the relationships among these messages. Second, the secrecy of the scheme should remain intact if some encrypted messages are known by the adversary. In fact, we can go beyond this last requirement and mandate that the scheme remain "secure" even if the adversary can request the encryption of messages of his choice (a *chosen-plaintext attack* of this sort arises when an adversary can influence the messages sent).

We model chosen-plaintext attacks by giving the adversary unlimited and unrestricted access to an *encryption oracle* denoted $\mathcal{E}_s(\cdot)$. This is simply a "black-box" that, on inputting a message m, returns an encryption of m using key s (in case $\mathcal{E}$ is randomized, the oracle chooses fresh randomness each time). Note that the resulting attack is perhaps stronger than what a real-world adversary can do (a real-world adversary likely cannot request as many encryptions — of arbitrary messages — as he likes); by the same token, if we can construct a scheme secure against this attack, then certainly the scheme will be secure in the real world. A formal definition of security follows.

Definition 9.6 A private-key encryption scheme $(\mathcal{E}, \mathcal{D})$ is said to be secure against chosen-plaintext attacks if, for all messages m_1, m_2 and all PPT adversaries A, the following is negligible:

$$\left|\Pr[s \leftarrow \{0,1\}^k : A^{\mathcal{E}_s(\cdot)}(1^k, \mathcal{E}_s(m_1)) = 1] - \Pr[s \leftarrow \{0,1\}^k : A^{\mathcal{E}_s(\cdot)}(1^k, \mathcal{E}_s(m_2)) = 1]\right|.$$

Namely, a PPT adversary cannot distinguish between the encryption of m_1 and m_2 *even if the adversary is given unlimited access to an encryption oracle.*

We stress one important corollary of the above definition: an encryption scheme secure against chosen-plaintext attacks *must be randomized* (in particular, the one-time pad does not satisfy the above definition).

This is so for the following reason: if the scheme were deterministic, an adversary could obtain $C_1 = \mathcal{E}_s(m_1)$ and $C_2 = \mathcal{E}_s(m_2)$ from its encryption oracle and then compare the given ciphertext to each of these values; thus, the adversary could immediately tell which message was encrypted. Our strong definition of security forces us to consider more complex encryption schemes.

Fortunately, many encryption schemes satisfying the above definition are known. We present two examples here; the first is mainly of theoretical interest (but is also practical for short messages), and its simplicity is illuminating. The second is more frequently used in practice.

Our first encryption scheme uses a key of length k to encrypt messages of length k (we remind the reader, however, that this scheme will be a tremendous improvement over the one-time pad because the present scheme can be used to encrypt polynomially-many messages). Let $\mathcal{F} = \{F_s : \{0,1\}^k \to \{0,1\}^k\}_{k \geq 1; s \in \{0,1\}^k}$ be a PRF (cf. Section 9.4.2); alternatively, one can think of k as being fixed and using a block cipher for $\mathcal{F}$ instead. We define encryption using key s as follows [26]: on input a message $m \in \{0,1\}^k$, choose a random $r \in \{0,1\}^k$ and output $\langle r, F_s(r) \oplus m \rangle$. To decrypt ciphertext $\langle r, c \rangle$ using key s, simply compute $m = c \oplus F_s(r)$.

We give some intuition for the security of this scheme against chosen-plaintext attacks. Assume the adversary queries the encryption oracle n times, receiving in return the ciphertexts $\langle r_1, c_1 \rangle, \ldots, \langle r_n, c_n \rangle$ (the messages to which these ciphertexts correspond are unimportant). Let the ciphertext given to the adversary — corresponding to the encryption of either m_1 or m_2 — be $\langle r, c \rangle$. By the definition of a PRF, the value $F_s(r)$ "looks random" to the PPT adversary A unless $F_s(\cdot)$ was previously computed on input r; in other words, $F_s(r)$ "looks random" to A unless $r \in \{r_1, \ldots, r_n\}$ (we call this occurrence a *collision*). Security of the scheme is now evident from the following: (1) assuming a collision does *not* occur, $F_s(r)$ is pseudorandom as discussed and hence the adversary cannot determine whether m_1 or m_2 was encrypted (as in the one-time pad scheme); furthermore, (2) the probability that a collision occurs is $\frac{n}{2^k}$, which is negligible (because n is polynomial in k). We thus have Theorem 9.6.

Theorem 9.6 ([26]) *If there exists a PRF $\mathcal{F}$, then there exists an encryption scheme secure against chosen-plaintext attacks.*

The previous construction applies to small messages whose length is equal to the output length of the PRF. From a theoretical point of view, an encryption scheme (secure against chosen-plaintext attacks) for longer messages follows immediately from the construction given previously; namely, to encrypt message $M = m_1, \ldots, m_\ell$ (where $m_i \in \{0,1\}^k$), simply encrypt each block of the message using the previous scheme, giving ciphertext:

$$\langle r_1, F_s(r_1) \oplus m_1, \ldots, r_\ell, F_s(r_\ell) \oplus m_\ell \rangle.$$

This approach gives a ciphertext twice as long as the original message and is therefore not very practical.

A better idea is to use a **mode of encryption**, which is a method for encrypting long messages using a block cipher with fixed input/output length. Four modes of encryption were introduced along with DES [18], and we discuss one such mode here (not all of the DES modes of encryption are secure). In cipher block chaining (CBC) mode, a message $M = m_1, \ldots, m_\ell$ is encrypted using key s as follows:

```
Choose $C_0 \in \{0,1\}^k$ at random
For $i = 1$ to $\ell$:
  $C_i = F_s(m_i \oplus C_{i-1})$
Output $\langle C_0, C_1, \ldots, C_\ell \rangle$
```

Decryption of a ciphertext $\langle C_0, \ldots, C_\ell \rangle$ is done by reversing the above steps:

```
For $i = 1$ to $\ell$:
  $m_i = F_s^{-1}(C_i) \oplus C_{i-1}$
Output $m_1, \ldots, m_\ell$
```

It is known that CBC mode is secure against chosen-plaintext attacks [3].

9.6 Message Authentication

The preceding section discussed how to achieve message *secrecy*; we now discuss techniques for message *integrity*. In the private-key setting, this is accomplished using message authentication codes (MACs). We stress that secrecy and authenticity are two incomparable goals, and it is certainly possible to achieve either one without the other. As an example, the one-time pad — which achieves perfect secrecy — provides no message integrity whatsoever because *any* ciphertext C of the appropriate length decrypts to some valid message. Even worse, if C represents the encryption of a particular message m (so that $C = m \oplus s$ where s is the shared key), then flipping the first bit of C has the effect of flipping the first bit of the resulting decrypted message.

Before continuing, let us first define the semantics of a MAC.

Definition 9.7 A message authentication code consists of a pair of PPT algorithms $(\mathcal{T}, \mathsf{Vrfy})$ such that (here, the length of the key is taken to be the security parameter):

- The tagging algorithm $\mathcal{T}$ takes as input a key s and a message m and outputs a tag $t = \mathcal{T}_s(m)$.
- The verification algorithm Vrfy takes as input a key s, a message m, and a (purported) tag t and outputs a bit signifying acceptance (1) or rejection (0).

We require that for all m and all t output by $\mathcal{T}_s(m)$ we have $\mathsf{Vrfy}_s(m, t) = 1$.

Actually, a MAC should also be defined over a particular message space and this must either be specified or else clear from the context.

Schemes designed to detect "random" modifications of a message (e.g., error-correcting codes) do not constitute secure MACs because they are not designed to provide message authenticity in an *adversarial* setting. Thus, it is worth considering carefully the exact security goal we desire. Ideally, even if an adversary can request tags for multiple messages $m_1, \ldots$ of his choice, it should be impossible for the adversary to "forge" a valid-looking tag t on a new message m. (As in the case of encryption, this adversary is likely stronger than what is encountered in practice; however, if we can achieve security against even this strong attack so much the better!) To formally model this, we give the adversary access to an oracle $\mathcal{T}_s(\cdot)$, which returns a tag t for any message m of the adversary's choice. Let $m_1, \ldots, m_\ell$ denote the messages submitted by the adversary to this oracle. We say a forgery occurs if the adversary outputs (m, t) such that $m \notin \{m_1, \ldots, m_\ell\}$ and $\mathsf{Vrfy}_s(m, t) = 1$. Finally, we say a MAC is secure if the probability of a forgery is negligible for all PPT adversaries A. For completeness, we give a formal definition following [4].

Definition 9.8 MAC $(\mathcal{T}, \mathsf{Vrfy})$ is said to be secure against adaptive chosen-message attacks if, for all PPT adversaries A, the following is negligible:

$$\Pr[s \leftarrow \{0,1\}^k; (m, t) \leftarrow A^{\mathcal{T}_s(\cdot)}(1^k) : \mathsf{Vrfy}_s(m, t) = 1 \wedge m \notin \{m_1, \ldots, m_\ell\}],$$

where $m_1, \ldots, m_\ell$ are the messages that A submitted to $\mathcal{T}_s(\cdot)$.

We now give two constructions of a secure MAC. For the first, let $\mathcal{F} = \{F_s : \{0,1\}^k \rightarrow \{0,1\}^k\}_{k \geq 1; s \in \{0,1\}^k}$ be a PRF (we can also let $\mathcal{F}$ be a block cipher for some fixed value k). The discussion of PRFs in Section 9.4.2 should motivate the following construction of a MAC for messages of length k [26]: the tagging algorithm $\mathcal{T}_s(m)$ (where $|s| = |m| = k$) returns $t = F_s(m)$, and the verification algorithm $\mathsf{Vrfy}_s(m, t)$ outputs 1 if and only if $F_s(m) = t$. A proof of security for this construction is immediate: Let $m_1, \ldots, m_\ell$ denote those messages for which adversary A has requested a tag from $\mathcal{T}_s(\cdot)$. Because $\mathcal{F}$ is a PRF, $\mathcal{T}_s(m) = F_s(m)$ "looks random" for any $m \notin \{m_1, \ldots, m_\ell\}$ (call m of this sort *new*). Thus, the adversary's probability of outputting (m, t) such that $t = F_s(m)$ and m is new is (roughly) 2^{-k}; that is, the probability of guessing the output of a random function with output length k at a particular point m. This is negligible, as desired.

Because PRFs exist for any (polynomial-size) input length, the above construction can be extended to achieve secure message authentication for polynomially-long messages. We summarize the theoretical implications of this result in Theorem 9.7.

Theorem 9.7 ([26]) *If there exists a PRF $\mathcal{F}$, then there exists a MAC secure against adaptive chosen-message attack.*

Although the above result gives a theoretical solution to the problem of message authentication (and can be made practical for short messages by using a block cipher to instantiate the PRF), it does not give a practical solution for authenticating long messages. So, we conclude this section by showing a practical and widely used MAC construction for long messages. Let $\mathcal{F} = \{F_s : \{0,1\}^n \to \{0,1\}^n\}_{s \in \{0,1\}^k}$ denote a block cipher. For fixed ℓ, define the CBC-MAC for messages of length $(\{0,1\}^n)^\ell$ as follows (note the similarity with the CBC mode of encryption from Section 9.5): the tag of a message $m_1, \ldots, m_\ell$ with $m_i \in \{0,1\}^n$ is computed as:

```
$C_0 = 0^n$
For $i = 1$ to $\ell$:
  $C_i = F_s(m_i \oplus C_{i-1})$
Output $C_\ell$
```

Verification of a tag t on a message $m_1, \ldots, m_\ell$ is done by re-computing C_ℓ as above and outputting 1 if and only if $t = C_\ell$. It is known that the CBC-MAC is secure against adaptive chosen-message attacks [4] for n sufficiently large. We stress that this is true only when fixed-length messages are authenticated (this was why ℓ was fixed, above). Subsequent work has focused on extending CBC-MAC to allow authentication of arbitrary-length messages [8, 41].

9.7 Public-Key Encryption

The advent of public-key encryption [15, 39, 45] marked a revolution in the field of cryptography. For hundreds of years, cryptographers had relied exclusively on shared, secret keys to achieve secure communication. Public-key cryptography, however, enables two parties to secretly communicate without having arranged for any *a priori* shared information. We first describe the semantics of a public-key encryption scheme, and then discuss two general ways such a scheme can be used.

Definition 9.9 A public-key encryption scheme is a triple of PPT algorithms $(\mathcal{K}, \mathcal{E}, \mathcal{D})$ such that:

- The key generation algorithm $\mathcal{K}$ takes as input a security parameter 1^k and outputs a public key PK and a secret key SK.
- The encryption algorithm $\mathcal{E}$ takes as input a public key PK and a message m and outputs a ciphertext C. We write this as $C \leftarrow \mathcal{E}_{PK}(m)$.
- The deterministic decryption algorithm $\mathcal{D}$ takes as input a secret key SK and a ciphertext C and outputs a message m. We write this as $m = \mathcal{D}_{SK}(C)$.

We require that for all k, all (PK, SK) output by $\mathcal{K}(1^k)$, for all m, and for all C output by $\mathcal{E}_{PK}(m)$, we have $\mathcal{D}_{SK}(C) = m$.

For completeness, a message space must be specified; however, the message space is generally taken to be $\{0,1\}^*$.

There are a number of ways in which a public-key encryption scheme can be used to enable communication between a sender $\mathcal{S}$ and a receiver $\mathcal{R}$. First, we can imagine that when $\mathcal{S}$ and $\mathcal{R}$ wish to communicate, $\mathcal{R}$ executes algorithm $\mathcal{K}$ to generate the pair of keys (PK, SK). The public key PK is sent (in the clear) to $\mathcal{S}$, and the secret key SK is (of course) kept secret by $\mathcal{R}$. To send a message m, $\mathcal{S}$ computes $C \leftarrow \mathcal{E}_{PK}(m)$ and transmits C to $\mathcal{R}$. The receiver $\mathcal{R}$ can now recover the original message by computing $m = \mathcal{D}_{SK}(C)$. Note that to fully ensure secrecy against an eavesdropping adversary, it must be the case that m remains hidden even if the adversary sees both PK and C (i.e., the adversary eavesdrops on the *entire* communication between $\mathcal{S}$ and $\mathcal{R}$).

A second way to picture the situation is to imagine that $\mathcal{R}$ runs $\mathcal{K}$ to generate keys (PK, SK) *independent of any particular sender* $\mathcal{S}$ (indeed, the identity of $\mathcal{S}$ need not be known at the time the keys are generated). The public key PK of $\mathcal{R}$ is then widely distributed — for example, published on $\mathcal{R}$'s personal homepage — and may be used by *anyone* wishing to securely communicate with $\mathcal{R}$. Thus, when a sender $\mathcal{S}$ wishes to confidentially send a message m to $\mathcal{R}$, the sender simply looks up $\mathcal{R}$'s public key PK, computes $C \leftarrow \mathcal{E}_{PK}(m)$, and sends C to $\mathcal{R}$; decryption by $\mathcal{R}$ is done as before. In this way, multiple senders can communicate multiple times with $\mathcal{R}$ using the same public key PK for all communication.

Note that, as was the case above, secrecy must be guaranteed even when an adversary knows PK. This is so because, by necessity, $\mathcal{R}$'s public key is widely distributed so that anyone can communicate with $\mathcal{R}$. Thus, it is only natural to assume that the adversary also knows PK. The following definition of security extends the definition given in the case of private-key encryption.

Definition 9.10 A public-key encryption scheme $(\mathcal{K}, \mathcal{E}, \mathcal{D})$ is said to be secure against chosen-plaintext attacks if, for all messages m_1, m_2 and all PPT adversaries A, the following is negligible:

$$\left|\Pr[(PK, SK) \leftarrow \mathcal{K}(1^k) : A(PK, \mathcal{E}_{PK}(m_0)) = 1] - \Pr[(PK, SK) \leftarrow \mathcal{K}(1^k) : A(PK, \mathcal{E}_{PK}(m_1) = 1]\right|.$$

The astute reader will notice that this definition is analogous to the definition of one-time security for private-key encryption (with the exception that the adversary is now given the public key as input), but seems inherently different from the definition of security against chosen-plaintext attacks (cf. Definition 9.6). Indeed, the above definition makes no mention of any "encryption oracle" as does Definition 9.6. However, it is known for the case of public-key encryption that the definition above implies security against chosen-plaintext attacks (of course, we have seen already that the definitions are *not* equivalent in the private-key setting).

Definition 9.10 has the following immediate and important consequence, first noted by Goldwasser and Micali [29]: for a public-key encryption scheme to be secure, encryption *must* be probabilistic. To see this, note that if encryption were deterministic, an adversary could always tell whether a given ciphertext C corresponds to an encryption of m_1 or m_2 by simply computing $\mathcal{E}_{PK}(m_1)$ and $\mathcal{E}_{PK}(m_2)$ himself (recall the adversary knows PK) and comparing the results to C.

The definition of public-key encryption — in which determining the message corresponding to a ciphertext is "hard" in general, but becomes "easy" with the secret key — is reminiscent of the definition of trapdoor permutations. Indeed, the following feasibility result is known.

Theorem 9.8 ([54]) *If there exists a trapdoor permutation (generator), there exists a public-key encryption scheme secure against chosen-plaintext attacks.*

Unfortunately, public-key encryption schemes constructed via this generic result are generally quite inefficient, and it is difficult to construct *practical* encryption schemes secure in the sense of Definition 9.10. At this point, some remarks about the practical efficiency of public-key encryption are in order. Currently known public-key encryption schemes are roughly three orders of magnitude slower (per bit of plaintext) than private-key encryption schemes with comparable security. For encrypting long messages, however, all is not lost: in practice, a long message m is encrypted by first choosing at random a "short" (i.e., 128-bit) key s, encrypting this key using a public-key encryption scheme, and then encrypting m using a private-key scheme with key s. So, the public-key encryption of m under public key PK is given by:

$$\langle \mathcal{E}_{PK}(s) \circ \mathcal{E}'_s(m) \rangle,$$

where $\mathcal{E}$ is the public-key encryption algorithm and $\mathcal{E}'$ represents a private-key encryption algorithm. If both the public-key and private-key components are secure against chosen-plaintext attacks, so is the scheme above. Thus, the problem of designing efficient public-key encryption schemes for long messages is reduced to the problem of designing efficient public-key encryption for short messages.

We discuss the well-known El Gamal encryption scheme [16] here. Let G be a cyclic (multiplicative) group of order q with generator $g \in G$. Key generation consists of choosing a random $x \in \mathbb{Z}_q$ and setting $y = g^x$. The public key is (G, q, g, y) and the secret key is x. To encrypt a message $m \in G$, the sender chooses a random $r \in \mathbb{Z}_q$ and sends:

$$\langle g^r, y^r m \rangle .$$

To decrypt a ciphertext $\langle A, B \rangle$ using secret key x, the receiver computes $m = B/A^x$. It is easy to see that decryption correctly recovers the intended message.

Clearly, security of the scheme requires the discrete logarithm problem in G to be hard; if the discrete logarithm problem were easy, then the secret key x could be recovered from the information contained in the public key. Hardness of the discrete logarithm problem is not, however, sufficient for the scheme to be secure in the sense of Definition 9.10; a stronger assumption (first introduced by Diffie and Hellman [15] and hence called the *decisional Diffie-Hellman (DDH) assumption*) is, in fact, needed. (See [52] or [7] for further details.)

We have thus far *not* mentioned the "textbook RSA" encryption scheme. Here, key generation results in public key (N, e) and secret key d such that $ed = 1 \bmod \varphi(N)$ (see Section 9.3.2 for further details) and encryption of message $m \in \mathbb{Z}_N^*$ is done by computing $C = m^e \bmod N$. The reason for its omission is that this scheme is simply *not secure* in the sense of Definition 9.10; for one thing, encryption in this scheme is deterministic and therefore cannot possibly be secure.

Of course — and as discussed in Section 9.3.2 — the RSA assumption gives a trapdoor permutation generator, which in turn can be used to construct a secure encryption scheme (cf. Theorem 9.8). Such an approach, however, is inefficient and not used in practice. The public-key encryption schemes used in practice that are based on the RSA problem seem to require additional assumptions regarding certain hash functions; we refer to [5] for details that are beyond our present scope.

We close this section by noting that current, widely used encryption schemes in fact satisfy stronger definitions of security than that of Definition 9.10; in particular, encryption schemes are typically designed to be secure against chosen-ciphertext attacks (see [7] for a definition). Two efficient examples of encryption schemes meeting this stronger notion of security include the Cramer-Shoup encryption scheme [12] (based on the DDH assumption) and OAEP-RSA [6, 10, 22, 48] (based on the RSA assumption and an assumption regarding certain hash functions [5]).

9.8 Digital Signature Schemes

As public-key encryption is to private-key encryption, so are digital signature schemes to message authentication codes. Digital signature schemes are the public-key analog of MACs; they allow a *signer* who has established a public key to "sign" messages in a way that is verifiable to anyone who knows the signer's public key. Furthermore (by analogy with MACs), no adversary can forge valid-looking signatures on messages that were not explicitly authenticated (i.e., signed) by the legitimate signer.

In more detail, to use a signature scheme, a user first runs a key generation algorithm to generate a public-key/private-key pair (PK, SK); the user then publishes and widely distributes PK (as in the case of public-key encryption). When the user wants to authenticate a message m, she may do so using the signing algorithm along with her secret key SK; this results in a signature σ. Now, *anyone* who knows PK can verify correctness of the signature by running the public verification algorithm using the known public key PK, message m, and (purported) signature σ. We formalize the semantics of digital signature schemes in the following definition.

Definition 9.11 A signature scheme consists of a triple of PPT algorithms $(\mathcal{K}, \mathsf{Sign}, \mathsf{Vrfy})$ such that:

- The key generation algorithm $\mathcal{K}$ takes as input a security parameter 1^k and outputs a public key PK and a secret key SK.

- The signing algorithm Sign takes as input a secret key SK and a message m and outputs a signature $\sigma = \mathsf{Sign}_{\mathsf{SK}}(\mathsf{m})$.
- The verification algorithm Vrfy takes as input a public key PK, a message m, and a (purported) signature σ and outputs a bit signifying acceptance (1) or rejection (0).

We require that for all (PK, SK) output by $\mathcal{K}$, for all m, and for all σ output by $\mathsf{Sign}_{\mathsf{SK}}(\mathsf{m})$, we have $\mathsf{Vrfy}_{PK}(m, \sigma) = 1$.

As in the case of MACs, the message space for a signature scheme should be specified. This is also crucial when discussing the security of a scheme.

A definition of security for signature schemes is obtainable by a suitable modification of the definition of security for MACs* (cf. Definition 9.8) with oracle $\mathsf{Sign}_{\mathsf{SK}}(\cdot)$ replacing oracle $\mathcal{T}_s(\cdot)$, and the adversary now having as additional input the signer's public key. For reference, the definition (originating in [30]) is included here.

Definition 9.12 Signature scheme $(\mathcal{K}, \mathsf{Sign}, \mathsf{Vrfy})$ is said to be secure against adaptive chosen-message attacks if, for all PPT adversaries A, the following is negligible:

$$\Pr\left[(PK, SK) \leftarrow \mathcal{K}(1^k); (m, \sigma) \leftarrow A^{\mathsf{Sign}_{\mathsf{SK}}(\cdot)}(1^k, PK) : \\ \mathsf{Vrfy}_{PK}(m, \sigma) = 1 \wedge m \notin \{m_1, \ldots, m_\ell\}\right],$$

where $m_1, \ldots, m_\ell$ are the messages that A submitted to $\mathsf{Sign}_{\mathsf{SK}}(\cdot)$.

Under this definition of security, a digital signature emulates (the ideal qualities of) a handwritten signature. The definition shows that a digital signature on a message or document is easily verifiable by any recipient who knows the signer's public key; furthermore, a secure signature scheme is unforgeable in the sense that a third party cannot affix someone else's signature to a document without the signer's agreement.

Signature schemes also possess the important quality of *non-repudiation*; namely, a signer who has digitally signed a message cannot later deny doing so (of course, he can claim that his secret key was stolen or otherwise illegally obtained). Note that this property is not shared by MACs, because a tag on a given message could have been generated by either of the parties who share the secret key. Signatures, on the other hand, uniquely bind one party to the signed document.

It will be instructive to first look at a simple proposal of a signature scheme based on the RSA assumption, which is *not secure*. Unfortunately, this scheme is presented in many textbooks as a secure implementation of a signature scheme; hence, we refer to the scheme as the "textbook RSA scheme." Here, key generation involves choosing two large primes p, q of equal length and computing $N = pq$. Next, choose $e < N$ which is relatively prime to $\varphi(N)$ and compute d such that $ed = 1 \bmod \varphi(N)$. The public key is (N, e) and the secret key is (N, d). To sign a message $m \in \mathbb{Z}_N^*$, the signer computes

$$\sigma = m^d \bmod N;$$

verification of signature σ on message m is performed by checking that

$$\sigma^e \stackrel{?}{=} m \bmod N.$$

That this is indeed a signature scheme follows from the fact that $(m^d)^e = m^{de} = m \bmod N$ (see Section 9.3.2). What can we say about the security of the scheme?

*Historically, the definition of security for MACs was based on the earlier definition of security for signatures.

It is not hard to see that the textbook RSA scheme is completely insecure! An adversary can forge a valid message/signature pair as follows: choose arbitrary $\sigma \in \mathbb{Z}_N^*$ and set $m = \sigma^e \bmod N$. It is clear that the verification algorithm accepts σ as a valid signature on m.

In the previous attack, the adversary generates a signature on an essentially random message m. Here, we show how an adversary can forge a signature on a *particular* message m. First, the adversary finds arbitrary m_1, m_2 such that $m_1 m_2 = m \bmod N$; the adversary then requests and obtains signatures σ_1, σ_2 on m_1, m_2, respectively (recall that this is allowed by Definition 9.12). Now we claim that the verification algorithm accepts $\sigma = \sigma_1\sigma_2 \bmod N$ as a valid signature on m. Indeed:

$$(\sigma_1\sigma_2)^e = \sigma_1^e\sigma_2^e = m_1 m_2 = m \bmod N.$$

The two preceding examples illustrate that textbook RSA is *not* secure. The general approach, however, may be secure if the message is hashed (using a cryptographic hash function) before signing; this approach yields the *full-domain hash* (FDH) signature scheme [5]. In more detail, let $H : \{0,1\}^* \to \mathbb{Z}_N^*$ be a cryptographic hash function that might be included as part of the signer's public key. Now, message m is signed by computing $\sigma = H(m)^d \bmod N$; a signature σ on message m is verified by checking that $\sigma^e \stackrel{?}{=} H(m) \bmod N$. The presence of the hash (assuming a "good" hash function) prevents the two attacks mentioned above: for example, an adversary will still be able to generate σ, m' with $\sigma^e = m' \bmod N$ as before, but now the adversary will *not* be able to find a message m for which $H(m) = m'$. Similarly, the second attack is foiled because it is is difficult for an adversary to find m_1, m_2, m with $H(m_1)H(m_2) = H(m) \bmod N$. The use of the hash H has the additional advantage that messages of arbitrary length can now be signed.

It is, in fact, possible to prove the security of the FDH signature scheme based on the assumption that RSA is a trapdoor permutation and a (somewhat non-standard) assumption about the hash function H; however, it is beyond the scope of this work to discuss the necessary assumptions on H in order to enable a proof of security. We refer the interested reader to [5] for further details.

The Digital Signature Algorithm (DSA) (also known as the Digital Signature Standard [DSS]) [2, 20] is another widely used and standardized signature scheme whose security is related to the hardness of computing discrete logarithms (and which therefore offers an alternative to schemes whose security is based on, e.g., the RSA problem). Let p, q be primes such that $|q| = 160$ and q divides $p - 1$; typically, we might have $|p| = 512$. Let g be an element of order q in the multiplicative group $\mathbb{Z}_p^*$, and let $\langle g \rangle$ denote the subgroup of $\mathbb{Z}_p^*$ generated by g. Finally, let $H : \{0,1\}^* \to \{0,1\}^{160}$ be a cryptographic hash function. Parameters (p, q, g, H) are public, and can be shared by multiple signers. A signer's personal key is computed by choosing a random $x \in \mathbb{Z}_q$ and setting $y = g^x \bmod p$; the signer's public key is y and their private key is x. (Note that if computing discrete logarithms in $\langle g \rangle$ were easy, then it would be possible to compute a signer's secret key from their public key and the scheme would immediately be insecure.)

To sign a message $m \in \{0,1\}^*$ using secret key x, the signer generates a random $k \in \mathbb{Z}_q$ and computes

$$\begin{aligned} r &= (g^k \bmod p) \bmod q \\ s &= (H(m) + xr)k^{-1} \bmod q \end{aligned}$$

The signature is (r, s). Verification of signature (r, s) on message m with respect to public key y is done by checking that $r, s \in \mathbb{Z}_q^*$ and

$$r \stackrel{?}{=} (g^{H(m)s^{-1}} y^{rs^{-1}} \bmod p) \bmod q.$$

It can be easily verified that signatures produced by the legitimate signer are accepted (with all but negligible probability) by the verification algorithm.

It is beyond the scope of this work to discuss the security of DSA; we refer the reader to a recent survey article [53] for further discussion and details.

Finally, we state the following result, which is of great theoretical importance but (unfortunately) of limited practical value.

Theorem 9.9 ([35, 40, 46]) *If there exists a one-way function family $\mathcal{F}$, then there exists a digital signature scheme secure against adaptive chosen-message attack.*

Defining Terms

Block cipher: An efficient instantiation of a pseudorandom function.
Ciphertext: The result of encrypting a message.
Collision-resistant hash function: Hash function for which it is infeasible to find two different inputs mapping to the same output.
Data integrity: Ensuring that modifications to a communicated message are detected.
Data secrecy: Hiding the contents of a communicated message.
Decrypt: To recover the original message from the transmitted ciphertext.
Digital signature scheme: Method for protecting data integrity in the public-key setting.
Encrypt: To apply an encryption scheme to a plaintext message.
Message-authentication code: Algorithm preserving data integrity in the private-key setting.
Mode of encryption: A method for using a block cipher to encrypt arbitrary-length messages.
One-time pad: A private-key encryption scheme achieving perfect secrecy.
One-way function: A function that is "easy" to compute but "hard" to invert.
Plaintext: The communicated data, or message.
Private-key encryption: Technique for ensuring data secrecy in the private-key setting.
Private-key setting: Setting in which communicating parties secretly share keys in advance of their communication.
Pseudorandom function: A keyed function that is indistinguishable from a truly random function.
Pseudorandom generator: A deterministic function that converts a short, random string to a longer, pseudorandom string.
Public-key encryption: Technique for ensuring data secrecy in the public-key setting.
Public-key setting: Setting in which parties generate public/private keys and widely disseminate their public keys.
Trapdoor permutation: A one-way permutation that is "easy" to invert if some trapdoor information is known.

References

[1] Alexi, W.B., Chor, B., Goldreich, O., and Schnorr, C.P. 1988. RSA/Rabin functions: certain parts are as hard as the whole. *SIAM J. Computing*, 17(2):194–209.

[2] ANSI X9.30. 1997. Public key cryptography for the financial services industry. Part 1: The digital signature algorithm (DSA). American National Standards Institute. American Bankers Association.

[3] Bellare, M., Desai, A., Jokipii, E., and Rogaway, P. 1997. A concrete security treatment of symmetric encryption. *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, IEEE, pp. 394–403.

[4] Bellare, M., Kilian, J., and Rogaway, P. 2000. The security of the cipher block chaining message authentication code. *J. of Computer and System Sciences*, 61(3):362–399.

[5] Bellare, M. and Rogaway, P. 1993. Random oracles are practical: a paradigm for designing efficient protocols. *First ACM Conference on Computer and Communications Security*, ACM, pp. 62–73.

[6] Bellare, M. and Rogaway, P. 1995. Optimal asymmetric encryption. *Advances in Cryptology — Eurocrypt '94*, Lecture Notes in Computer Science, Vol. 950, A. De Santis, Ed., Springer-Verlag, pp. 92–111.

[7] Bellare, M. and Rogaway, P. January 2003. Introduction to modern cryptography. Available at http://www.cs.ucsd.edu/users/mihir/cse207/classnotes.html.

[8] Black, J. and Rogaway, P. 2000. CBC MACs for arbitrary-length messages: the three-key constructions. *Advances in Cryptology — Crypto 2000*, Lecture Notes in Computer Science, Vol. 1880, M. Bellare, Ed., Springer-Verlag, pp. 197–215.

[9] Blum, M. and Micali, S. 1984. How to generate cryptographically strong sequences of pseudorandom bits. *SIAM J. Computing*, 13(4):850–864.

[10] Boneh, D. 2001. Simplified OAEP for the RSA and Rabin functions. *Advances in Cryptology — Crypto 2001*, Lecture Notes in Computer Science, Vol. 2139, J. Kilian, Ed., Springer-Verlag, pp. 275–291.

[11] Childs, L.N. 2000. *A Concrete Introduction to Higher Algebra*. Springer-Verlag, Berlin.

[12] Cramer, R. and Shoup, V. 1998. A practical public-key cryptosystem provably secure against adaptive chosen ciphertext attack. *Advances in Cryptology — Crypto '98*, Lecture Notes in Computer Science, Vol. 1462, H. Krawczyk, Ed., Springer-Verlag, pp. 13–25.

[13] Daemen, J. and Rijmen, V. 2002. *The Design of Rijndael: AES — The Advanced Encryption Standard*. Springer-Verlag, Berlin.

[14] Delfs, H. and Knebl, H. 2002. *Introduction to Cryptography: Principles and Applications*. Springer-Verlag, Berlin.

[15] Diffie, W. and Hellman, M. 1976. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6): 644–654.

[16] El Gamal, T. 1985. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472.

[17] *Federal Information Processing Standards* publication #46. 1977. Data encryption standard. U.S. Department of Commerce/National Bureau of Standards.

[18] *Federal Information Processing Standards* publication #81. 1980. DES modes of operation. U.S. Department of Commerce/National Bureau of Standards.

[19] *Federal Information Processing Standards* Publication #180-1. 1995. Secure hash standard. U.S. Department of Commerce/National Institute of Standards and Technology.

[20] *Federal Information Processing Standards* Publication #186-2. 2000. Digital signature standard (DSS). U.S. Department of Commerce/National Institute of Standards and Technology.

[21] Fischlin, R. and Schnorr, C.P. 2000. Stronger security proofs for RSA and Rabin bits. *J. Cryptology*, 13(2):221–244.

[22] Fujisaki, E., Okamoto, T., Pointcheval, D., and Stern, J. 2001. RSA-OAEP is secure under the RSA assumption. *Advances in Cryptology — Crypto 2001*, Lecture Notes in Computer Science, Vol. 2139, J. Kilian, Ed., Springer-Verlag, pp. 260–274.

[23] Goldreich, O. 2001. *Foundations of Cryptography: Basic Tools*. Cambridge University Press.

[24] Goldreich, O. Foundations of cryptography, Vol. 2: basic applications. Available at http://www.wisdom.weizmann.ac.il/~oded/foc-vol2.html.

[25] Goldreich, O., Goldwasser, S., and Micali, S. 1986. How to construct random functions. *Journal of the ACM*, 33(4):792–807.

[26] Goldreich, O., Goldwasser, S., and Micali, S. 1985. On the cryptographic applications of random functions. *Advances in Cryptology — Crypto '84*, Lecture Notes in Computer Science, Vol. 196, G.R. Blakley and D. Chaum, Eds., Springer-Verlag, pp. 276–288.

[27] Goldreich, O. and Levin, L. 1989. Hard-core predicates for any one-way function. *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, ACM, pp. 25–32.

[28] Goldwasser, S. and Bellare, M. August, 2001. Lecture notes on cryptography. Available at http://www.cs.ucsd.edu/users/mihir/papers/gb.html.

[29] Goldwasser, S. and Micali, S. 1984. Probabilistic encryption. *J. Computer and System Sciences*, 28(2):270–299.

[30] Goldwasser, S., Micali, S., and Rivest, R. 1988. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Computing*, 17(2):281–308.

[31] Håstad, J., Impagliazzo, R., Levin, L., and Luby, M. 1999. A pseudorandom generator from any one-way function. *SIAM J. Computing*, 28(4):1364–1396.

[32] Håstad, J., Schrift, A.W., and Shamir, A. 1993. The discrete logarithm modulo a composite hides $O(n)$ bits. *J. Computer and System Sciences*, 47(3):376–404.

[33] Knuth, D.E. 1997. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms (third edition)*. Addison-Wesley Publishing Company.

[34] Koblitz, N. 1999. *Algebraic Aspects of Cryptography*. Springer-Verlag, Berlin.

[35] Lamport, L. 1979. Constructing digital signatures from any one-way function. Technical Report CSL-98, SRI International, Palo Alto.

[36] Long, D.L. and Wigderson, A. 1988. The discrete logarithm problem hides $O(\log n)$ bits. *SIAM J. Computing*, 17(2):363–372.

[37] Luby, M. and Rackoff, C. 1988. How to construct pseudorandom permutations from pseudorandom functions. *SIAM J. Computing*, 17(2):412–426.

[38] Menezes, A.J., van Oorschot, P.C., and Vanstone, S.A. 2001. *Handbook of Applied Cryptography*. CRC Press.

[39] Merkle, R. and Hellman, M. 1978. Hiding information and signatures in trapdoor knapsacks. *IEEE Transactions on Information Theory*, 24:525–530.

[40] Naor, M. and Yung, M. 1989. Universal one-way hash functions and their cryptographic applications. *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, ACM, pp. 33–43.

[41] Petrank, E. and Rackoff, C. 2000. CBC MAC for real-time data sources. *J. of Cryptology*, 13(3): 315–338.

[42] Rabin, M.O. 1979. Digitalized signatures and public key functions as intractable as factoring. MIT/LCS/TR-212, MIT Laboratory for Computer Science.

[43] Rivest, R. 1990. Cryptography. Chapter 13 of *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity*, J. van Leeuwen, Ed., MIT Press.

[44] Rivest, R. 1992. The MD5 message-digest algorithm. RFC 1321, available at ftp://ftp.rfc-editor.org/in-notes/rfc1321.txt.

[45] Rivest, R., Shamir, A., and Adleman, L.M. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.

[46] Rompel, J. 1990. One-way functions are necessary and sufficient for secure signatures. *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, ACM, pp. 387–394.

[47] Schneier, B. 1995. *Applied Cryptography: Protocols, Algorithms, and Source Code in C (second edition)*. John Wiley & Sons.

[48] Shoup, V. 2001. OAEP reconsidered. *Advances in Cryptology — Crypto 2001*, Lecture Notes in Computer Science, Vol. 2139, J. Kilian, Ed., Springer-Verlag, pp. 239–259.

[49] Simon, D. 1998. Finding collisions on a one-way street: can secure hash functions be based on general assumptions? *Advances in Cryptology — Eurocrypt '98*, Lecture Notes in Computer Science, Vol. 1403, K. Nyberg, Ed., Springer-Verlag, pp. 334–345.

[50] Sipser, M. 1996. *Introduction to the Theory of Computation*. Brooks/Cole Publishing Company.

[51] Stinson, D.R. 2002. *Cryptography: Theory and Practice (second edition)*. Chapman & Hall.

[52] Tsiounis, Y. and Yung, M. 1998. On the security of El Gamal based encryption. *Public Key Cryptography — PKC '98*, Lecture Notes in Computer Science, Vol. 1431, H. Imai and Y. Zheng, Eds., Springer-Verlag, pp. 117–134.

[53] Vaudenay, S. 2003. The security of DSA and ECDSA. *Public-Key Cryptography — PKC 2003*, Lecture Notes in Computer Science, Vol. 2567, Y. Desmedt, Ed., Springer-Verlag, pp. 309–323.

[54] Yao, A.C. 1982. Theory and application of trapdoor functions. *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, IEEE, pp. 80–91.

Further Information

A number of excellent sources are available for the reader interested in more information about modern cryptography. An excellent and enjoyable review of the field up to 1990 is given by Rivest [43]. Details on the more practical aspects of cryptography appear in the approachable textbooks of Stinson [51] and Schneier [47]; the latter also includes detail on implementing many popular cryptographic algorithms.

More formal and mathematical approaches to the subject (of which the present treatment is an example) are available in a number of well-written textbooks and online texts, including those by Goldwasser and Bellare [28], Goldreich [23, 24], Delfs and Knebl [14], and Bellare and Rogaway [7]. We also mention the comprehensive reference book by Menezes, van Oorschot, and Vanstone [38].

The International Association for Cryptologic Research (IACR) sponsors a number of conferences covering all areas of cryptography, with Crypto and Eurocrypt being perhaps the best known. Proceedings of these conferences (dating, in some cases, to the early 1980s) are published as part of Springer-Verlag's *Lecture Notes in Computer Science*. Research in theoretical cryptography often appears at the ACM Symposium on Theory of Computing, the Annual Symposium on Foundations of Computer Science (sponsored by IEEE), and elsewhere; more practice-oriented aspects of cryptography are covered in many security conferences, including the ACM Conference on Computer and Communications Security.

The IACR publishes the *Journal of Cryptology*, which is devoted exclusively to cryptography. Articles on cryptography frequently appear in the *Journal of Computer and System Sciences*, the *Journal of the ACM*, and the *SIAM Journal of Computing*.

10

Parallel Algorithms

Guy E. Blelloch
Carnegie Mellon University

Bruce M. Maggs
Carnegie Mellon University

10.1 Introduction

The subject of this chapter is the design and analysis of parallel algorithms. Most of today's computer algorithms are sequential, that is, they specify a sequence of steps in which each step consists of a single operation. As it has become more difficult to improve the performance of sequential computers, however, researchers have sought performance improvements in another place: parallelism. In contrast to a sequential algorithm, a parallel algorithm may perform multiple operations in a single step. For example, consider the problem of computing the sum of a sequence, A, of n numbers. The standard sequential algorithm computes the sum by making a single pass through the sequence, keeping a running sum of the numbers seen so far. It is not difficult, however, to devise an algorithm for computing the sum that performs many operations in parallel. For example, suppose that, in parallel, each element of A with an even index is paired and summed with the next element of A, which has an odd index, i.e., $A[0]$ is paired with $A[1]$, $A[2]$ with $A[3]$, and so on. The result is a new sequence of $\lceil n/2 \rceil$ numbers whose sum is identical to the sum that we wish to compute. This pairing and summing step can be repeated, and after $\lceil \log_2 n \rceil$ steps, only the final sum remains.

The parallelism in an algorithm can yield improved performance on many different kinds of computers. For example, on a parallel computer, the operations in a parallel algorithm can be performed simultaneously by different processors. Furthermore, even on a single-processor computer it is possible to exploit the parallelism in an algorithm by using multiple functional units, pipelined functional units, or pipelined memory systems. As these examples show, it is important to make a distinction between the parallelism in an algorithm and the ability of any particular computer to perform multiple operations in parallel. Typically, a parallel algorithm will run efficiently on a computer if the algorithm contains at least as much parallelism as the computer. Thus, good parallel algorithms generally can be expected to run efficiently on sequential computers as well as on parallel computers.

The remainder of this chapter consists of eight sections. Section 10.2 begins with a discussion of how to model parallel computers. Next, in Section 10.3 we cover some general techniques that have proven useful in the design of parallel algorithms. Section 10.4 to Section 10.8 present algorithms for solving problems from different domains. We conclude in Section 10.9 with a brief discussion of parallel complexity theory. Throughout this chapter, we assume that the reader has some familiarity with sequential algorithms and asymptotic analysis.

10.2 Modeling Parallel Computations

To analyze parallel algorithms it is necessary to have a formal model in which to account for costs. The designer of a sequential algorithm typically formulates the algorithm using an abstract model of computation called a *random-access machine* (RAM) [Aho et al. 1974, ch. 1]. In this model, the machine consists of a single processor connected to a memory system. Each basic central processing unit (CPU) operation, including arithmetic operations, logical operations, and memory accesses, requires one time step. The designer's goal is to develop an algorithm with modest time and memory requirements. The random-access machine model allows the algorithm designer to ignore many of the details of the computer on which the algorithm ultimately will be executed, but it captures enough detail that the designer can predict with reasonable accuracy how the algorithm will perform.

Modeling parallel computations is more complicated than modeling sequential computations because in practice parallel computers tend to vary more in their organizations than do sequential computers. As a consequence, a large proportion of the research on parallel algorithms has gone into the question of modeling, and many debates have raged over what the *right* model is, or about how practical various models are. Although there has been no consensus on the right model, this research has yielded a better understanding of the relationships among the models. Any discussion of parallel algorithms requires some understanding of the various models and the relationships among them.

Parallel models can be broken into two main classes: **multiprocessor models** and **work-depth models**. In this section we discuss each and then discuss how they are related.

10.2.1 Multiprocessor Models

A multiprocessor model is a generalization of the sequential RAM model in which there is more than one processor. Multiprocessor models can be classified into three basic types: local memory machines, modular memory machines, and **parallel random-access machines (PRAMs)**. Figure 10.1 illustrates the structures of these machines. A local memory machine consists of a set of n processors, each with its own local memory. These processors are attached to a common communication network. A modular memory machine consists of m memory modules and n processors all attached to a common network. A PRAM consists of a set of n processors all connected to a common shared memory [Fortune and Wyllie 1978, Goldshlager 1978, Savitch and Stimson 1979].

The three types of multiprocessors differ in the way memory can be accessed. In a local memory machine, each processor can access its own local memory directly, but it can access the memory in another processor only by sending a memory request through the network. As in the RAM model, all local operations, including local memory accesses, take unit time. The time taken to access the memory in another processor,

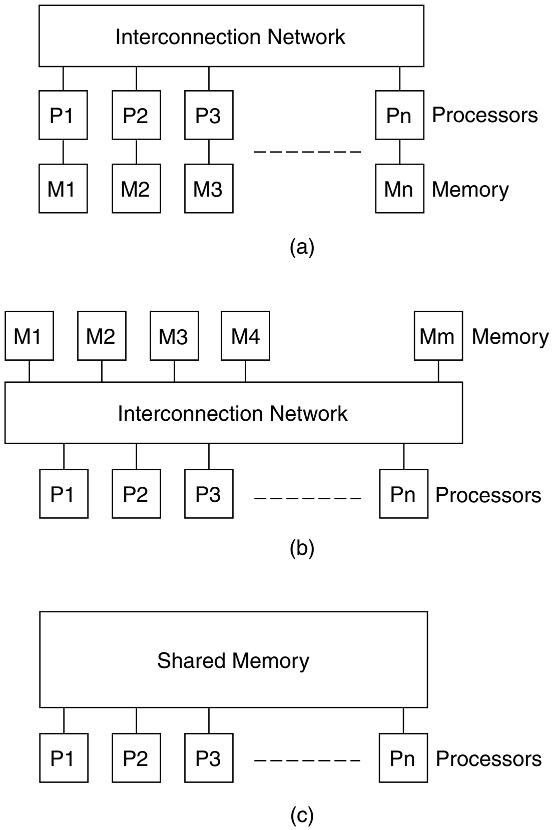


FIGURE 10.1 The three classes of multiprocessor machine models: (a) a local memory machine, (b) a modular memory machine, and (c) a parallel random-access machine (PRAM).

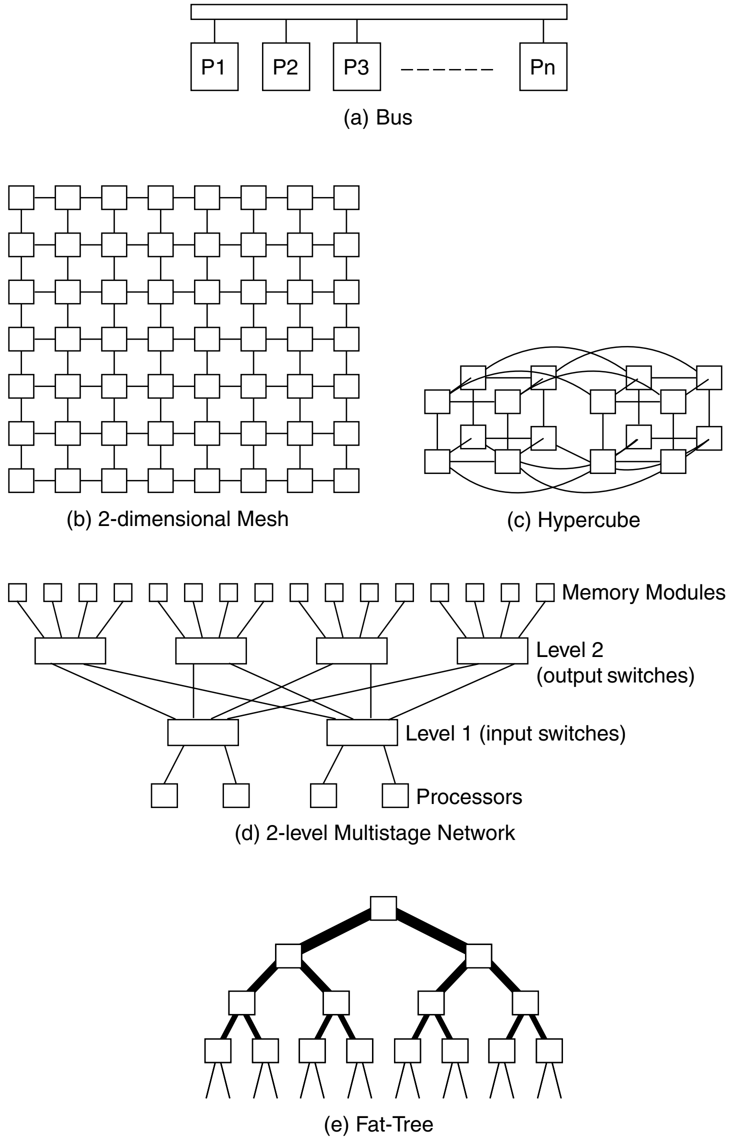
however, will depend on both the capabilities of the communication network and the pattern of memory accesses made by other processors, since these other accesses could congest the network. In a modular memory machine, a processor accesses the memory in a memory module by sending a memory request through the network. Typically, the processors and memory modules are arranged so that the time for any processor to access any memory module is roughly uniform. As in a local memory machine, the exact amount of time depends on the communication network and the memory access pattern. In a PRAM, in a single step each processor can simultaneously access any word of the memory by issuing a memory request directly to the shared memory.

The PRAM model is controversial because no real machine lives up to its ideal of unit-time access to shared memory. It is worth noting, however, that the ultimate purpose of an abstract model is not to directly model a real machine but to help the algorithm designer produce efficient algorithms. Thus, if an algorithm designed for a PRAM (or any other model) can be translated to an algorithm that runs efficiently on a real computer, then the model has succeeded. Later in this section, we show how algorithms designed for one parallel machine model can be translated so that they execute efficiently on another model.

The three types of multiprocessor models that we have defined are very broad, and these models further differ in network topology, network functionality, control, synchronization, and cache coherence. Many of these issues are discussed elsewhere in this volume. Here we will briefly discuss some of them.

10.2.1.1 Network Topology

A network is a collection of switches connected by communication channels. A processor or memory module has one or more communication ports that are connected to these switches by communication



(a) Bus

(b) 2-dimensional Mesh

(c) Hypercube

(d) 2-level Multistage Network

(e) Fat-Tree

FIGURE 10.2 Various network topologies: (a) bus, (b) two-dimensional mesh, (c) hypercube, (d) two-level multistage network, and (e) fat-tree.

channels. The pattern of interconnection of the switches is called the network topology. The topology of a network has a large influence on the performance and also on the cost and difficulty of constructing the network. Figure 10.2 illustrates several different topologies.

The simplest network topology is a bus. This network can be used in both local memory machines and modular memory machines. In either case, all processors and memory modules are typically connected to a single bus. In each step, at most one piece of data can be written onto the bus. This datum might be a request from a processor to read or write a memory value, or it might be the response from the processor or memory module that holds the value. In practice, the advantages of using buses are that they are simple to build, and, because all processors and memory modules can observe the traffic on a bus, it is relatively easy to develop protocols that allow processors to cache memory values locally.

The disadvantage of using a bus is that the processors have to take turns accessing the bus. Hence, as more processors are added to a bus, the average time to perform a memory access grows proportionately.

A two-dimensional *mesh* is a network that can be laid out in a rectangular fashion. Each switch in a mesh has a distinct label (x, y) where $0 \leq x \leq X - 1$ and $0 \leq y \leq Y - 1$. The values X and Y determine the length of the sides of the mesh. The number of switches in a mesh is thus $X \cdot Y$. Every switch, except those on the sides of the mesh, is connected to four neighbors: one to the north, one to the south, one to the east, and one to the west. Thus, a switch labeled (x, y), where $0 < x < X - 1$ and $0 < y < Y - 1$ is connected to switches $(x, y + 1)$, $(x, y - 1)$, $(x + 1, y)$, and $(x - 1, y)$. This network typically appears in a local memory machine, i.e., a processor along with its local memory is connected to each switch, and remote memory accesses are made by routing messages through the mesh. Figure 10.2b shows an example of an 8×8 mesh.

Several variations on meshes are also popular, including three-dimensional meshes, toruses, and hypercubes. A *torus* is a mesh in which the switches on the sides have connections to the switches on the opposite sides. Thus, every switch (x, y) is connected to four other switches: $(x, y + 1 \bmod Y)$, $(x, y - 1 \bmod Y)$, $(x + 1 \bmod X, y)$, and $(x - 1 \bmod X, y)$. A hypercube is a network with 2^n switches in which each switch has a distinct n-bit label. Two switches are connected by a communication channel in a hypercube if their labels differ in precisely one-bit position.

A *multistage network* is used to connect one set of switches called the *input switches* to another set called the *output switches* through a sequence of stages of switches. Such networks were originally designed for telephone networks [Beneš 1965]. The stages of a multistage network are numbered 1 through L, where L is the **depth** of the network. The input switches form stage 1 and the output switches form stage L. In most multistage networks, it is possible to send a message from any input switch to any output switch along a path that traverses the stages of the network in order from 1 to L. Multistage networks are frequently used in modular memory computers; typically, processors are attached to input switches, and memory modules to output switches. There are many different multistage network topologies. Figure 10.2d, for example, shows a 2-stage network that connects 4 processors to 16 memory modules. Each switch in this network has two channels at the bottom and four channels at the top. The ratio of processors to memory modules in this example is chosen to reflect the fact that, in practice, a processor is capable of generating memory access requests faster than a memory module is capable of servicing them.

A *fat-tree* is a network whose overall structure is that of a tree [Leiserson 1985]. Each edge of the tree, however, may represent many communication channels, and each node may represent many network switches (hence the name fat). Figure 10.2e shows a fat-tree whose overall structure is that of a binary tree. Typically the capacities of the edges near the root of the tree are much larger than the capacities near the leaves. For example, in this tree the two edges incident on the root represent 8 channels each, whereas the edges incident on the leaves represent only 1 channel each. One way to construct a local memory machine is to connect a processor along with its local memory to each leaf of the fat-tree. In this scheme, a message from one processor to another first travels up the tree to the least common ancestor of the two processors and then down the tree.

Many algorithms have been designed to run efficiently on particular network topologies such as the mesh or the hypercube. For an extensive treatment such algorithms, see Leighton [1992]. Although this approach can lead to very fine-tuned algorithms, it has some disadvantages. First, algorithms designed for one network may not perform well on other networks. Hence, in order to solve a problem on a new machine, it may be necessary to design a new algorithm from scratch. Second, algorithms that take advantage of a particular network tend to be more complicated than algorithms designed for more abstract models such as the PRAM because they must incorporate some of the details of the network. Nevertheless, there are some operations that are performed so frequently by a parallel machine that it makes sense to design a fine-tuned network-specific algorithm. For example, the algorithm that routes messages or memory access requests through the network should exploit the network topology. Other examples include algorithms for broadcasting a message from one processor to many other processors, for collecting the results computed in many processors in a single processor, and for synchronizing processors.

An alternative to modeling the topology of a network is to summarize its routing capabilities in terms of two parameters, its latency and bandwidth. The latency L of a network is the time it takes for a message to traverse the network. In actual networks this will depend on the topology of the network, which particular ports the message is passing between, and the congestion of messages in the network. The latency, however, often can be usefully modeled by considering the worst-case time assuming that the network is not heavily congested. The bandwidth at each port of the network is the rate at which a processor can inject data into the network. In actual networks this will depend on the topology of the network, the bandwidths of the network's individual communication channels, and, again, the congestion of messages in the network. The bandwidth often can be usefully modeled as the maximum rate at which processors can inject messages into the network without causing it to become heavily congested, assuming a uniform distribution of message destinations. In this case, the bandwidth can be expressed as the minimum *gap* g between successive injections of messages into the network.

Three models that characterize a network in terms of its latency and bandwidth are the postal model [Bar-Noy and Kipnis 1992], the bulk-synchronous parallel (BSP) model [Valiant 1990a], and the LogP model [Culler et al. 1993]. In the postal model, a network is described by a single parameter, L, its latency. The bulk-synchronous parallel model adds a second parameter, g, the minimum ratio of computation steps to communication steps, i.e., the gap. The LogP model includes both of these parameters and adds a third parameter, o, the overhead, or wasted time, incurred by a processor upon sending or receiving a message.

10.2.1.2 Primitive Operations

As well as specifying the general form of a machine and the network topology, we need to define what operations the machine supports. We assume that all processors can perform the same instructions as a typical processor in a sequential machine. In addition, processors may have special instructions for issuing nonlocal memory requests, for sending messages to other processors, and for executing various global operations, such as synchronization. There can also be restrictions on when processors can simultaneously issue instructions involving nonlocal operations. For example a machine might not allow two processors to write to the same memory location at the same time. The particular set of instructions that the processors can execute may have a large impact on the performance of a machine on any given algorithm. It is therefore important to understand what instructions are supported before one can design or analyze a parallel algorithm. In this section we consider three classes of nonlocal instructions: (1) how global memory requests interact, (2) synchronization, and (3) global operations on data.

When multiple processors simultaneously make a request to read or write to the same resource — such as a processor, memory module, or memory location — there are several possible outcomes. Some machine models simply forbid such operations, declaring that it is an error if more than one processor tries to access a resource simultaneously. In this case we say that the machine allows only *exclusive* access to the resource. For example, a PRAM might allow only exclusive read or write access to each memory location. A PRAM of this type is called an **exclusive-read exclusive-write (EREW)** PRAM. Other machine models may allow unlimited access to a shared resource. In this case we say that the machine allows *concurrent* access to the resource. For example, a **concurrent-read concurrent-write (CRCW)** PRAM allows both concurrent read and write access to memory locations, and a **CREW** PRAM allows **concurrent reads but only exclusive writes**. When making a concurrent write to a resource such as a memory location there are many ways to resolve the conflict. Some possibilities are to choose an arbitrary value from those written, to choose the value from the processor with the lowest index, or to take the *logical or* of the values written. A final choice is to allow for *queued* access, in which case concurrent access is permitted but the time for a step is proportional to the maximum number of accesses to any resource. A queue-read queue-write (QRQW) PRAM allows for such accesses [Gibbons et al. 1994].

In addition to reads and writes to nonlocal memory or other processors, there are other important primitives that a machine may supply. One class of such primitives supports synchronization. There are a variety of different types of synchronization operations and their costs vary from model to model. In the PRAM model, for example, it is assumed that all processors operate in lock step, which provides implicit synchronization. In a local-memory machine the cost of synchronization may be a function of the particular network topology. Some machine models supply more powerful primitives that combine arithmetic operations with communication. Such operations include the prefix and **multiprefix** operations, which are defined in the subsections on scans and multiprefix and fetch-and-add.

10.2.2 Work-Depth Models

Because there are so many different ways to organize parallel computers, and hence to model them, it is difficult to select one multiprocessor model that is appropriate for all machines. The alternative to focusing on the machine is to focus on the algorithm. In this section we present a class of models called work-depth models. In a work-depth model, the cost of an algorithm is determined by examining the total number of operations that it performs and the dependencies among those operations. An algorithm's **work** W is the total number of operations that it performs; its *depth* D is the longest chain of dependencies among its operations. We call the ratio $\mathcal{P} = W/D$ the *parallelism* of the algorithm. We say that a parallel algorithm is work-efficient relative to a sequential algorithm if it does at most a constant factor more work.

The work-depth models are more abstract than the multiprocessor models. As we shall see, however, algorithms that are efficient in the work-depth models often can be translated to algorithms that are efficient in the multiprocessor models and from there to real parallel computers. The advantage of a work-depth model is that there are no machine-dependent details to complicate the design and analysis of algorithms. Here we consider three classes of work-depth models: circuit models, vector machine models, and language-based models. We will be using a language-based model in this chapter, and so we will return to these models later in this section.

The most abstract work-depth model is the *circuit model.* In this model, an algorithm is modeled as a family of directed acyclic circuits. There is a circuit for each possible size of the input. A circuit consists of nodes and arcs. A node represents a basic operation, such as adding two values. For each input to an operation (i.e., node), there is an incoming arc from another node or from an input to the circuit. Similarly, there are one or more outgoing arcs from each node representing the result of the operation. The work of a circuit is the total number of nodes. (The work is also called the *size.*) The depth of a circuit is the length of the longest directed path between any pair of nodes. Figure 10.3 shows a circuit in which the inputs are at the top, each + is an adder circuit, and each of the arcs carries the result of an adder circuit. The final

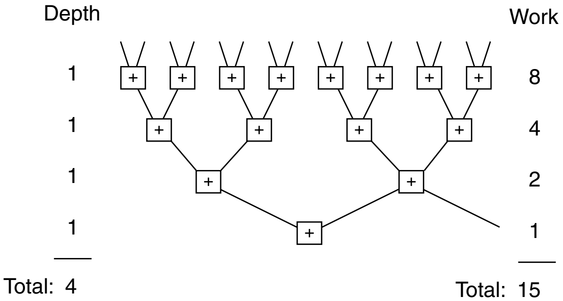


FIGURE 10.3 Summing 16 numbers on a tree. The total depth (longest chain of dependencies) is 4 and the total work (number of operations) is 15.

sum is returned at the bottom. Circuit models have been used for many years to study various theoretical aspects of parallelism, for example, to prove that certain problems are hard to solve in parallel (see Karp and Ramachandran [1990] for an overview).

In a *vector model*, an algorithm is expressed as a sequence of steps, each of which performs an operation on a vector (i.e., sequence) of input values, and produces a vector result [Pratt and Stockmeyer 1976, Blelloch 1990]. The work of each step is equal to the length of its input (or output) vector. The work of an algorithm is the sum of the work of its steps. The depth of an algorithm is the number of vector steps.

In a *language* model, a work-depth cost is associated with each programming language construct [Blelloch and Greiner 1995, Blelloch 1996]. For example, the work for calling two functions in parallel is equal to the sum of the work of the two calls. The depth, in this case, is equal to the maximum of the depth of the two calls.

10.2.3 Assigning Costs to Algorithms

In the work-depth models, the cost of an algorithm is determined by its work and by its depth. The notions of work and depth also can be defined for the multiprocessor models. The work W performed by an algorithm is equal to the number of processors times the time required for the algorithm to complete execution. The depth D is equal to the total time required to execute the algorithm.

The depth of an algorithm is important because there are some applications for which the time to perform a computation is crucial. For example, the results of a weather-forecasting program are useful only if the program completes execution before the weather does!

Generally, however, the most important measure of the cost of an algorithm is the work. This can be justified as follows. The cost of a computer is roughly proportional to the number of processors in the computer. The cost for purchasing time on a computer is proportional to the cost of the computer times the amount of time used. The total cost of performing a computation, therefore, is roughly proportional to the number of processors in the computer times the amount of time used, i.e., the work.

In many instances, the cost of running a computation on a parallel computer may be slightly larger than the cost of running the same computation on a sequential computer. If the time to completion is sufficiently improved, however, this extra cost often can be justified. As we shall see, in general there is a tradeoff between work and time to completion. It is rarely the case, however, that a user is willing to give up any more than a small constant factor in cost for an improvement in time.

10.2.4 Emulations Among Models

Although it may appear that a different algorithm must be designed for each of the many parallel models, there are often automatic and efficient techniques for translating algorithms designed for one model into algorithms designed for another. These translations are *work preserving* in the sense that the work performed by both algorithms is the same, to within a constant factor. For example, the following theorem, known as Brent's theorem [1974], shows that an algorithm designed for the circuit model can be translated in a work-preserving fashion to a PRAM algorithm.

Theorem 10.1 (Brent's theorem) *Any algorithm that can be expressed as a circuit of size (i.e., work) W and depth D in the circuit model can be executed in $O(W/P + D)$ steps in the PRAM model.*

***Proof* 10.1** The basic idea is to have the PRAM emulate the computation specified by the circuit in a level-by-level fashion. The level of a node is defined as follows. A node is on level 1 if all of its inputs are also inputs to the circuit. Inductively, the level of any other node is one greater than the maximum of the level of the nodes with arcs into it. Let l_i denote the number of nodes on level i. Then, by assigning $\lceil l_i/P \rceil$ operations to each of the P processors in the PRAM, the operations for level i can be performed in $O(\lceil l_i/P \rceil)$ steps. Summing the time over all D levels, we have

$$\begin{aligned}
T_{\text{PRAM}}(W, D, P) &= O\left(\sum_{i=1}^{D} \left\lceil \frac{l_i}{P} \right\rceil\right) \\
&= O\left(\sum_{i=1}^{D} \left(\frac{l_i}{P} + 1\right)\right) \\
&= O\left(\frac{1}{P}\left(\sum_{i=1}^{D} l_i\right) + D\right) \\
&= O\left(W/P + D\right) \qquad \square
\end{aligned}$$

The total work performed by the PRAM, i.e., the processor-time product, is $O(W + PD)$. This emulation is work preserving to within a constant factor when the parallelism ($\mathcal{P} = W/D$) is at least as large as the number of processors P, in this case the work is $O(W)$. The requirement that the parallelism exceed the number of processors is typical of work-preserving emulations.

Brent's theorem shows that an algorithm designed for one of the work-depth models can be translated in a work-preserving fashion on to a multiprocessor model. Another important class of work-preserving translations is those that translate between different multiprocessor models. The translation we consider here is the work-preserving translation of algorithms written for the PRAM model to algorithms for a more realistic machine model. In particular, we consider a *butterfly machine* in which P processors are attached through a butterfly network of depth $\log P$ to P memory banks. We assume that, in constant time, a processor can hash a virtual memory address to a physical memory bank and an address within that bank using a sufficiently powerful hash function. This scheme was first proposed by Karlin and Upfal [1988] for the EREW PRAM model. Ranade [1991] later presented a more general approach that allowed the butterfly to efficiently emulate CRCW algorithms.

Theorem 10.2 *Any algorithm that takes time T on a P-processor PRAM can be translated into an algorithm that takes time $O(T(P/P' + \log P'))$, with high probability, on a P'-processor butterfly machine.*

Sketch of proof Each of the P' processors in the butterfly machine emulates a set of P/P' PRAM processors. The butterfly machine emulates the PRAM in a step-by-step fashion. First, each butterfly processor emulates one step of each of its P/P' PRAM processors. Some of the PRAM processors may wish to perform memory accesses. For each memory access, the butterfly processor hashes the memory address to a physical memory bank and an address within the bank and then routes a message through the network to that bank. These messages are pipelined so that a processor can have multiple outstanding requests. Ranade proved that if each processor in a P-processor butterfly machine sends at most P/P' messages whose destinations are determined by a sufficiently powerful hash function, then the network can deliver all of the messages, along with responses, in $O(P/P' + \log P')$ time. The $\log P'$ term accounts for the latency of the network and for the fact that there will be some congestion at memory banks, even if each processor sends only a single message.

This theorem implies that, as long as $P \geq P' \log P'$, i.e., if the number of processors employed by the PRAM algorithm exceeds the number of processors in the butterfly machine by a factor of at least $\log P'$, then the emulation is work preserving. When translating algorithms from a guest multiprocessor model (e.g., the PRAM) to a host multiprocessor model (e.g., the butterfly machine), it is not uncommon to require that the number of guest processors exceed the number of host processors by a factor proportional to the latency of the host. Indeed, the latency of the host often can be hidden by giving it a larger guest to emulate. If the bandwidth of the host is smaller than the bandwidth of a comparably sized guest, however, it usually is much more difficult for the host to perform a work-preserving emulation of the guest.

For more information on PRAM emulations, the reader is referred to Harris [1994] and Valiant [1990].

10.2.5 Model Used in This Chapter

Because there are so many work-preserving translations between different parallel models of computation, we have the luxury of choosing the model that we feel most clearly illustrates the basic ideas behind the algorithms, a work-depth language model. Here we define the model we will use in this chapter in terms of a set of language constructs and a set of rules for assigning costs to the constructs. The description we give here is somewhat informal, but it should suffice for the purpose of this chapter. The language and costs can be properly formalized using a profiling semantics [Blelloch and Greiner 1995].

Most of the syntax that we use should be familiar to readers who have programmed in Algol-like languages, such as Pascal and C. The constructs for expressing parallelism, however, may be unfamiliar. We will be using two parallel constructs — a parallel *apply-to-each* construct and a *parallel-do* construct — and a small set of parallel primitives on sequences (one-dimensional arrays). Our language constructs, syntax, and cost rules are based on the NESL language [Blelloch 1996].

The apply-to-each construct is used to apply an expression over a sequence of values in parallel. It uses a setlike notation. For example, the expression

$$\{a * a : a \in [3, -4, -9, 5]\}$$

squares each element of the sequence $[3, -4, -9, 5]$ returning the sequence $[9, 16, 81, 25]$. This can be read: "in parallel, for each a in the sequence $[3, -4, -9, 5]$, square a." The apply-to-each construct also provides the ability to subselect elements of a sequence based on a filter. For example,

$$\{a * a : a \in [3, -4, -9, 5] \mid a > 0\}$$

can be read: "in parallel, for each a in the sequence $[3, -4, -9, 5]$ such that a is greater than 0, square a." It returns the sequence $[9, 25]$. The elements that remain maintain their relative order.

The parallel-do construct is used to evaluate multiple statements in parallel. It is expressed by listing the set of statements after an **in parallel do**. For example, the following fragment of code calls FUN1(X) and assigns the result to A and in parallel calls FUN2(Y) and assigns the result to B:

```
in parallel do
    $A$ := FUN1($X$)
    $B$ := FUN2($Y$)
```

The parallel-do completes when all the parallel subcalls complete.

Work and depth are assigned to our language constructs as follows. The work and depth of a scalar primitive operation is one. For example, the work and depth for evaluating an expression such as $3 + 4$ is one. The work for applying a function to every element in a sequence is equal to the sum of the work for each of the individual applications of the function. For example, the work for evaluating the expression

$$\{a * a : a \in [0..n)\}$$

which creates an n-element sequence consisting of the squares of 0 through $n - 1$, is n. The depth for applying a function to every element in a sequence is equal to the maximum of the depths of the individual applications of the function. Hence, the depth of the previous example is one. The work for a parallel-do construct is equal to the sum of the work for each of its statements. The depth is equal to the maximum depth of its statements. In all other cases, the work and depth for a sequence of operations is the sum of the work and depth for the individual operations.

In addition to the parallelism supplied by apply-to-each, we will use four built-in functions on sequences, *dist*, ++ (append), *flatten*, and ← (write), each of which can be implemented in parallel. The function *dist* creates a sequence of identical elements. For example, the expression *dist* (3, 5) creates the sequence

$$[3, 3, 3, 3, 3]$$

The $++$ function appends two sequences. For example, $[2, 1] ++ [5, 0, 3]$ create the sequence $[2, 1, 5, 0, 3]$. The *flatten* function converts a nested sequence (a sequence for which each element is itself a sequence) into a flat sequence. For example,

$$\mathit{flatten}([[3,5],[3,2],[1,5],[4,6]])$$

creates the sequence

$$[3,5,3,2,1,5,4,6]$$

The $\leftarrow$ function is used to write multiple elements into a sequence in parallel. It takes two arguments. The first argument is the sequence to modify and the second is a sequence of integer-value pairs that specify what to modify. For each pair (i, v), the value v is inserted into position i of the destination sequence. For example,

$$[0,0,0,0,0,0,0,0] \leftarrow [(4,-2),(2,5),(5,9)]$$

inserts the -2, 5, and 9 into the sequence at locations 4, 2, and 5, respectively, returning

$$[0,0,5,0,-2,9,0,0]$$

As in the PRAM model, the issue of concurrent writes arises if an index is repeated. Rather than choosing a single policy for resolving concurrent writes, we will explain the policy used for the individual algorithms. All of these functions have depth one and work n, where n is the size of the sequence(s) involved. In the case of the $\leftarrow$, the work is proportional to the length of the sequence of integer-value pairs, not the modified sequence, which might be much longer. In the case of $++$, the work is proportional to the length of the second sequence.

We will use a few shorthand notations for specifying sequences. The expression $[-2..1]$ specifies the same sequence as the expression $[-2, -1, 0, 1]$. Changing the left or right brackets surrounding a sequence omits the first or last elements, i.e., $[-2..1)$ denotes the sequence $[-2, -1, 0]$. The notation $A[i..j]$ denotes the subsequence consisting of elements $A[i]$ through $A[j]$. Similarly, $A[i, j)$ denotes the subsequence $A[i]$ through $A[j-1]$. We will assume that sequence indices are zero based, i.e., $A[0]$ extracts the first element of the sequence A.

Throughout this chapter, our algorithms make use of random numbers. These numbers are generated using the functions *rand_bit()*, which returns a random bit, and *rand_int(h)*, which returns a random integer in the range $[0, h-1]$.

10.3 Parallel Algorithmic Techniques

As with sequential algorithms, in parallel algorithm design there are many general techniques that can be used across a variety of problem areas. Some of these are variants of standard sequential techniques, whereas others are new to parallel algorithms. In this section we introduce some of these techniques, including parallel divide-and-conquer, randomization, and parallel pointer manipulation. In later sections on algorithms we will make use of them.

10.3.1 Divide-and-Conquer

A divide-and-conquer algorithm first splits the problem to be solved into subproblems that are easier to solve than the original problem either because they are smaller instances of the original problem, or because they are different but easier problems. Next, the algorithm solves the subproblems, possibly recursively. Typically, the subproblems can be solved independently. Finally, the algorithm merges the solutions to the subproblems to construct a solution to the original problem.

The divide-and-conquer paradigm improves program modularity and often leads to simple and efficient algorithms. It has, therefore, proven to be a powerful tool for sequential algorithm designers. Divide-and-conquer plays an even more prominent role in parallel algorithm design. Because the subproblems created in the first step are typically independent, they can be solved in parallel. Often the subproblems are solved recursively and thus the next divide step yields even more subproblems to be solved in parallel. As a consequence, even divide-and-conquer algorithms that were designed for sequential machines typically have some inherent parallelism. Note, however, that in order for divide-and-conquer to yield a highly parallel algorithm, it often is necessary to parallelize the divide step and the merge step. It is also common in parallel algorithms to divide the original problem into as many subproblems as possible, so that they all can be solved in parallel.

As an example of parallel divide-and-conquer, consider the sequential mergesort algorithm. Mergesort takes a set of n keys as input and returns the keys in sorted order. It works by splitting the keys into two sets of $n/2$ keys, recursively sorting each set, and then merging the two sorted sequences of $n/2$ keys into a sorted sequence of n keys. To analyze the sequential running time of mergesort we note that two sorted sequences of $n/2$ keys can be merged in $O(n)$ time. Hence, the running time can be specified by the recurrence

$$T(n) = \begin{cases} 2T(n/2) + O(n) & n > 1 \\ O(1) & n = 1 \end{cases}$$

which has the solution $T(n) = O(n \log n)$. Although not designed as a parallel algorithm, mergesort has some inherent parallelism since the two recursive calls can be made in parallel. This can be expressed as:

Algorithm: MERGESORT(A).

```
if ($|A| = 1$) then return $A$
else
    in parallel do
        $L$ := MERGESORT($A[0..|A|/2]$)
        $R$ := MERGESORT($A[|A|/2..|A|]$)
    return MERGE($L, R$)
```

Recall that in our work-depth model we can analyze the depth of an algorithm that makes parallel calls by taking the maximum depth of the two calls, and the work by taking the sum. We assume that the merging remains sequential so that the work and depth to merge two sorted sequences of $n/2$ keys is $O(n)$. Thus, for mergesort the work and depth are given by the recurrences:

$$\begin{aligned} W(n) &= 2W(n/2) + O(n) \\ D(n) &= \max(D(n/2), D(n/2)) + O(n) \\ &= D(n/2) + O(n) \end{aligned}$$

As expected, the solution for the work is $W(n) = O(n \log n)$, i.e., the same as the time for the sequential algorithm. For the depth, however, the solution is $D(n) = O(n)$, which is smaller than the work. Recall that we defined the parallelism of an algorithm as the ratio of the work to the depth. Hence, the parallelism of this algorithm is $O(\log n)$ (not very much). The problem here is that the merge step remains sequential, and this is the bottleneck.

As mentioned earlier, the parallelism in a divide-and-conquer algorithm often can be enhanced by parallelizing the divide step and/or the merge step. Using a parallel merge [Shiloach and Vishkin 1982], two sorted sequences of $n/2$ keys can be merged with work $O(n)$ and depth $O(\log n)$. Using this merge algorithm, the recurrence for the depth of mergesort becomes

$$D(n) = D(n/2) + O(\log n)$$

which has solution $D(n) = O(\log^2 n)$. Using a technique called **pipelined divide-and-conquer**, the depth of mergesort can be further reduced to $O(\log n)$ [Cole 1988]. The idea is to start the merge at the top level before the recursive calls complete.

Divide-and-conquer has proven to be one of the most powerful techniques for solving problems in parallel. In this chapter we will use it to solve problems from computational geometry, sorting, and performing fast Fourier transforms. Other applications range from linear systems to factoring large numbers to n-body simulations.

10.3.2 Randomization

The use of random numbers is ubiquitous in parallel algorithms. Intuitively, randomness is helpful because it allows processors to make local decisions which, with high probability, add up to good global decisions. Here we consider three uses of randomness.

10.3.2.1 Sampling

One use of randomness is to select a representative sample from a set of elements. Often, a problem can be solved by selecting a sample, solving the problem on that sample, and then using the solution for the sample to guide the solution for the original set. For example, suppose we want to sort a collection of integer keys. This can be accomplished by partitioning the keys into buckets and then sorting within each bucket. For this to work well, the buckets must represent nonoverlapping intervals of integer values and contain approximately the same number of keys. **Random sampling** is used to determine the boundaries of the intervals. First, each processor selects a random sample of its keys. Next, all of the selected keys are sorted together. Finally, these keys are used as the boundaries. Such random sampling also is used in many parallel computational geometry, graph, and string matching algorithms.

10.3.2.2 Symmetry Breaking

Another use of randomness is in **symmetry breaking**. For example, consider the problem of selecting a large independent set of vertices in a graph in parallel. (A set of vertices is *independent* if no two are neighbors.) Imagine that each vertex must decide, in parallel with all other vertices, whether to join the set or not. Hence, if one vertex chooses to join the set, then all of its neighbors must choose not to join the set. The choice is difficult to make simultaneously by each vertex if the local structure at each vertex is the same, for example, if each vertex has the same number of neighbors. As it turns out, the impasse can be resolved by using randomness to break the symmetry between the vertices [Luby 1985].

10.3.2.3 Load Balancing

A third use is load balancing. One way to quickly partition a large number of data items into a collection of approximately evenly sized subsets is to randomly assign each element to a subset. This technique works best when the average size of a subset is at least logarithmic in the size of the original set.

10.3.3 Parallel Pointer Techniques

Many of the traditional sequential techniques for manipulating lists, trees, and graphs do not translate easily into parallel techniques. For example, techniques such as traversing the elements of a linked list, visiting the nodes of a tree in postorder, or performing a depth-first traversal of a graph appear to be inherently sequential. Fortunately, these techniques often can be replaced by parallel techniques with roughly the same power.

10.3.3.1 Pointer Jumping

One of the earliest parallel pointer techniques is **pointer jumping** [Wyllie 1979]. This technique can be applied to either lists or trees. In each pointer jumping step, each node in parallel replaces its pointer with that of its successor (or parent). For example, one way to label each node of an n-node list (or tree) with the label of the last node (or root) is to use pointer jumping. After at most $\lceil \log n \rceil$ steps, every node points to the same node, the end of the list (or root of the tree). This is described in more detail in the subsection on pointer jumping.

10.3.3.2 Euler Tour

An Euler tour of a directed graph is a path through the graph in which every edge is traversed exactly once. In an undirected graph each edge is typically replaced with two oppositely directed edges. The Euler tour of an undirected tree follows the perimeter of the tree visiting each edge twice, once on the way down and once on the way up. By keeping a linked structure that represents the Euler tour of a tree, it is possible to compute many functions on the tree, such as the size of each subtree [Tarjan and Vishkin 1985]. This technique uses linear work and parallel depth that is independent of the depth of the tree. The Euler tour often can be used to replace standard traversals of a tree, such as a depth-first traversal.

10.3.3.3 Graph Contraction

Graph contraction is an operation in which a graph is reduced in size while maintaining some of its original structure. Typically, after performing a graph contraction operation, the problem is solved recursively on the contracted graph. The solution to the problem on the contracted graph is then used to form the final solution. For example, one way to partition a graph into its connected components is to first contract the graph by merging some of the vertices into their neighbors, then find the connected components of the contracted graph, and finally undo the contraction operation. Many problems can be solved by contracting trees [Miller and Reif 1989, 1991], in which case the technique is called **tree contraction**. More examples of graph contraction can be found in Section 10.5.

10.3.3.4 Ear Decomposition

An ear decomposition of a graph is a partition of its edges into an ordered collection of paths. The first path is a cycle, and the others are called ears. The endpoints of each ear are anchored on previous paths. Once an ear decomposition of a graph is found, it is not difficult to determine if two edges lie on a common cycle. This information can be used in algorithms for determining biconnectivity, triconnectivity, 4-connectivity, and planarity [Maon et al. 1986, Miller and Ramachandran 1992]. An ear decomposition can be found in parallel using linear work and logarithmic depth, independent of the structure of the graph. Hence, this technique can be used to replace the standard sequential technique for solving these problems, depth-first search.

10.3.4 Other Techniques

Many other techniques have proven to be useful in the design of parallel algorithms. Finding small graph separators is useful for partitioning data among processors to reduce communication [Reif 1993, ch. 14]. Hashing is useful for load balancing and mapping addresses to memory [Vishkin 1984, Karlin and Upfal 1988]. Iterative techniques are useful as a replacement for direct methods for solving linear systems [Bertsekas and Tsitsiklis 1989].

10.4 Basic Operations on Sequences, Lists, and Trees

We begin our presentation of parallel algorithms with a collection of algorithms for performing basic operations on sequences, lists, and trees. These operations will be used as subroutines in the algorithms that follow in later sections.

10.4.1 Sums

As explained at the opening of this chapter, there is a simple recursive algorithm for computing the sum of the elements in an array:

Algorithm: SUM(A).

```
if $|A| = 1$ then return $A[0]$
else return SUM($\{A[2i] + A[2i+1] : i \in [0..|A|/2)\}$)
```

The work and depth for this algorithm are given by the recurrences

$$W(n) = W(n/2) + O(n) = O(n)$$
$$D(n) = D(n/2) + O(1) = O(\log n)$$

which have solutions $W(n) = O(n)$ and $D(n) = O(\log n)$. This algorithm also can be expressed without recursion (using a **while** loop), but the recursive version forshadows the recursive algorithm for implementing the **scan** function.

As written, the algorithm works only on sequences that have lengths equal to powers of 2. Removing this restriction is not difficult by checking if the sequence is of odd length and separately adding the last element in if it is. This algorithm also can easily be modified to compute the sum relative to any associative operator in place of +. For example, the use of max would return the maximum value of a sequence.

10.4.2 Scans

The *plus-scan* operation (also called **all-prefix-sums**) takes a sequence of values and returns a sequence of equal length for which each element is the sum of all previous elements in the original sequence. For example, executing a plus-scan on the sequence [3, 5, 3, 1, 6] returns [0, 3, 8, 11, 12]. The scan operation can be implemented by the following algorithm [Stone 1975]:

Algorithm: SCAN(A).

```
1  if $|A| = 1$ then return $[0]$
2  else
3      $S$ = SCAN($\{A[2i] + A[2i+1] : i \in [0..|A|/2)\}$)
4      $R = \{$if $(i \bmod 2) = 0$ then $S[i/2]$ else $S[(i-1)/2] + A[i-1] : i \in [0..|A|)\}$
5  return $R$
```

The algorithm works by elementwise adding the even indexed elements of A to the odd indexed elements of A and then recursively solving the problem on the resulting sequence (line 3). The result S of the recursive call gives the plus-scan values for the even positions in the output sequence R. The value for each of the odd positions in R is simply the value for the preceding even position in R plus the value of the preceding position from A.

The asymptotic work and depth costs of this algorithm are the same as for the SUM operation, $W(n) = O(n)$ and $D(n) = O(\log n)$. Also, as with the SUM operation, any associative function can be used in place of the +. In fact, the algorithm described can be used more generally to solve various recurrences, such as the first-order linear recurrences $x_i = (x_{i-1} \otimes a_i) \oplus b_i$, $0 \leq i \leq n$, where $\otimes$ and $\oplus$ are both associative [Kogge and Stone 1973].

Scans have proven so useful in the implementation of parallel algorithms that some parallel machines provide support for scan operations in hardware.

10.4.3 Multiprefix and Fetch-and-Add

The multiprefix operation is a generalization of the scan operation in which multiple independent scans are performed. The input to the multiprefix operation is a sequence A of n pairs (k, a), where k specifies a key and a specifies an integer data value. For each key value, the multiprefix operation performs an independent scan. The output is a sequence B of n integers containing the results of each of the scans such that if $A[i] = (k, a)$ then

$$B[i] = \text{sum}(\{b : (t, b) \in A[0..i) | t = k\})$$

In other words, each position receives the sum of all previous elements that have the same key. As an example,

$$\text{MULTIPREFIX}([(1, 5), (0, 2), (0, 3), (1, 4), (0, 1), (2, 2)])$$

returns the sequence

$$[0, 0, 2, 5, 5, 0]$$

The *fetch-and-add* operation is a weaker version of the multiprefix operation, in which the order of the input elements for each scan is not necessarily the same as their order in the input sequence A. In this chapter we omit the implementation of the multiprefix operation, but it can be solved by a function that requires work $O(n)$ and depth $O(\log n)$ using concurrent writes [Matias and Vishkin 1991].

10.4.4 Pointer Jumping

Pointer jumping is a technique that can be applied to both linked lists and trees [Wyllie 1979]. The basic pointer jumping operation is simple. Each node i replaces its pointer $P[i]$ with the pointer of the node that it points to, $P[P[i]]$. By repeating this operation, it is possible to compute, for each node in a list or tree, a pointer to the end of the list or root of the tree. Given set P of pointers that represent a tree (i.e., pointers from children to their parents), the following code will generate a pointer from each node to the root of the tree. We assume that the root points to itself.

Algorithm: POINT_TO_ROOT(P).

```
for $j$ from 1 to $\lceil \log |P| \rceil$
    $P := \{P[P[i]] : i \in [0..|P|)\}$
```

The idea behind this algorithm is that in each loop iteration the distance spanned by each pointer, with respect to the original tree, will double, until it points to the root. Since a tree constructed from $n = |P|$ pointers has depth at most $n - 1$, after $\lceil \log n \rceil$ iterations each pointer will point to the root. Because each iteration has constant depth and performs $\Theta(n)$ work, the algorithm has depth $\Theta(\log n)$ and work $\Theta(n \log n)$.

10.4.5 List Ranking

The problem of computing the distance from each node to the end of a linked list is called *list ranking*. Algorithm POINT_TO_ROOT can be easily modified to compute these distances, as follows.

Algorithm: LIST_RANK (P).

```
$V = \{$if $P[i] = i$ then 0 else $1 : i \in [0..|P|)\}$
for $j$ from 1 to $\lceil \log |P| \rceil$
    $V := \{V[i] + V[P[i]] : i \in [0..|P|)\}$
    $P := \{P[P[i]] : i \in [0..|P|)\}$
return $V$
```

In this function, $V[i]$ can be thought of as the distance spanned by pointer $P[i]$ with respect to the original list. Line 1 initializes V by setting $V[i]$ to 0 if i is the last node (i.e., points to itself), and 1 otherwise. In each iteration, line 3 calculates the new length of $P[i]$. The function has depth $\Theta(\log n)$ and work $\Theta(n \log n)$.

It is worth noting that there is a simple sequential solution to the list-ranking problem that performs only $O(n)$ work: you just walk down the list, incrementing a counter at each step. The preceding parallel algorithm, which performs $\Theta(n \log n)$ work, is not **work efficient**. There are, however, a variety of work-efficient parallel solutions to this problem.

The following parallel algorithm uses the technique of random sampling to construct a pointer from each node to the end of a list of n nodes in a work-efficient fashion [Reid-Miller 1994]. The algorithm is easily generalized to solve the list-ranking problem:

1. Pick m list nodes at random and call them the *start* nodes.
2. From each start node u, follow the list until reaching the next start node v. Call the list nodes between u and v the *sublist* of u.
3. Form a shorter list consisting only of the start nodes and the final node on the list by making each start node point to the next start node on the list.
4. Using pointer jumping on the shorter list, for each start node create a pointer to the last node in the list.
5. For each start node u, distribute the pointer to the end of the list to all of the nodes in the sublist of u.

The key to analyzing the work and depth of this algorithm is to bound the length of the longest sublist. Using elementary probability theory, it is not difficult to prove that the expected length of the longest sublist is at most $O((n \log m)/m)$. The work and depth for each step of the algorithm are thus computed as follows:

1. $W(n, m) = O(m)$ and $D(n, m) = O(1)$.
2. $W(n, m) = O(n)$ and $D(n, m) = O((n \log m)/m)$.
3. $W(n, m) = O(m)$ and $D(n, m) = O(1)$.
4. $W(n, m) = O(m \log m)$ and $D(n, m) = O(\log m)$.
5. $W(n, m) = O(n)$ and $D(n, m) = O((n \log m)/m)$.

Thus, the work for the entire algorithm is $W(m, n) = O(n + m \log m)$, and the depth is $O((n \log m)/m)$. If we set $m = n/\log n$, these reduce to $W(n) = O(n)$ and $D(n) = O(\log^2 n)$.

Using a technique called **contraction**, it is possible to design a list ranking algorithm that runs in $O(n)$ work and $O(\log n)$ depth [Anderson and Miller 1988, 1990]. This technique also can be applied to trees [Miller and Reif 1989, 1991].

10.4.6 Removing Duplicates

Given a sequence of items, the remove-duplicates algorithm removes all duplicates, returning the resulting sequence. The order of the resulting sequence does not matter.

10.4.6.1 Approach 1: Using an Array of Flags

If the items are all nonnegative integers drawn from a small range, we can use a technique similar to bucket sort to remove the duplicates. We begin by creating an array equal in size to the range and initializing all of its elements to 0. Next, using concurrent writes we set a flag in the array for each number that appears in the input list. Finally, we extract those numbers whose flags are set. This algorithm is expressed as follows.

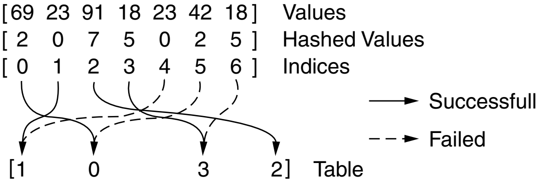

[69 23 91 18 23 42 18] Values
[2 0 7 5 0 2 5] Hashed Values
[0 1 2 3 4 5 6] Indices
Successfull
Failed
[1 0 3 2] Table

FIGURE 10.4 Each key attempts to write its index into a hash table entry.

Algorithm: REM_DUPLICATES (V).

```
RANGE $:= 1 + \text{MAX}(V)$
FLAGS $:= dist(0, \text{RANGE}) \leftarrow \{(i, 1) : i \in V\}$
**return** $\{j : j \in [0..\text{RANGE}) \mid \text{FLAGS}[j] = 1\}$
```

This algorithm has depth $O(1)$ and performs work $O(\text{MAX}(V))$. Its obvious disadvantage is that it explodes when given a large range of numbers, both in memory and in work.

10.4.6.2 Approach 2: Hashing

A more general approach is to use a hash table. The algorithm has the following outline. First, we create a hash table whose size is prime and approximately two times as large as the number of items in the set V. A prime size is best, because it makes designing a good hash function easier. The size also must be large enough that the chances of collisions in the hash table are not too great. Let m denote the size of the hash table. Next, we compute a hash value, $hash(V[j], m)$, for each item $V[j] \in V$ and attempt to write the index j into the hash table entry $hash(V[j], m)$. For example, Figure 10.4 describes a particular hash function applied to the sequence [69, 23, 91, 18, 23, 42, 18]. We assume that if multiple values are simultaneously written into the same memory location, one of the values will be correctly written. We call the values $V[j]$ whose indices j are successfully written into the hash table *winners*. In our example, the winners are $V[0]$, $V[1]$, $V[2]$, and $V[3]$, that is, 69, 23, 91, and 18. The winners are added to the duplicate-free set that we are constructing, and then set aside. Among the losers, we must distinguish between two types of items: those that were defeated by an item with the same value, and those that were defeated by an item with a different value. In our example, $V[5]$ and $V[6]$ (23 and 18) were defeated by items with the same value, and $V[4]$ (42) was defeated by an item with a different value. Items of the first type are set aside because they are duplicates. Items of the second type are retained, and we repeat the entire process on them using a different hash function. In general, it may take several iterations before all of the items have been set aside, and in each iteration we must use a different hash function.

Removing duplicates using hashing can be implemented as follows:

Algorithm: REMOVE_DUPLICATES (V).

```
$m :=$ NEXT_PRIME $(2 * |V|)$
TABLE $:= dist(-1, m)$
$i := 0$
$R := \{\}$
**while** $|V| > 0$
    TABLE := TABLE $\leftarrow \{(hash(V[j], m, i), j) : j \in [0..|V|)\}$
    $W := \{V[j] : j \in [0..|V|) \mid$ TABLE $[hash(V[j], m, i)] = j\}$
    $R := R \mathbin{++} W$
    TABLE := TABLE $\leftarrow \{(hash(k, m, i), k) : k \in W\}$
    $V := \{k \in V \mid$ TABLE $[hash(k, m, i)] \neq k\}$
    $i := i + 1$
**return** $R$
```

The first four lines of function REMOVE_DUPLICATES initialize several variables. Line 1 finds the first prime number larger than $2 * |V|$ using the built-in function NEXT_PRIME. Line 2 creates the hash table and initializes its entries with an arbitrary value (-1). Line 3 initializes i, a variable that simply counts iterations of the **while** loop. Line 4 initializes the sequence R, the result, to be empty. Ultimately, R will contain a single copy of each distinct item in the sequence V.

The bulk of the work in function REMOVE_DUPLICATES is performed by the **while** loop. Although there are items remaining to be processed, we perform the following steps. In line 6, each item $V[j]$ attempts to write its index j into the table entry given by the hash function $hash(V[j], m, i)$. Note that the hash function takes the iteration i as an argument, so that a different hash function is used in each iteration. Concurrent writes are used so that if several items attempt to write to the same entry, precisely one will win. Line 7 determines which items successfully wrote their indices in line 6 and stores their values in an array called W (for *winners*). The winners are added to the result array R in line 8. The purpose of lines 9 and 10 is to remove all of the items that are either winners or duplicates of winners. These lines reuse the hash table. In line 9, each winner writes its value, rather than its index, into the hash table. In this step there are no concurrent writes. Finally, in line 10, an item is retained only if it is not a winner, and the item that defeated it has a different value.

It is not difficult to prove that, with high probability, each iteration reduces the number of items remaining by some constant fraction until the number of items remaining is small. As a consequence, $D(n) = O(\log n)$ and $W(n) = O(n)$.

The remove-duplicates algorithm is frequently used for set operations; for instance, there is a trivial implementation of the set union operation given the code for REMOVE_DUPLICATES.

10.5 Graphs

Graphs present some of the most challenging problems to parallelize since many standard sequential graph techniques, such as depth-first or priority-first search, do not parallelize well. For some problems, such as minimum spanning tree and biconnected components, new techniques have been developed to generate efficient parallel algorithms. For other problems, such as single-source shortest paths, there are no known efficient parallel algorithms, at least not for the general case.

We have already outlined some of the parallel graph techniques in Section 10.3. In this section we describe algorithms for breadth-first search, connected components, and minimum spanning trees. These algorithms use some of the general techniques. In particular, randomization and graph contraction will play an important role in the algorithms. In this chapter we will limit ourselves to algorithms on sparse undirected graphs. We suggest the following sources for further information on parallel graph algorithms Reif [1993, Chap. 2 to 8], JáJá [1992, Chap. 5], and Gibbons and Ritter [1990, Chap. 2].

10.5.1 Graphs and Their Representation

A *graph* $G = (V, E)$ consists of a set of *vertices* V and a set of *edges* E in which each edge connects two vertices. In a *directed graph* each edge is directed from one vertex to another, whereas in an *undirected graph* each edge is symmetric, i.e., goes in both directions. A *weighted graph* is a graph in which each edge $e \in E$ has a weight $w(e)$ associated with it. In this chapter we will use the convention that $n = |V|$ and $m = |E|$. Qualitatively, a graph is considered sparse if $m \ll n^2$ and dense otherwise. The *diameter* of a graph, denoted $D(G)$, is the maximum, over all pairs of vertices (u, v), of the minimum number of edges that must be traversed to get from u to v.

There are three standard representations of graphs used in sequential algorithms: edge lists, adjacency lists, and adjacency matrices. An *edge list* consists of a list of edges, each of which is a pair of vertices. The list directly represents the set E. An *adjacency list* is an array of lists. Each array element corresponds to one vertex and contains a linked list of the neighboring vertices, i.e., the linked list for a vertex v would contain pointers to the vertices $\{u \mid (v, u) \in E\}$). An *adjacency matrix* is an $n \times n$ array A such that A_{ij} is 1 if $(i, j) \in E$ and 0 otherwise. The adjacency matrix representation is typically used only when the graph

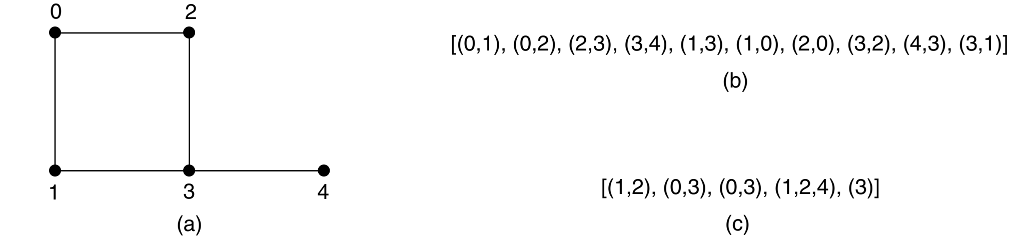


FIGURE 10.5 Representations of an undirected graph: (a) a graph, G, with 5 vertices and 5 edges, (b) the edge-list representation of G, and (c) the adjacency-list representation of G. Values between square brackets are elements of an array, and values between parentheses are elements of a pair.

is dense since it requires $\Theta(n^2)$ space, as opposed to $\Theta(m)$ space for the other two representations. Each of these representations can be used to represent either directed or undirected graphs.

For parallel algorithms we use similar representations for graphs. The main change we make is to replace the linked lists with arrays. In particular, the edge list is represented as an array of edges and the adjacency list is represented as an array of arrays. Using arrays instead of lists makes it easier to process the graph in parallel. In particular, they make it easy to grab a set of elements in parallel, rather than having to follow a list. Figure 10.5 shows an example of our representations for an undirected graph. Note that for the edge-list representation of the undirected graph each edge appears twice, once in each direction. We assume these double edges for the algorithms we describe in this chapter.* To represent a directed graph we simply store the edge only once in the desired direction. In the text we will refer to the left element of an edge pair as the *source vertex* and the right element as the *destination vertex*.

In algorithms it is sometimes more efficient to use the edge list and sometimes more efficient to use an adjacency list. It is, therefore, important to be able to convert between the two representations. To convert from an adjacency list to an edge list (representation c to representation b in Fig. 10.5) is straightforward. The following code will do it with linear work and constant depth:

$$\mathit{flatten}(\{\{(i, j) : j \in G[i]\} : i \in [0 \cdots |G|\})$$

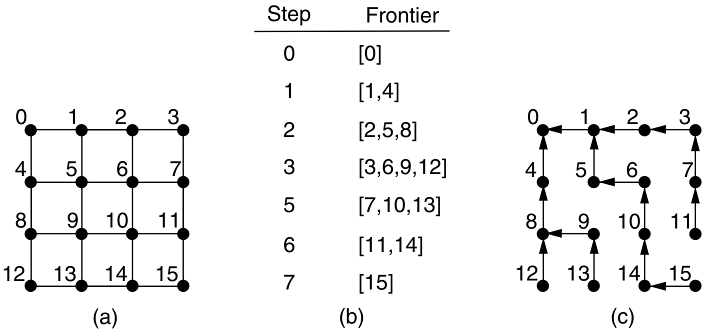
where G is the graph in the adjacency list representation. For each vertex i this code pairs up each of i's neighbors with i and then flattens the results.

To convert from an edge list to an adjacency list is somewhat more involved but still requires only linear work. The basic idea is to sort the edges based on the source vertex. This places edges from a particular vertex in consecutive positions in the resulting array. This array can then be partitioned into blocks based on the source vertices. It turns out that since the sorting is on integers in the range $[0 \ldots |V|)$, a radix sort can be used (see radix sort subsection in Section 10.6), which can be implemented in linear work. The depth of the radix sort depends on the depth of the multiprefix operation. (See previous subsection on multiprefix.)

10.5.2 Breadth-First Search

The first algorithm we consider is parallel breadth-first search (BFS). BFS can be used to solve various problems such as finding if a graph is connected or generating a spanning tree of a graph. Parallel BFS is similar to the sequential version, which starts with a source vertex s and visits levels of the graph one after the other using a queue. The main difference is that each level is going to be visited in parallel and no queue is required. As with the sequential algorithm, each vertex will be visited only once and each edge, at most twice, once in each direction. The work is therefore linear in the size of the graph $O(n + m)$. For a graph with diameter D, the number of levels processed by the algorithm will be at least $D/2$ and at most

*If space is of serious concern, the algorithms can be easily modified to work with edges stored in just one direction.



Step	Frontier
0	[0]
1	[1,4]
2	[2,5,8]
3	[3,6,9,12]
5	[7,10,13]
6	[11,14]
7	[15]

FIGURE 10.6 Example of parallel breadth-first search: (a) a graph, G, (b) the frontier at each step of the BFS of G with $s = 0$, and (c) a BFS tree.

D, depending on where the search is initiated. We will show that each level can be processed in constant depth assuming a concurrent-write model, so that the total depth of parallel BFS is $O(D)$.

The main idea of parallel BFS is to maintain a set of frontier vertices, which represent the current level being visited, and to produce a new frontier on each step. The set of frontier vertices is initialized with the singleton s (the source vertex) and during the execution of the algorithm each vertex will be visited only once. A new frontier is generated by collecting all of the neighbors of the current frontier vertices in parallel and removing any that have already been visited. This is not sufficient on its own, however, since multiple vertices might collect the same unvisited vertex. For example, consider the graph in Figure 10.6. On step 2 vertices 5 and 8 will both collect vertex 9. The vertex will therefore appear twice in the new frontier. If the duplicate vertices are not removed, the algorithm can generate an exponential number of vertices in the frontier. This problem does not occur in the sequential BFS because vertices are visited one at a time. The parallel version therefore requires an extra step to remove duplicates.

The following algorithm implements the parallel BFS. It takes as input a source vertex s and a graph G represented as an adjacency array and returns as its result a breadth-first search tree of G. In a BFS tree each vertex processed at level i points to one of its neighbors processed at level $i - 1$ [see Figure 10.6c]. The source s is the root of the tree.

Algorithm: BFS (s, G).

```
1  $Fr := [s]$
2  $Tr := dist(-1, |G|)$
3  $Tr[s] := s$
4  while $(|Fr| \neq 0)$
5      $E := flatten(\{\{(u,v) : u \in G[v]\} : v \in Fr\})$
6      $E' := \{(u,v) \in E \mid Tr[u] = -1\}$
7      $Tr := Tr \leftarrow E'$
8      $Fr := \{u : (u,v) \in E' \mid v = Tr[u]\}$
9  return $Tr$
```

In this code Fr is the set of frontier vertices, and Tr is the current BFS tree, represented as an array of indices (pointers). The pointers in Tr are all initialized to -1, except for the source s, which is initialized to point to itself. The algorithm assumes the arbitrary concurrent-write model.

We now consider each iteration of the algorithm. The iterations terminate when there are no more vertices in the frontier (line 4). The new frontier is generated by first collecting together the set of edges from the current frontier vertices to their neighbors into an edge array (line 5). An edge from v to u is represented as the pair (u, v). We then remove any edges whose destination has already been visited (line 6). Now each edge writes its source index into the destination vertex (line 7). In the case that more than one edge has the same destination, one of the source vertices will be written arbitrarily; this is the only place the algorithm will require a concurrent write. These indices will act as the back pointers for the BFS tree, and they also will be used to remove the duplicates for the next frontier set. In particular, each edge checks whether it succeeded by reading back from the destination, and if it succeeded, then the destination is included in the new frontier (line 8). Since only one edge that points to a given destination vertex will succeed, no duplicates will appear in the new frontier.

The algorithm requires only constant depth per iteration of the while loop. Since each vertex and its associated edges are visited only once, the total work is $O(m + n)$. An interesting aspect of this parallel BFS is that it can generate BFS trees that cannot be generated by a sequential BFS, even allowing for any order of visiting neighbors in the sequential BFS. We leave the generation of an example as an exercise. We note, however, that if the algorithm used a priority concurrent write (see previous subsection describing the model used in this chapter) on line 7, then it would generate the same tree as a sequential BFS.

10.5.3 Connected Components

We now consider the problem of labeling the connected components of an undirected graph. The problem is to label all of the vertices in a graph G such that two vertices u and v have the same label if and only if there is a path between the two vertices. Sequentially, the connected components of a graph can easily be labeled using either depth-first or breadth-first search. We have seen how to implement breadth-first search, but the technique requires a depth proportional to the diameter of a graph. This is fine for graphs with a small diameter, but it does not work well in the general case. Unfortunately, in terms of work, even the most efficient polylogarithmic depth parallel algorithms for depth-first search and breadth-first search are very inefficient. Hence, the efficient algorithms for solving the connected components problem use different techniques.

The two algorithms we consider are based on graph contraction. Graph contraction proceeds by contracting the vertices of a connected subgraph into a single vertex to form a new smaller graph. The techniques we use allow the algorithms to make many such contractions in parallel across the graph. The algorithms, therefore, proceed in a sequence of steps, each of which contracts a set of subgraphs, and forms a smaller graph in which each subgraph has been converted into a vertex. If each such step of the algorithm contracts the size of the graph by a constant fraction, then each component will contract down to a single vertex in $O(\log n)$ steps. By running the contraction in reverse, the algorithms can label all of the vertices in the components. The two algorithms we consider differ in how they select subgraphs for contraction. The first uses randomization and the second is deterministic. Neither algorithm is work efficient because they require $O((n + m) \log n)$ work for worst-case graphs, but we briefly discuss how they can be made to be work efficient in the subsequent improved version subsection. Both algorithms require the concurrent-write model.

10.5.3.1 Random Mate Graph Contraction

The random mate technique for graph contraction is based on forming a set of star subgraphs and contracting the stars. A *star* is a tree of depth one; it consists of a root and an arbitrary number of children. The random mate algorithm finds a set of nonoverlapping stars in a graph and then contracts each star into a single vertex by merging the children into their parents. The technique used to form the stars uses randomization. It works by having each vertex flip a coin and then identify itself as either a parent or a child based on the outcome. We assume the coin is unbiased so that every vertex has a 50% probability of being a parent. Now every vertex that has come up a child looks at its neighbors to see if any are parents. If at least one is a parent, then the child picks one of the neighboring parents as its parent. This process has selected a set of stars, which can be contracted. When contracting, we relabel all of the edges that were incident on a contracting child to its parent's label. Figure 10.7 illustrates a full contraction step. This contraction step is repeated until all components are of size 1.

To analyze the costs of the algorithm we need to know how many vertices are expected to be removed on each contraction step. First, we note that the step is going to remove only children and only if they have

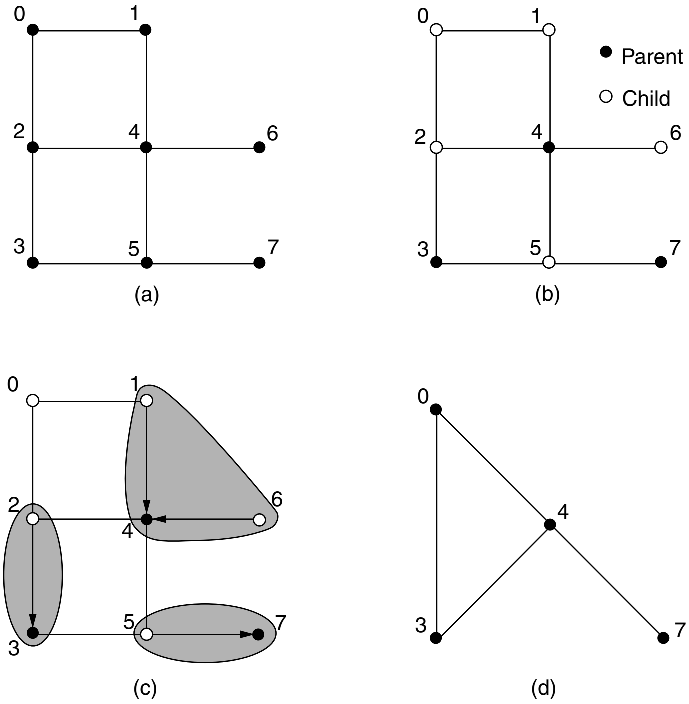


FIGURE 10.7 Example of one step of random mate graph contraction: (a) the original graph G, (b) G after selecting the parents randomly, (c) contracting the children into the parents (the shaded regions show the subgraphs), and (d) the contracted graph G'.

a neighboring parent. The probability that a vertex will be deleted is therefore the probability that it is a child multiplied by the probability that at least one of its neighbors is a parent. The probability that it is a child is 1/2 and the probability that at least one neighbor is a parent is at least 1/2 (every vertex has one or more neighbors, otherwise it would be completed). We, therefore, expect to remove at least 1/4 of the remaining vertices at each step and expect the algorithm to complete in no more than $\log_{4/3} n$ steps. The full probabilistic analysis is somewhat more involved since we could have a streak of bad flips, but it is not too hard to show that the algorithm is very unlikely to require more than $O(\log n)$ steps.

The following algorithm implements the random mate technique. The input is a graph G in the edge list representation (note that this is a different representation than used in BFS), along with the labels L of the vertices. We assume the labels are initialized to the index of the vertex. The output of the algorithm is a label for each vertex, such that all vertices in a component will be labeled with one of the original labels of a vertex in the component.

Algorithm: CC_RANDOM_MATE (L, E).

```
if $(|E| = 0)$ then return $L$
else
    CHILD := $\{rand\_bit() : v \in [1..n]\}$
    $H := \{(u, v) \in E \mid$ CHILD$[u] \wedge \neg$CHILD$[v]\}$
    $L := L \leftarrow H$
    $E' := \{(L[u], L[v]) : (u, v) \in E \mid L[u] \neq L[v]\}$
    $L' :=$ CC_RANDOM_MATE$(L, E')$
    $L' := L' \leftarrow \{(u, L'[v]) : (u, v) \in H\}$
    return $L'$
```

The algorithm works recursively by contracting the graph, labeling the components of the contracted graph, and then passing the labels to the children of the original graph. The termination condition is when there are no more edges (line 1). To make a contraction step the algorithm first flips a coin on each vertex (line 3). Now the algorithm subselects the edges-with a child on the left and a parent on the right (line 4). These are called the *hook edges*. Each of the hook edges-writes the parent index into the child's label (line 5). If a child has multiple neighboring parents, then one of the parents will be written arbitrarily; we are assuming an arbitrary concurrent write. At this point each child is labeled with one of its neighboring parents, if it has one. Now all edges update themselves to point to the parents by reading from their two endpoints and using these as their new endpoints (line 6). In the same step the edges can check if their two endpoints are within the same contracted vertex (self-edges) and remove themselves if they are. This gives a new sequence of edges E^1. The algorithm has now completed the contraction step and is called recursively on the contracted graph (line 7). The resulting labeling L' of the recursive call is used to update the labels of the children (line 8).

Two things should be noted about this algorithm. First, the algorithm flips coins on all of the vertices on each step even though many have already been contracted (there are no more edges that point to them). It turns out that this will not affect our worst-case asymptotic work or depth bounds, but in practice it is not hard to flip coins only on active vertices by keeping track of them: just keep an array of the labels of the active vertices. Second, if there are cycles in the graph, then the algorithm will create redundant edges in the contracted subgraphs. Again, keeping these edges is not a problem for the correctness or cost bounds, but they could be removed using hashing as previously discussed in the section on removing duplicates.

To analyze the full work and depth of the algorithm we note that each step requires only constant depth and $O(n+m)$ work. Since the number of steps is $O(\log n)$ with high probability, as mentioned earlier, the total depth is $O(\log n)$ and the work is $O((n+m)\log n)$, both with high probability. One might expect that the work would be linear since the algorithm reduces the number of vertices on each step by a constant fraction. We have no guarantee, however, that the number of edges also is going to contract geometrically, and in fact for certain graphs they will not. Subsequently, in this section we will discuss how this can be improved to lead to a work-efficient algorithm.

10.5.3.2 Deterministic Graph Contraction

Our second algorithm for graph contraction is deterministic [Greiner 1994]. It is based on forming trees as subgraphs and contracting these trees into a single vertex using pointer jumping. To understand the algorithm, consider the graph in Figure 10.8a. The overall goal is to contract all of the vertices of the

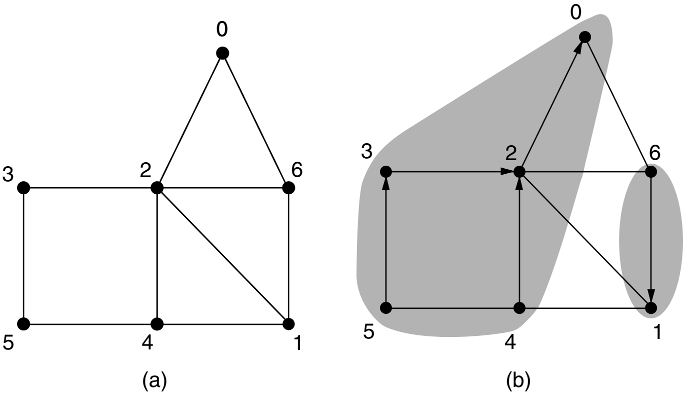


FIGURE 10.8 Tree-based graph contraction: (a) a graph, G, and (b) the hook edges induced by hooking larger to smaller vertices and the subgraphs induced by the trees.

graph into a single vertex. If we had a spanning tree that was imposed on the graph, we could contract the graph by contracting the tree using pointer jumping as discussed previously. Unfortunately, finding a spanning tree turns out to be as hard as finding the connected components of the graph. Instead, we will settle for finding a number of trees that cover the graph, contract each of these as our subgraphs using pointer jumping, and then recurse on the smaller graph. To generate the trees, the algorithm hooks each vertex into a neighbor with a smaller label. This guarantees that there are no cycles since we are only generating pointers from larger to smaller numbered vertices. This hooking will impose a set of disjoint trees on the graph. Figure 10.8b shows an example of such a hooking step. Since a vertex can have more than one neighbor with a smaller label, there can be many possible hookings for a given graph. For example, in Figure 10.8, vertex 2 could have hooked into vertex 1.

The following algorithm implements the tree-based graph contraction. We assume that the labels L are initialized to the index of the vertex.

Algorithm: CC_TREE_CONTRACT(L, E).

```
1  if$(|E| = 0)$
2  then return $L$
3  else
4      $H := \{(u, v) \in E \mid u < v\}$
5      $L := L \leftarrow H$
6      $L :=$ POINT_TO_ROOT$(L)$
7      $E' := \{(L[u], L[v]) : (u, v) \in E \mid L[u] \neq L[v]\}$
8      return CC_TREE_CONTRACT$(L, E')$
```

The structure of the algorithm is similar to the random mate graph contraction algorithm. The main differences are inhow the hooks are selected (line 4), the pointer jumping step to contract the trees (line 6), and the fact that no relabeling is required when returning from the recursive call. The hooking step simply selects edges that point from smaller numbered vertices to larger numbered vertices. This is called a *conditional hook*. The pointer jumping step uses the algorithm given earlier in Section 10.4. This labels every vertex in the tree with the root of the tree. The edge relabeling is the same as in a random mate algorithm. The reason we do not need to relabel the vertices after the recursive call is that the pointer jumping will do the relabeling.

Although the basic algorithm we have described so far works well in practice, in the worst case it can take $n - 1$ steps. Consider the graph in Figure 10.9a. After hooking and contracting, only one vertex has been removed. This could be repeated up to $n - 1$ times. This worst-case behavior can be avoided by trying to hook in both directions (from larger to smaller and from smaller to larger) and picking the hooking that hooks more vertices. We will make use of the following lemma.

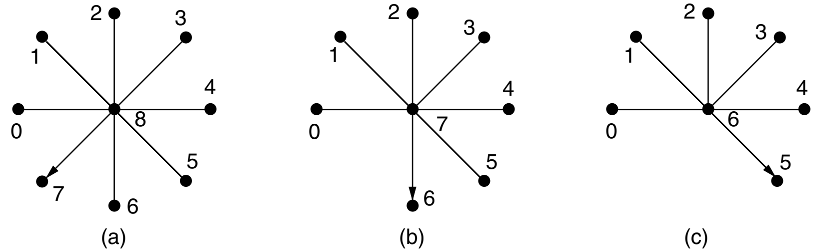


FIGURE 10.9 A worst-case graph: (a) a star graph, G, with the maximum index at the root of the star, (b) G after one step of contraction, and (c) G after two steps of contraction.

Lemma 10.1 *Let $G = (V, E)$ be an undirected graph in which each vertex has at least one neighbor, then either $|\{u|(u, v) \in E, u < v\}| \geq |V|/2$ or $|\{u|(u, v) \in E, u > v\}| > |V|/2$.*

Proof **10.2** Every vertex must have either a neighbor with a lesser index or a neighbor with a greater index. This means that if we consider the set of vertices with a lesser neighbor and the set of vertices with a greater neighbor, then one of those sets must consist of at least one-half the vertices. □

This lemma will guarantee that if we try hooking in both directions and pick the better one we will remove at least one-half of the vertices on each step, so that the number of steps is bounded by $\log n$.

We now consider the total cost of the algorithm. The hooking and relabeling of edges on each step takes $O(m)$ work and constant depth. The tree contraction using pointer jumping on each step requires $O(n \log n)$ work and $O(\log n)$ depth, in the worst case. Since there are $O(\log n)$ steps, in the worst case, the total work is $O((m + n \log n) \log n)$ and depth $O(\log^2 n)$. However, if we keep track of the active vertices (the roots) and only pointer jump on active vertices, then the work is reduced to $O((m + n) \log n)$ since the number of vertices geometrically decreases. This requires that the algorithm relabels on the way back up the recursion as done for the random mate algorithm. The total work with this modification is the same work as the randomized technique, although the depth has increased.

10.5.3.3 Improved Versions of Connected Components

There are many improvements to the two basic connected component algorithms we described. Here we mention some of them.

The deterministic algorithm can be improved to run in $O(\log n)$ depth with the same work bounds [Awerbuch and Shiloach 1987, Shiloach and Vishkin 1982]. The basic idea is to interleave the hooking steps with the **shortcutting** steps. The one tricky aspect is that we must always hook in the same direction (i.e., from smaller to larger), so as not to create cycles. Our previous technique to solve the star-graph problem, therefore, does not work. Instead, each vertex checks if it belongs to any tree after hooking. If it does not, then it can hook to any neighbor, even if it has a larger index. This is called an *unconditional hook*.

The randomized algorithm can be improved to run in optimal work $O(n + m)$ [Gazit 1991]. The basic idea is to not use all of the edges for hooking on each step and instead use a sample of the edges. This basic technique developed for parallel algorithms has since been used to improve some sequential algorithms, such as deriving the first linear work algorithm for minimum spanning trees [Klein and Tarjan 1994].

Another improvement is to use the EREW model instead of requiring concurrent reads and writes [Halperin and Zwick 1994]. However, this comes at the cost of greatly complicating the algorithm. The basic idea is to keep circular linked lists of the neighbors of each vertex and then to splice these lists when merging vertices.

10.5.3.4 Extensions to Spanning Trees and Minimum Spanning Trees

The connected component algorithms can be extended to finding a spanning tree of a graph or minimum spanning tree of a weighted graph. In both cases we assume the graphs are undirected.

A *spanning tree* of a connected graph $G = (V, E)$ is a connected graph $T = (V, E')$ such that $E' \subseteq E$ and $|E'| = |V| - 1$. Because of the bound on the number of edges, the graph T cannot have any cycles and therefore forms a tree. Any given graph can have many different spanning trees.

It is not hard to extend the connectivity algorithms to return the spanning tree. In particular, whenever two components are hooked together the algorithm can keep track of which edges were used for hooking. Since each edge will hook together two components that are not connected yet, and only one edge will succeed in hooking the components, the collection of these edges across all steps will form a spanning tree (they will connect all vertices and have no cycles). To determine which edges were used for contraction, each edge checks if it successfully hooked after the attempted hook.

A minimum spanning tree of a connected weighted graph $G = (V, E)$ with weights $w(e)$ for $e \in E$ is a spanning tree $T = (V, E')$ of G such that

$$w(T) = \sum_{e \in E'} w(e)$$

is minimized. The connected component algorithms also can be extended to determine the minimum spanning tree. Here we will briefly consider an extension of the random mate technique. The algorithm will take advantage of the property that, given any $W \subset V$, the minimum edge from W to $V - W$ must be in some minimum spanning tree. This implies that the minimum edge incident on a vertex will be on a minimum spanning tree. This will be true even after we contract subgraphs into vertices since each subgraph is a subset of V.

To implement the minimum spanning tree algorithm we therefore modify the random mate technique so that each child u, instead of picking an arbitrary parent to hook into, finds the incident edge (u, v) with minimum weight and hooks into v if it is a parent. If v is not a parent, then the child u does nothing (it is left as an orphan). Figure 10.10 illustrates the algorithm. As with the spanning tree algorithm, we keep track of the edges we use for hooks and add them to a set E'. This new rule will still remove 1/4 of the vertices on each step on average since a vertex has 1/2 probability of being a child, and there is 1/2 probability that the vertex at the other end of the minimum edge is a parent. The one complication in this minimum spanning tree algorithm is finding for each child the incident edge with minimum weight. Since we are keeping an edge list, this is not trivial to compute. If we had an adjacency list, then it would be easy, but since we are updating the endpoints of the edges, it is not easy to maintain the adjacency list. One way to solve this problem is to use a priority concurrent write. In such a write, if multiple values are written to the same location, the one coming from the leftmost position will be written. With such a scheme the minimum edge can be found by presorting the edges by their weight so that the lowest weighted edge will always win when executing a concurrent write. Assuming a priority write, this minimum spanning tree algorithm has the same work and depth as the random mate connected components algorithm.

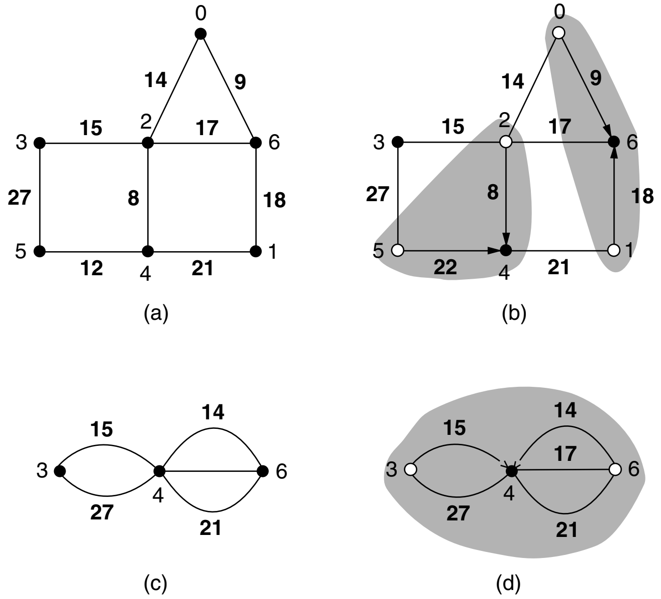


FIGURE 10.10 Example of the minimum spanning tree algorithm. (a) The original weighted graph G. (b) Each child (light) hooks across its minimum weighted edge to a parent (dark), if the edge is incident on a parent. (c) The graph after one step of contraction. (d) The second step in which children hook across minimum weighted edges to parents.

10.6 Sorting

Sorting is a problem that admits a variety of parallel solutions. In this section we limit our discussion to two parallel sorting algorithms, QuickSort and radix sort. Both of these algorithms are easy to program, and both work well in practice. Many more sorting algorithms can be found in the literature. The interested reader is referred to Akl [1985], JáJá [1992], and Leighton [1992] for more complete coverage.

10.6.1 QuickSort

We begin our discussion of sorting with a parallel version of QuickSort. This algorithm is one of the simplest to code.

Algorithm: QUICKSORT(A).

```
if $|A| = 1$ then return $A$
$i := rand\_int(|A|)$
$p := A[i]$
in parallel do
    $L :=$ QUICKSORT($\{a : a \in A \mid a < p\}$)
    $E := \{a : a \in A \mid a = p\}$
    $G :=$ QUICKSORT($\{a : a \in A \mid a > p\}$)
return $L$ ++ $E$ ++ $G$
```

We can make an optimistic estimate of the work and depth of this algorithm by assuming that each time a partition element, p, is selected, it divides the set A so that neither L nor H has more than half of the elements. In this case, the work and depth are given by the recurrences

$$W(n) = 2W(n/2) + O(n)$$
$$D(n) = D(n/2) + 1$$

whose solutions are $W(n) = O(n \log n)$ and $D(n) = O(\log n)$. A more sophisticated analysis [Knuth 1973] shows that the expected work and depth are indeed $W(n) = O(n \log n)$ and $D(n) = O(\log n)$, independent of the values in the input sequence A.

In practice, the performance of parallel QuickSort can be improved by selecting more than one partition element. In particular, on a machine with P processors, choosing $P - 1$ partition elements divides the keys into P sets, each of which can be sorted by a different processor using a fast sequential sorting algorithm. Since the algorithm does not finish until the last processor finishes, it is important to assign approximately the same number of keys to each processor. Simply choosing $p - 1$ partition elements at random is unlikely to yield a good partition. The partition can be improved, however, by choosing a larger number, *sp*, of candidate partition elements at random, sorting the candidates (perhaps using some other sorting algorithm), and then choosing the candidates with ranks $s, 2s, \ldots, (p - 1)s$ to be the partition elements. The ratio s of candidates to partition elements is called the *oversampling ratio*. As s increases, the quality of the partition increases, but so does the time to sort the *sp* candidates. Hence, there is an optimum value of s, typically larger than one, which minimizes the total time. The sorting algorithm that selects partition elements in this fashion is called *sample sort* [Blelloch et al. 1991, Huang and Chow 1983, Reif and Valiant 1983].

10.6.2 Radix Sort

Our next sorting algorithm is radix sort, an algorithm that performs well in practice. Unlike QuickSort, radix sort is not a *comparison sort*, meaning that it does not compare keys directly in order to determine the relative ordering of keys. Instead, it relies on the representation of keys as b-bit integers.

The basic radix sort algorithm (whether serial or parallel) examines the keys to be sorted one *digit* at a time, starting with the least significant digit in each key. Of fundamental importance is that this intermediate sort on digits be *stable*: the output ordering must preserve the input order of any two keys whose bits are the same.

The most common implementation of the intermediate sort is as a counting sort. A counting sort first counts to determine the *rank* of each key — its position in the output order — and then we permute the keys to their respective locations. The following algorithm implements radix sort assuming one-bit digits.

Algorithm: RADIX_SORT(A, b)

```
1  for $i$ from 0 to $b - 1$
2      $B := \{(a \gg i) \; mod \, 2 \; : \; a \in A\}$
3      $NB := \{1 - b \; : \; b \in B\}$
4      $R_0 :=$ SCAN$(NB)$
5      $s_0 :=$ SUM$(NB)$
6      $R_1 :=$ SCAN$(B)$
7      $R := \{$if $B[j] = 0$ then $R_0[j]$ else $R_1[j] + s_0 : j \in [0..|A|)\}$
8      $A := A \leftarrow \{(R[j], A[j]) : j \in [0..|A|)\}$
9  return $A$
```

For keys with b bits, the algorithm consists of b sequential iterations of a **for** loop, each iteration sorting according to one of the bits. Lines 2 and 3 compute the value and inverse value of the bit in the current position for each key. The notation $a \gg i$ denotes the operation of shifting a i bit positions to the right. Line 4 computes the rank of each key whose bit value is 0. Computing the ranks of the keys with bit value 1 is a little more complicated, since these keys follow the keys with bit value 0. Line 5 computes the number of keys with bit value 0, which serves as the rank of the first key whose bit value is 1. Line 6 computes the relative order of the keys with bit value 1. Line 7 merges the ranks of the even keys with those of the odd keys. Finally, line 8 permutes the keys according to their ranks.

The work and depth of RADIX_SORT are computed as follows. There are b iterations of the **for** loop. In each iteration, the depths of lines 2, 3, 7, 8, and 9 are constant, and the depths of lines 4, 5, and 6 are $O(\log n)$. Hence, the depth of the algorithm is $O(b \log n)$. The work performed by each of lines 2–9 is $O(n)$. Hence, the work of the algorithm is $O(bn)$.

The radix sort algorithm can be generalized so that each b-bit key is viewed as b/r blocks of r bits each, rather than as b individual bits. In the generalized algorithm, there are b/r iterations of the **for** loop, each of which invokes the SCAN function 2^r times. When r is large, a multiprefix operation can be used for generating the ranks instead of executing a SCAN for each possible value [Blelloch et al. 1991]. In this case, and assuming the multiprefix runs in linear work, it is not hard to show that as long as $b = O(\log n)$, the total work for the radix sort is $O(n)$, and the depth is the same order as the depth of the multiprefix.

Floating-point numbers also can be sorted using radix sort. With a few simple bit manipulations, floating-point keys can be converted to integer keys with the same ordering and key size. For example, IEEE double-precision floating-point numbers can be sorted by inverting the mantissa and exponent bits if the sign bit is 1 and then inverting the sign bit. The keys are then sorted as if they were integers.

10.7 Computational Geometry

Problems in computational geometry involve determining various properties about sets of objects in a k-dimensional space. Some standard problems include finding the closest distance between a pair of points (closest pair), finding the smallest convex region that encloses a set of points (convex hull), and finding line or polygon intersections. Efficient parallel algorithms have been developed for most standard problems in computational geometry. Many of the sequential algorithms are based on divide-and-conquer and lead in a relatively straightforward manner to efficient parallel algorithms. Some others are based on a technique called plane sweeping, which does not parallelize well, but for which an analogous parallel technique, the *plane sweep tree* has been developed [Aggarwal et al. 1988, Atallah et al. 1989]. In this section we describe parallel algorithms for two problems in two dimensions — closest pair and convex hull. For the convex hull we describe two algorithms. These algorithms are good examples of how sequential algorithms can be parallelized in a straightforward manner.

We suggest the following sources for further information on parallel algorithms for computational geometry: Reif [1993, Chap. 9 and Chap. 11], JáJá [1992, Chap. 6], and Goodrich [1996].

10.7.1 Closest Pair

The *closest pair problem* takes a set of points in k dimensions and returns the two points that are closest to each other. The distance is usually defined as Euclidean distance. Here we describe a closest pair algorithm for two-dimensional space, also called the planar closest pair problem. The algorithm is a parallel version of a standard sequential algorithm [Bentley and Shamos 1976], and, for n points, it requires the same work as the sequential versions $O(n \log n)$ and has depth $O(\log^2 n)$. The work is optimal.

The algorithm uses divide-and-conquer based on splitting the points along lines parallel to the y axis and is implemented as follows.

Algorithm: CLOSEST_PAIR(P).

```
if ($|P| < 2$) then return ($P, \infty$)
$x_m$ := MEDIAN ($\{x : (x, y) \in P\}$)
$L := \{(x, y) \in P \mid x < x_m\}$
$R := \{(x, y) \in P \mid x \geq x_m\}$
in parallel do
    $(L', \delta_L)$ := CLOSEST_PAIR($L$)
    $(R', \delta_R)$ := CLOSEST_PAIR($R$)
$P'$ := MERGE_BY_Y($L', R'$)
$\delta_P$ := BOUNDARY_MERGE($P', \delta_L, \delta_R, x_m$)
return ($P', \delta_P$)
```

This function takes a set of points P in the plane and returns both the original points sorted along the y axis and the distance between the closest two points. The sorted points are needed to help merge the results from recursive calls and can be thrown away at the end. It would be easy to modify the routine to return the closest pair of points in addition to the distance between them. The function works by dividing the points in half based on the median x value, recursively solving the problem on each half, and then merging the results. The MERGE_BY_Y function merges L' and R' along the y axis and can use a standard parallel merge routine. The interesting aspect of the code is the BOUNDARY_MERGE routine, which works on the same principle as described by Bentley and Shamos [1976] and can be computed with $O(\log n)$ depth and $O(n)$ work. We first review the principle and then show how it is implemented in parallel.

The inputs to BOUNDARY_MERGE are the original points P sorted along the y axis, the closest distance within L and R, and the median point x_m. The closest distance in P must be either the distance δ_L, the distance δ_R, or the distance between a point in L and a point in R. For this distance to be less than δ_L or δ_R, the two points must lie within $\delta = \min(\delta_L, \delta_R)$ of the line $x = x_m$. Thus, the two vertical lines at $x_r = x_m + \delta$ and $x_l = x_m - \delta$ define the borders of a region M in which the points must lie (see Figure 10.11). If we could find the closest distance in M, call it δ_M, then the closest overall distance is $\delta_P = \min(\delta_L, \delta_R, \delta_M)$.

To find δ_M, we take advantage of the fact that not many points can be packed closely together within M since all points within L or R must be separated by at least δ. Figure 10.11 shows the tightest possible packing of points in a $2\delta \times \delta$ rectangle within M. This packing implies that if the points in M are sorted along the y axis, each point can determine the minimum distance to another point in M by looking at a fixed number of neighbors in the sorted order, at most seven in each direction. To see this, consider one of the points along the top of the $2\delta \times \delta$ rectangle. To find if there are any points below it that are closer

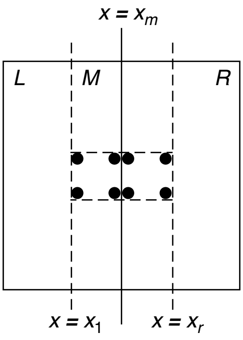


FIGURE 10.11 Merging two rectangles to determine the closest pair. Only 8 points can fit in the $2\delta \times \delta$ dashed rectangle.

than δ, it needs only to consider the points within the rectangle (points below the rectangle must be farther than δ away). As the figure illustrates, there can be at most seven other points within the rectangle. Given this property, the following function implements the border merge.

Algorithm: BOUNDARY_MERGE($P, \delta_L, \delta_R, x_m$).

```
$\delta := \min(\delta_L, \delta_R)$
$M := \{(x, y) \in P \mid (x \geq x_m - \delta) \wedge (x \leq x_m + \delta)\}$
$\delta_M := \min(\{= \min(\{distance(M[i], M[i + j]) : j \in [1..7]\})$
              $: i \in [0..|P - 7)\}$
return $\min(\delta, \delta_M)$
```

In this function each point in M looks at seven points in front of it in the sorted order and determines the distance to each of these points. The minimum over all distances is taken. Since the distance relationship is symmetric, there is no need for each point to consider points behind it in the sorted order.

The work of BOUNDARY_MERGE is $O(n)$ and the depth is dominated by taking the minimum, which has $O(\log n)$ depth.* The work of the merge and median steps in CLOSEST_PAIR is also $O(n)$, and the depth of both is bounded by $O(\log n)$. The total work and depth of the algorithm therefore can be solved with the recurrences

$$\begin{aligned} W(n) &= 2W(n/2) + O(n) &&= O(n \log n) \\ D(n) &= D(n/2) + O(\log n) &&= O(\log^2 n) \end{aligned}$$

10.7.2 Planar Convex Hull

The convex hull problem takes a set of points in k dimensions and returns the smallest convex region that contains all of the points. In two dimensions, the problem is called the planar convex hull problem and it returns the set of points that form the corners of the region. These points are a subset of the original points. We will describe two parallel algorithms for the planar convex hull problem. They are both based on divide-and-conquer, but one does most of the work before the divide step, and the other does most of the work after.

*The depth of finding the minimum or maximum of a set of numbers actually can be improved to $O(\log \log n)$ with concurrent reads [Shiloach and Vishkin 1981].

10.7.2.1 QuickHull

The parallel *QuickHull* algorithm [Blelloch and Little 1994] is based on the sequential version [Preparata and Shamos 1985], so named because of its similarity to the QuickSort algorithm. As with QuickSort, the strategy is to pick a *pivot* element, split the data based on the pivot, and recurse on each of the split sets. Also as with QuickSort, the pivot element is not guaranteed to split the data into equally sized sets, and in the worst case the algorithm requires $O(n^2)$ work; however, in practice the algorithm is often very efficient, probably the most practical of the convex hull algorithms. At the end of the section we briefly describe how the splits can be made precisely so the work is bounded by $O(n \log n)$.

The QuickHull algorithm is based on the recursive function SUBHULL, which is implemented as follows.

Algorithm: SUBHULL(P, p_1, p_2).

```
 1  $P' := \{p \in P \mid \text{RIGHT\_OF ?}(p, (p_1, p_2))\}$
 2  if $(|P'| < 2)$
 3  then return $[p_1]$ ++ $P'$
 4  else
 5      $i :=$ MAX_INDEX($\{\text{DISTANCE}(p, (p_1, p_2)) : p \in P'\}$)
 6      $p_m := P'[i]$
 7      in parallel do
 8          $H_l :=$ SUBHULL($P', p_1, p_m$)
 9          $H_r :=$ SUBHULL($P', p_m, p_2$)
10      return $H_l$ ++ $H_r$
```

This function takes a set of points P in the plane and two points p_1 and p_2 that are known to lie on the convex hull and returns all of the points that lie on the hull clockwise from p_1 to p_2, inclusive of p_1, but not of p_2. For example, in Figure 10.12 SUBHULL$([A, B, C, \ldots, P], A, P)$ would return the sequence $[A, B, J, O]$.

The function SUBHULL works as follows. Line 1 removes all of the elements that cannot be on the hull because they lie to the right of the line from p_1 to p_2. This can easily be calculated using a cross product. If the remaining set P' is either empty or has just one element, the algorithm is done. Otherwise, the algorithm finds the point p_m farthest from the line (p_1, p_2). The point p_m must be on the hull since as a line at infinity parallel to (p_1, p_2) moves toward (p_1, p_2), it must first hit p_m. In line 5, the function MAX_INDEX returns the index of the maximum value of a sequence, using $O(n)$ work $O(\log n)$ depth, which is then used to extract the point p_m. Once p_m is found, SUBHULL is called twice recursively to find

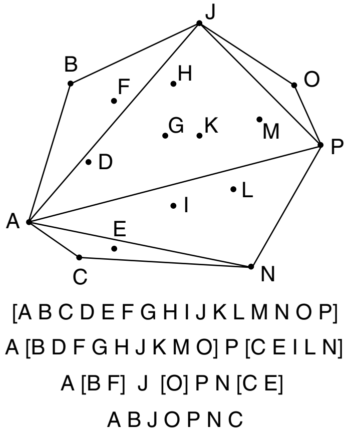

J
B
F
H
O
G
K
M
P
D
L
I
A
E
C
N
[A B C D E F G H I J K L M N O P]
A [B D F G H J K M O] P [C E I L N]
A [B F] J [O] P N [C E]
A B J O P N C

FIGURE 10.12 An example of the QuickHull algorithm.

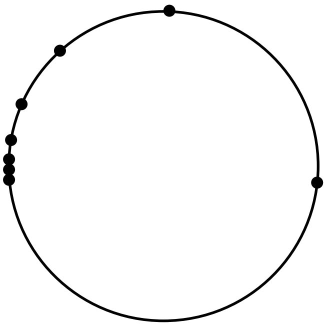


FIGURE 10.13 Contrived set of points for worst-case QuickHull.

the hulls from p_1 to p_m and from p_m to p_2. When the recursive calls return, the results are appended. The algorithm function uses SUBHULL to find the full convex hull.

Algorithm: QUICK_HULL(P).

```
$X := \{x : (x, y) \in P\}$
$x_{\min} := P[min\_index(X)]$
$x_{\max} := P[max\_index(X)]$
**return** SUBHULL($P, x_{\min}, x_{\max}$) ++  SUBHULL($P, x_{\max}, x_{\min}$)
```

We now consider the costs of the parallel QuickHull. The cost of everything other than the recursive calls is $O(n)$ work and $O(\log n)$ depth. If the recursive calls are balanced so that neither recursive call gets much more than half the data, then the number of levels of recursion will be $O(\log n)$. This will lead to the algorithm running in $O(\log^2 n)$ depth. Since the sum of the sizes of the recursive calls can be less than n (e.g., the points within the triangle AJP will be thrown out when making the recursive calls to find the hulls between A and J and between J and P), the work can be as little as $O(n)$ and often is in practice. As with QuickSort, however, when the recursive calls are badly partitioned, the number of levels of recursion can be as bad as $O(n)$ with work $O(n^2)$. For example, consider the case when all of the points lie on a circle and have the following unlikely distribution: $x_{\min}$ and $x_{\max}$ appear on opposite sides of the circle. There is one point that appears halfway between $x_{\min}$ and $x_{\max}$ on the sphere and this point becomes the new $x_{\max}$. The remaining points are defined recursively. That is, the points become arbitrarily close to $x_{\min}$ (see Figure 10.13). Kirkpatrick and Seidel [1986] have shown that it is possible to modify QuickHull so that it makes provably good partitions. Although the technique is shown for a sequential algorithm, it is easy to parallelize. A simplification of the technique is given by Chan et al. [1995]. This parallelizes even better and leads to an $O(\log^2 n)$ depth algorithm with $O(n \log h)$ work where h is the number of points on the convex hull.

10.7.2.2 MergeHull

The MergeHull algorithm [Overmars and Van Leeuwen 1981] is another divide-and-conquer algorithm for solving the planar convex hull problem. Unlike QuickHull, however, it does most of its work after returning from the recursive calls. The algorithm is implemented as follows.

Algorithm: MERGEHULL(P).

```
**if** ($|P| < 3$) **then return** $P$
**else**
    **in parallel do**
        $H_1$ = MERGEHULL ($P[0..|P|/2)$)
        $H_2$ = MERGEHULL ($P[|P|/2..|P|)$)
    **return** JOIN\_HULLS($H_1, H_2$)
```

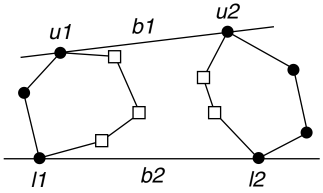


FIGURE 10.14 Merging two convex hulls.

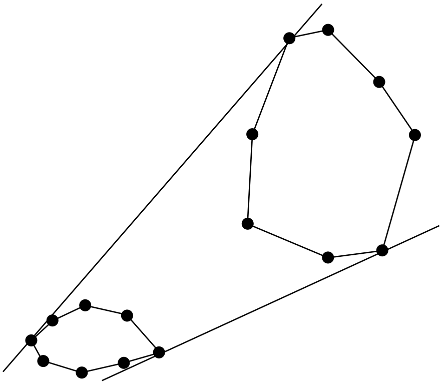


FIGURE 10.15 A bridge that is far from the top of the convex hull.

This function assumes the input P is presorted according to the x coordinates of the points. Since the points are presorted, H_1 is a convex hull on the left and H_2 is a convex hull on the right. The JOIN_HULLS routine is the interesting part of the algorithm. It takes the two hulls and merges them into one. To do this, it needs to find upper and lower points u_1 and l_1 on H_1 and u_2 and l_2 on H_2 such that u_1, u_2 and l_1, l_2 are successive points on H (see Figure 10.14). The lines b_1 and b_2 joining these upper and lower points are called the upper and lower bridges, respectively. All of the points between u_1 and l_1 and between u_2 and l_2 on the *outer* sides of H_1 and H_2 are on the final convex hull, whereas the points on the *inner* sides are not on the convex hull. Without loss of generality we consider only how to find the upper bridge b_1. Finding the lower bridge b_2 is analogous.

To find the upper bridge, one might consider taking the points with the maximum y. However, this does not work in general; u_1 can lie as far down as the point with the minimum x or maximum x value (see Figure 10.15). Instead, there is a nice solution based on binary search. Assume that the points on the convex hulls are given in order (e.g., clockwise). At each step the search algorithm will eliminate half the remaining points from consideration in either H_1 or H_2 or both. After at most $\log|H_1| + \log|H_2|$ steps the search will be left with only one point in each hull, and these will be the desired points u_1 and u_2. Figure 10.16 illustrates the rules for eliminating part of H_1 or H_2 on each step.

We now consider the cost of the algorithm. Each step of the binary search requires only constant work and depth since we only need to consider the middle two points M_1 and M_2, which can be found in constant time if the hull is kept sorted. The cost of the full binary search to find the upper bridge is therefore bounded by $D(n) = W(n) = O(\log n)$. Once we have found the upper and lower bridges, we need to remove the points on H_1 and H_2 that are not on H and append the remaining convex hull points. This requires linear work and constant depth. The overall costs of MERGEHULL are, therefore,

$$D(n) = D(n/2) + \log n = O(\log^2 n)$$
$$W(n) = 2W(n/2) + \log n + n = O(n \log n)$$

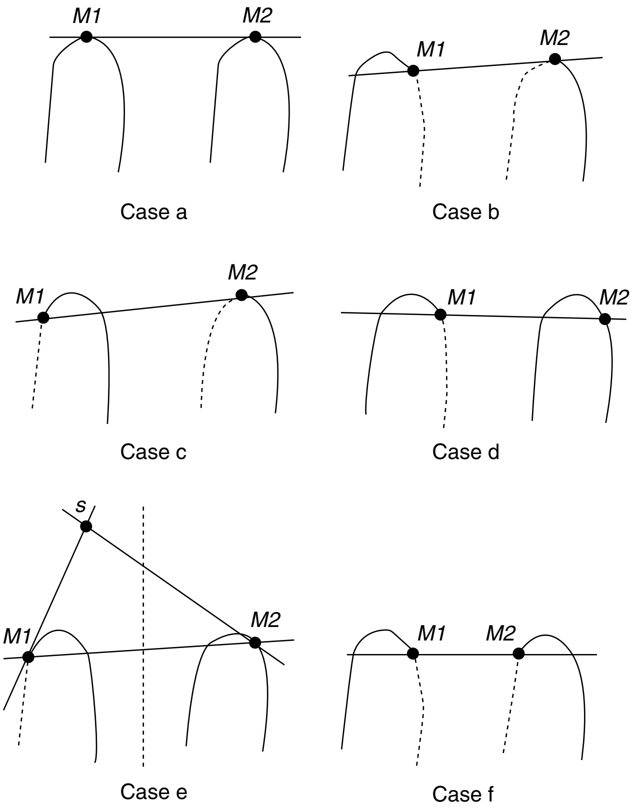


FIGURE 10.16 Cases used in the binary search for finding the upper bridge for the MergeHull. The points $M1$ and $M2$ mark the middle of the remaining hulls. The dotted lines represent the part of the hull that can be eliminated from consideration. The mirror images of cases b–e are also used. In case e, the region to eliminate depends on which side of the separating line the intersection of the tangents appears.

This algorithm can be improved to run in $O(\log n)$ depth using one of two techniques. The first involves implementing the search for the bridge points such that it runs in constant depth with linear work [Atallah and Goodrich 1988]. This involves sampling every $\sqrt{n}$th point on each hull and comparing all pairs of these two samples to narrow the search region down to regions of size $\sqrt{n}$ in constant depth. The patches then can be finished in constant depth by comparing all pairs between the two patches. The second technique [Aggarwal et al. 1988, Atallah and Goodrich 1986] uses a divide-and-conquer to separate the point set into $\sqrt{n}$ regions, solves the convex hull on each region recursively, and then merges all pairs of these regions using the binary search method. Since there are $\sqrt{n}$ regions and each of the searches takes $O(\log n)$ work, the total work for merging is $O((\sqrt{n})^2 \log n) = O(n \log n)$ and the depth is $O(\log n)$. This leads to an overall algorithm that runs in $O(n \log n)$ work and $O(\log n)$ depth.

10.8 Numerical Algorithms

There has been an immense amount of work on parallel algorithms for numerical problems. Here we briefly discuss some of the problems and results. We suggest the following sources for further information on parallel numerical algorithms: Reif [1993, Chap. 12 and Chapter 14], JáJá [1992, Chap. 8], Kumar et al. [1994, Chap. 5, Chapter 10 and Chapter 11], and Bertsekas and Tsitsiklis [1989].

10.8.1 Matrix Operations

Matrix operations form the core of many numerical algorithms and led to some of the earliest work on parallel algorithms. The most basic matrix operation is matrix multiply. The standard triply nested loop for multiplying two dense matrices is highly parallel since each of the loops can be parallelized:

Algorithm: MATRIX_MULTIPLY (A, B).

```
(l, m) := dimensions(A)
(m, n) := dimensions(B)
in parallel for i ∈ [0..l) do
    in parallel for j ∈ [0..n) do
        R_ij := sum({A_ik * B_kj : k ∈ [0..m)})
return R
```

If $l = m = n$, this routine does $O(n^3)$ work and has depth $O(\log(n))$, due to the depth of the summation. This has much more parallelism than is typically needed, and most of the research on parallel matrix multiplication has concentrated on how to use a subset of the parallelism to minimize communication costs. Sequentially, it is known that matrix multiplication can be done in better than $O(n^3)$ work. For example, Strassen's [1969] algorithm requires only $O(n^{2.81})$ work. Most of these more efficient algorithms are also easy to parallelize because of their recursive nature (Strassen's algorithm has $O(\log n)$ depth using a simple parallelization).

Another basic matrix operation is to invert matrices. Inverting dense matrices turns out to be somewhat less parallel than matrix multiplication, but still supplies plenty of parallelism for most practical purposes. When using Gauss–Jordan elimination, two of the three nested loops can be parallelized leading to an algorithm that runs with $O(n^3)$ work and $O(n)$ depth. A recursive block-based method using matrix multiplies leads to the same depth, although the work can be reduced by using one of the more efficient matrix multiplies.

Parallel algorithms for many other matrix operations have been studied, and there has also been significant work on algorithms for various special forms of matrices, such as tridiagonal, triangular, and general sparse matrices. Iterative methods for solving sparse linear systems have been an area of significant activity.

10.8.2 Fourier Transform

Another problem for which there has been a long history of parallel algorithms is the discrete Fourier transform (DFT). The fast Fourier transform (FFT) algorithm for solving the DFT is quite easy to parallelize and, as with matrix multiplication, much of the research has gone into reducing communication costs. In fact, the butterfly network topology is sometimes called the FFT network since the FFT has the same communication pattern as the network [Leighton 1992, Section 3.7]. A parallel FFT over complex numbers can be expressed as follows.

Algorithm: FFT(A).

```
n := |A|
if (n = 1) then return A
else
    in parallel do
        E := FFT({A[2i] : i ∈ [0..n/2)})
        O := FFT ({A[2i + 1] : i ∈ [0..n/2)})
    return {E[j] + O[j]e^{2πij/n} : j ∈ [0..n/2)} ++ {E[j] − O[j]e^{2πij/n} : j ∈ [0..n/2)}
```

It simply calls itself recursively on the odd and even elements and then puts the results together. This algorithm does $O(n \log n)$ work, as does the sequential version, and has a depth of $O(\log n)$.

10.9 Parallel Complexity Theory

Researchers have developed a complexity theory for parallel computation that is in some ways analogous to the theory of NP-completeness. A problem is said to belong to the class NC (Nick's class) if it can be solved in depth polylogarithmic in the size of the problem using work that is polynomial in the size of the problem [Cook 1981, Pippenger 1979]. The class NC in parallel complexity theory plays the role of P in sequential complexity, i.e., the problems in NC are thought to be tractable in parallel. Examples of problems in NC include sorting, finding minimum cost spanning trees, and finding convex hulls. A problem is said to be P-complete if it can be solved in polynomial time and if its inclusion in NC would imply that $NC = P$. Hence, the notion of P-completeness plays the role of NP-completeness in sequential complexity. (And few believe that $NC = P$.)

Although much early work in parallel algorithms aimed at showing that certain problems belong to the class NC (without considering the issue of efficiency), this work tapered off as the importance of work efficiency became evident. Also, even if a problem is P-complete, there may be efficient (but not necessarily polylogarithmic time) parallel algorithms for solving it. For example, several efficient and highly parallel algorithms are known for solving the maximum flow problem, which is P-complete.

We conclude with a short list of P-complete problems. Full definitions of these problems and proofs that they are P-complete can be found in textbooks and surveys such as Gibbons and Rytter [1990], JáJá [1992], and Karp and Ramachandran [1990]. P-complete problems are:

1. **Lexicographically first maximal independent set and clique.** Given a graph G with vertices $V = 1, 2, \ldots, n$, and a subset $S \subseteq V$, determine if S is the lexicographically first maximal independent set (or maximal clique) of G.
2. **Ordered depth-first search.** Given a graph $G = (V, E)$, an ordering of the edges at each vertex, and a subset $T \subset E$, determine if T is the depth-first search tree that the sequential depth-first algorithm would construct using this ordering of the edges.
3. **Maximum flow.**
4. **Linear programming.**
5. **The circuit value problem.** Given a Boolean circuit, and a set of inputs to the circuit, determine if the output value of the circuit is one.
6. **The binary operator generability problem.** Given a set S, an element e not in S, and a binary operator$\cdot$, determine if e can be generated from S using$\cdot$.
7. **The context-free grammar emptiness problem.** Given a context-free grammar, determine if it can generate the empty string.

Defining Terms

CRCW: This refers to a shared memory model that allows for concurrent reads (CR) and concurrent writes (CW) to the memory.

CREW: This refers to a shared memory model that allows for concurrent reads (CR) but only exclusive writes (EW) to the memory.

Depth: The longest chain of sequential dependences in a computation.

EREW: This refers to a shared memory model that allows for only exclusive reads (ER) and exclusive writes (EW) to the memory.

Graph contraction: Contracting a graph by removing a subset of the vertices.

List contraction: Contracting a list by removing a subset of the nodes.

Multiprefix: A generalization of the scan (prefix sums) operation in which the partial sums are grouped by keys.

Multiprocessor model: A model of parallel computation based on a set of communicating sequential processors.

Pipelined divide-and-conquer: A divide-and-conquer paradigm in which partial results from recursive calls can be used before the calls complete. The technique is often useful for reducing the depth of various algorithms.

Pointer jumping: In a linked structure replacing a pointer with the pointer it points to. Used for various algorithms on lists and trees. Also called recursive doubling.

PRAM model: A multiprocessor model in which all of the processors can access a shared memory for reading or writing with uniform cost.

Prefix sums: A parallel operation in which each element in an array or linked list receives the sum of all of the previous elements.

Random sampling: Using a randomly selected sample of the data to help solve a problem on the whole data.

Recursive doubling: Same as pointer jumping.

Scan: A parallel operation in which each element in an array receives the sum of all of the previous elements.

Shortcutting: Same as pointer jumping.

Symmetry breaking: A technique to break the symmetry in a structure such as a graph which can locally look the same to all of the vertices. Usually implemented with randomization.

Tree contraction: Contracting a tree by removing a subset of the nodes.

Work: The total number of operations taken by a computation.

Work-depth model: A model of parallel computation in which one keeps track of the total work and depth of a computation without worrying about how it maps onto a machine.

Work efficient: When an algorithm does no more work than some other algorithm or model. Often used when relating a parallel algorithm to the best known sequential algorithm but also used when discussing emulations of one model on another.

References

Aggarwal, A., Chazelle, B., Guibas, L., Ò'Dùnlaing, C., and Yap, C. 1988. Parallel computational geometry. *Algorithmica* 3(3):293–327.

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. 1974. *The Design and Analysis of Computer Algorithms.* Addison–Wesley, Reading, MA.

Akl, S. G. 1985. *Parallel Sorting Algorithms.* Academic Press, Toronto, Canada.

Anderson, R. J. and Miller, G. L. 1988. Deterministic parallel list ranking. In *Aegean Workshop on Computing: VLSI Algorithms and Architectures.* J. Reif, ed. Vol. 319, Lecture notes in computer science, pp. 81–90. Springer–Verlag, New York.

Anderson, G. L. and Miller, G. L. 1990. A simple randomized parallel algorithm for list-ranking. *Inf. Process. Lett.* 33(5):269–273.

Atallah, M. J., Cole, R., and Goodrich, M. T. 1989. Cascading divide-and-conquer: a technique for designing parallel algorithms. *SIAM J. Comput.* 18(3):499–532.

Atallah, M. J. and Goodrich, M. T. 1986. Efficient parallel solutions to some geometric problems. *J. Parallel Distrib. Comput.* 3(4):492–507.

Atallah, M. J. and Goodrich, M. T. 1988. Parallel algorithms for some functions of two convex polygons. *Algorithmica* 3(4):535–548.

Awerbuch, B. and Shiloach, Y. 1987. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Trans. Comput.* C-36(10):1258–1263.

Bar-Noy, A. and Kipnis, S. 1992. Designing broadcasting algorithms in the postal model for message-passing systems, pp. 13–22. In *Proc. 4th Annu. ACM Symp. Parallel Algorithms Architectures.* ACM Press, New York.

Beneš, V. E. 1965. *Mathematical Theory of Connecting Networks and Telephone Traffic.* Academic Press, New York.

Bentley, J. L. and Shamos, M. 1976. Divide-and-conquer in multidimensional space, pp. 220–230. In *Proc. ACM Symp. Theory Comput.* ACM Press, New York.

Bertsekas, D. P. and Tsitsiklis, J. N. 1989. *Parallel and Distributed Computation: Numerical Methods*. Prentice–Hall, Englewood Cliffs, NJ.

Blelloch, G. E. 1990. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, MA.

Blelloch, G. E. 1996. Programming parallel algorithms. *Commun. ACM* 39(3):85–97.

Blelloch, G. E., Chandy, K. M., and Jagannathan, S., eds. 1994. *Specification of Parallel Algorithms*. Vol. 18, DIMACS series in discrete mathematics and theoretical computer science. American Math. Soc. Providence, RI.

Blelloch, G. E. and Greiner, J. 1995. Parallelism in sequential functional languages, pp. 226–237. In *Proc. ACM Symp. Functional Programming Comput. Architecture*. ACM Press, New York.

Blelloch, G. E., Leiserson, C. E., Maggs, B. M., Plaxton, C. G., Smith, S. J., and Zagha, M. 1991. A comparison of sorting algorithms for the connection machine CM-2, pp. 3–16. In *Proc. ACM Symp. Parallel Algorithms Architectures*. Hilton Head, SC, July. ACM Press, New York.

Blelloch, G. E. and Little, J. J. 1994. Parallel solutions to geometric problems in the scan model of computation. *J. Comput. Syst. Sci.* 48(1):90–115.

Brent, R. P. 1974. The parallel evaluation of general arithmetic expressions. *J. Assoc. Comput. Mach.* 21(2):201–206.

Chan, T. M. Y., Snoeyink, J., and Yap, C. K. 1995. Output-sensitive construction of polytopes in four dimensions and clipped Voronoi diagrams in three, pp. 282–291. In *Proc. 6th Annu. ACM–SIAM Symp. Discrete Algorithms*. ACM–SIAM, ACM Press, New York.

Cole, R. 1988. Parallel merge sort. *SIAM J. Comput.* 17(4):770–785.

Cook, S. A. 1981. Towards a complexity theory of synchronous parallel computation. *Enseignement Mathematique* 27:99–124.

Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T. 1993. LogP: towards a realistic model of parallel computation, pp. 1–12. In *Proc. 4th ACM SIGPLAN Symp. Principles Pract. Parallel Programming*. ACM Press, New York.

Cypher, R. and Sanz, J. L. C. 1994. *The SIMD Model of Parallel Computation*. Springer–Verlag, New York.

Fortune, S. and Wyllie, J. 1978. Parallelism in random access machines, pp. 114–118. In *Proc. 10th Annu. ACM Symp. Theory Comput.* ACM Press, New York.

Gazit, H. 1991. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Comput.* 20(6):1046–1067.

Gibbons, P. B., Matias, Y., and Ramachandran, V. 1994. The QRQW PRAM: accounting for contention in parallel algorithms, pp. 638–648. In *Proc. 5th Annu. ACM–SIAM Symp. Discrete Algorithms*. Jan. ACM Press, New York.

Gibbons, A. and Ritter, W. 1990. *Efficient Parallel Algorithms*. Cambridge University Press, Cambridge, England.

Goldshlager, L. M. 1978. A unified approach to models of synchronous parallel machines, pp. 89–94. In *Proc. 10th Annu. ACM Symp. Theory Comput.* ACM Press, New York.

Goodrich, M. T. 1996. Parallel algorithms in geometry. In *CRC Handbook of Discrete and Computational Geometry*. CRC Press, Boca Raton, FL.

Greiner, J. 1994. A comparison of data-parallel algorithms for connected components, pp. 16–25. In *Proc. 6th Annu. ACM Symp. Parallel Algorithms Architectures*. June. ACM Press, New York.

Halperin, S. and Zwick, U. 1994. An optimal randomized logarithmic time connectivity algorithm for the EREW PRAM, pp. 1–10. In *Proc. ACM Symp. Parallel Algorithms Architectures*. June. ACM Press, New York.

Harris, T. J. 1994. A survey of pram simulation techniques. *ACM Comput. Surv.* 26(2):187–206.

Huang, J. S. and Chow, Y. C. 1983. Parallel sorting and data partitioning by sampling, pp. 627–631. In *Proc. IEEE Comput. Soc. 7th Int. Comput. Software Appl. Conf.* Nov.

JáJá, J. 1992. *An Introduction to Parallel Algorithms*. Addison–Wesley, Reading, MA.

Karlin, A. R. and Upfal, E. 1988. Parallel hashing: an efficient implementation of shared memory. *J. Assoc. Comput. Mach.* 35:876–892.

Karp, R. M. and Ramachandran, V. 1990. Parallel algorithms for shared memory machines. In *Handbook of Theoretical Computer Science — Volume A: Algorithms and Complexity*. J. Van Leeuwen, ed. MIT Press, Cambridge, MA.

Kirkpatrick, D. G. and Seidel, R. 1986. The ultimate planar convex hull algorithm? *SIAM J. Comput.* 15:287–299.

Klein, P. N. and Tarjan, R. E. 1994. A randomized linear-time algorithm for finding minimum spanning trees. In *Proc. ACM Symp. Theory Comput.* May. ACM Press, New York.

Knuth, D. E. 1973. *Sorting and Searching. Vol. 3. The Art of Computer Programming*. Addison–Wesley, Reading, MA.

Kogge, P. M. and Stone, H. S. 1973. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Comput.* C-22(8):786–793.

Kumar, V., Grama, A., Gupta, A., and Karypis, G. 1994. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin Cummings, Redwood City, CA.

Leighton, F. T. 1992. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes*. Morgan Kaufmann, San Mateo, CA.

Leiserson, C. E. 1985. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.* C-34(10):892–901.

Luby, M. 1985. A simple parallel algorithm for the maximal independent set problem, pp. 1–10. In *Proc. ACM Symp. Theory Comput.* May. ACM Press, New York.

Maon, Y., Schieber, B., and Vishkin, U. 1986. Parallel ear decomposition search (eds) and st-numbering in graphs. *Theor. Comput. Sci.* 47:277–298.

Matias, Y. and Vishkin, U. 1991. On parallel hashing and integer sorting. *J. Algorithms* 12(4):573–606.

Miller, G. L. and Ramachandran, V. 1992. A new graph triconnectivity algorithm and its parallelization. *Combinatorica* 12(1):53–76.

Miller, G. and Reif, J. 1989. Parallel tree contraction part 1: fundamentals. In *Randomness and Computation. Vol. 5. Advances in Computing Research*, pp. 47–72. JAI Press, Greenwich, CT.

Miller, G. L. and Reif, J. H. 1991. Parallel tree contraction part 2: further applications. *SIAM J. Comput.* 20(6):1128–1147.

Overmars, M. H. and Van Leeuwen, J. 1981. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.* 23:166–204.

Padua, D., Gelernter, D., and Nicolau, A., eds. 1990. *Languages and Compilers for Parallel Computing Research Monographs in Parallel and Distributed*. MIT Press, Cambridge, MA.

Pippenger, N. 1979. On simultaneous resource bounds, pp. 307–311. In *Proc. 20th Annu. Symp. Found. Comput. Sci.*

Pratt, V. R. and Stockmeyer, L. J. 1976. A characterization of the power of vector machines. *J. Comput. Syst. Sci.* 12:198–221.

Preparata, F. P. and Shamos, M. I. 1985. *Computational Geometry — An Introduction*. Springer–Verlag, New York.

Ranade, A. G. 1991. How to emulate shared memory. *J. Comput. Syst. Sci.* 42(3):307–326.

Reid-Miller, M. 1994. List ranking and list scan on the Cray C-90, pp. 104–113. In *Proc. 6th Annu. ACM Symp. Parallel Algorithms Architectures*. June. ACM Press, New York.

Reif, J. H., ed. 1993. *Synthesis of Parallel Algorithms*. Morgan Kaufmann, San Mateo, CA.

Reif, J. H. and Valiant, L. G. 1983. A logarithmic time sort for linear size networks, pp. 10–16. In *Proc. 15th Annu. ACM Symp. Theory Comput.* April. ACM Press, New York.

Savitch, W. J. and Stimson, M. 1979. Time bounded random access machines with parallel processing. *J. Assoc. Comput. Mach.* 26:103–118.

Shiloach, Y. and Vishkin, U. 1981. Finding the maximum, merging and sorting in a parallel computation model. *J. Algorithms* 2(1):88–102.

Shiloach, Y. and Vishkin, U. 1982. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms* 3:57–67.

Stone, H. S. 1975. Parallel tridiagonal equation solvers. *ACM Trans. Math. Software* 1(4):289–307.

Strassen, V. 1969. Gaussian elimination is not optimal. *Numerische Mathematik* 14(3):354–356.

Tarjan, R. E. and Vishkin, U. 1985. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.* 14(4):862–874.

Valiant, L. G. 1990a. A bridging model for parallel computation. *Commun. ACM* 33(8):103–111.

Valiant, L. G. 1990b. General purpose parallel architectures, pp. 943–971. In *Handbook of Theoretical Computer Science*. J. van Leeuwen, ed. Elsevier Science, B. V., Amsterdam, The Netherlands.

Vishkin, U. 1984. Parallel-design distributed-implementation (PDDI) general purpose computer. *Theor. Comp. Sci.* 32:157–172.

Wyllie, J. C. 1979. The Complexity of Parallel Computations. *Department of Computer Science, Tech. Rep.* TR-79-387, Cornell University, Ithaca, NY. Aug.

11 Computational Geometry

D. T. Lee
Northwestern University

11.1 Introduction

Computational geometry evolves from the classical discipline of design and analysis of algorithms, and has received a great deal of attention in the past two decades since its identification in 1975 by Shamos. It is concerned with the computational complexity of geometric problems that arise in various disciplines such as pattern recognition, computer graphics, computer vision, robotics, very large-scale integrated (VLSI) layout, operations research, statistics, etc. In contrast with the classical approach to proving mathematical theorems about geometry-related problems, this discipline emphasizes the computational aspect of these problems and attempts to exploit the underlying geometric properties possible, e.g., the metric space, to derive efficient algorithmic solutions.

The classical theorem, for instance, that a set S is convex if and only if for any $0 \leq \alpha \leq 1$ the convex combination $\alpha p + (1 - \alpha)q = r$ is in S for any pair of elements $p, q \in S$, is very fundamental in establishing convexity of a set. In geometric terms, a body S in the Euclidean space is convex if and only if the line segment joining any two points in S lies totally in S. But this theorem per se is not suitable for computational purposes as there are infinitely many possible pairs of points to be considered. However, other properties of convexity can be utilized to yield an algorithm. Consider the following problem. Given a simple closed Jordan polygonal curve, determine if the interior region enclosed by the curve is convex. This problem can be readily solved by observing that if the line segments defined by all pairs of vertices of the polygonal curve, $\overline{v_i, v_j}, i \neq j, 1 \leq i, j \leq n$, where n denotes the total number of vertices, lie totally inside the region, then the region is convex. This would yield a straightforward algorithm with time complexity $O(n^3)$, as there are $O(n^2)$ line segments, and to test if each line segment lies totally in the region takes $O(n)$ time by comparing it against every polygonal segment. As we shall show, this problem can be solved in $O(n)$ time by utilizing other geometric properties.

At this point, an astute reader might have come up with an $O(n)$ algorithm by making the observation: Because the interior angle of each vertex must be strictly less than π in order for the region to be convex, we just have to check for every consecutive three vertices v_{i-1}, v_i, v_{i+1} that the angle at vertex v_i is less than π. (A vertex whose internal angle has a measure less than π is said to be *convex*; otherwise, it is said to be *reflex*.) One may just be content with this solution. Mathematically speaking, this solution is fine and indeed runs in $O(n)$ time. The problem is that the algorithm implemented in this straightforward manner without care may produce an incorrect answer when the input polygonal curve is ill formed. That is, if the input polygonal curve is not simple, i.e., it self-intersects, then the *enclosed* region by this closed curve is not well defined. The algorithm, without checking this simplicity condition, may produce a wrong answer. Note that the preceding observation that all of the vertices must be convex in order to have a convex region is only a necessary condition. Only when the input polygonal curve is *verified* to be simple will the algorithm produce a correct answer. But to verify whether the input polygonal curve self-intersects or not is no longer as straightforward. The fact that we are dealing with computer solutions to geometric problems may make the task of designing an algorithm and proving its correctness nontrivial.

An objective of this discipline in the theoretical context is to prove lower bounds of the complexity of geometric problems and to devise algorithms (giving upper bounds) whose complexity *matches* the lower bounds. That is, we are interested in the *intrinsic* difficulty of geometric computational problems under a certain computation model and at the same time are concerned with the algorithmic solutions that are provably optimal in the worst or average case. In this regard, the asymptotic time (or space) complexity of an algorithm is of interest. Because of its applications to various science and engineering related disciplines, researchers in this field have begun to address the efficacy of the algorithms, the issues concerning robustness and numerical stability [Fortune 1993], and the actual running times of their implementions.

In this chapter, we concentrate mostly on the theoretical development of this field in the context of sequential computation. Parallel computation geometry is beyond the scope of this chapter. We will adopt the *real* random access machine (RAM) model of computation in which all arithmetic operations, comparisons, kth-root, exponential or logarithmic functions take unit time. For more details refer to Edelsbrunner [1987], Mulmuley [1994], and Preparata and Shamos [1985]. We begin with a summary of problem solving techniques that have been developed [Lee and Preparata 1982, O'Rourke 1994, Yao 1994] and then discuss a number of topics that are central to this field, along with additional references for further reading about these topics.

11.2 Problem Solving Techniques

We give an example for each of the eight major problem-solving paradigms that are prevalent in this field. In subsequent sections we make reference to these techniques whenever appropriate.

11.2.1 Incremental Construction

This is the simplest and most intuitive method, also known as *iterative method*. That is, we compute the solution in an iterative manner by considering the input incrementally.

Consider the problem of computing the line arrangements in the plane. Given is a set $\mathcal{L}$ of n straight lines in the plane, and we want to compute the partition of the plane induced by $\mathcal{L}$. One obvious approach is to compute the partition iteratively by considering one line at a time [Chazelle et al. 1985]. As shown in Figure 11.1, when line i is inserted, we need to traverse the regions that are intersected by the line and construct the new partition at the same time. One can show that the traversal and repartitioning of the intersected regions can be done in $O(n)$ time per insertion, resulting in a total of $O(n^2)$ time. This algorithm is asymptotically optimal because the running time is proportional to the amount of space required to represent the partition. This incremental approach also generalizes to higher dimensions. We conclude with the theorem [Edelsbrunner et al. 1986].

Theorem 11.1 *The problem of computing the arrangement $\mathcal{A}(H)$ of a set H of n hyperplanes in $\Re^k$ can be solved iteratively in $O(n^k)$ time and space, which is optimal.*

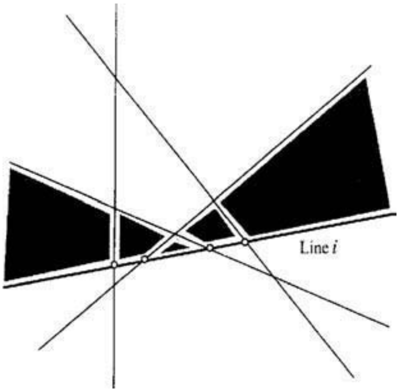


FIGURE 11.1 Incremental construction of line arrangement: phase i.

11.2.2 Plane Sweep

This approach works most effectively for two-dimensional problems for which the solution can be computed incrementally as the entire input is scanned in a certain order. The concept can be easily generalized to higher dimensions [Bieri and Nef 1982]. This is also known as the *scan-line* method in computer graphics and is used for a variety of applications such as shading and polygon filling, among others.

Consider the problem of computing the *measure* of the union of n isothetic rectangles, i.e., whose sides are parallel to the coordinate axes. We would proceed with a *vertical* sweep line, sweeping across the plane from left to right. As we sweep the plane, we need to keep track of the rectangles that intersect the current sweep line and those that are yet to be visited. In the meantime we compute the area covered by the union of the rectangles seen so far. More formally, associated with this approach there are two basic data structures containing all *relevant* information that should be maintained.

1. *Event schedule* defines a sequence of *event points* that the sweep-line status will change. In this example, the sweep-line status will change only at the left and right boundary edges of each rectangle.
2. *Sweep-line status* records the information of the geometric structure that is being swept. In this example the sweep-line status keeps track of the set of rectangles intersecting the current sweep line.

The event schedule is normally represented by a *priority queue*, and the list of events may change dynamically. In this case, the events are static; they are the x-coordinates of the left and right boundary edges of each rectangle. The sweep-line status is represented by a suitable data structure that supports insertions, deletions, and computation of the partial solution at each event point. In this example a *segment tree* attributed to Bentley is sufficient [Preparata and Shamos 1985]. Because we are computing the area of the rectangles, we need to be able to know the *new* area covered by the current sweep line between two adjacent event points. Suppose at event point x_{i-1} we maintain a partial solution $\mathcal{A}_{i-1}$. In Figure 11.2 the shaded area S needs to be added to the partial solution, that is, $\mathcal{A}_i = \mathcal{A}_{i-1} + S$. The shaded area is equal to the total measure, denoted sum_ℓ, of the union of vertical line segments representing the intersection of the rectangles and the current sweep line times the distance between the two event points x_i and x_{i-1}. If the next event corresponds to the left boundary of a rectangle, the corresponding vertical segment, $\overline{p, q}$ in Figure 11.2, needs to be inserted to the segment tree. If the next event corresponds to a right boundary edge, the segment, $\overline{u, v}$ needs to be deleted from the segment tree. In either case, the total measure sum_ℓ should be updated accordingly. The correctness of this algorithm can be established by observing that the partial solution obtained for the rectangles to the *left* of the sweep line is maintained correctly. In fact, this property is typical of any algorithm based on the plane-sweep technique.

Because the segment tree structure supports segment insertions and deletions and the update (of sum_ℓ) operation in $O(\log n)$ time per event point, the total amount of time needed is $O(n \log n)$.

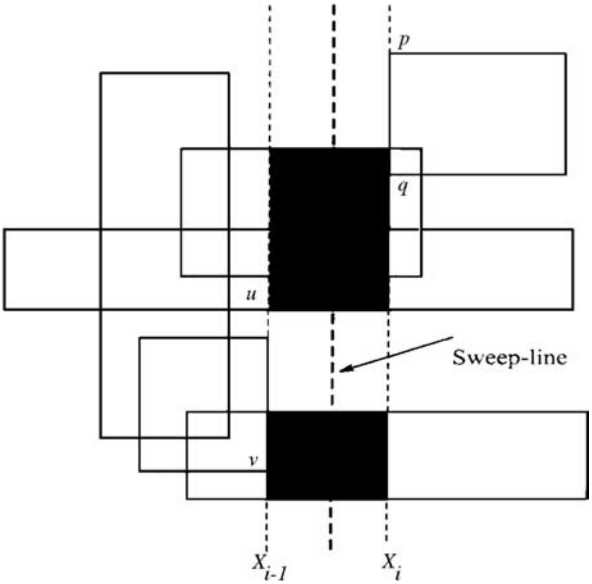


FIGURE 11.2 The plane-sweep approach to the measure problem in two dimensions.

The measure of the union of rectangles in higher dimensions also can be solved by the plane-sweep technique with quad trees, a generalization of segment trees.

Theorem 11.2 *The problem of computing the measure of n isothetic rectangles in k dimensions can be solved in $O(n \log n)$ time, for $k \leq 2$ and in $O(n^{k-1})$ time for $k \geq 3$.*

The time bound is asymptotically optimal. Even in one dimension, i.e., computing the total length of the union of n intervals requires $\Omega(n \log n)$ time (see Preparata and Shamos [1985]).

We remark that the sweep line used in this approach is not necessarily a straight line. It can be a topological line as long as the objects stored in the sweep line status are ordered, and the method is called *topological sweep* [Asano et al. 1994, Edelsbrunner and Guibas 1989]. Note that the measure of isothetic rectangles can also be solved using the divide-and-conquer paradigm to be discussed.

11.2.3 Geometric Duality

This is a geometric transformation that maps a given problem into its equivalent form, preserving certain geometric properties so as to manipulate the objects in a more convenient manner. We will see its usefulness for a number of problems to be discussed. Here let us describe a transformation in k-dimensions, known as *polarity* or *duality*, denoted $\mathcal{D}$, that maps d-dimensional varieties to $(k-1-d)$-dimensional varieties, $0 \leq d < k$.

Consider any point $p = (\pi_1, \pi_2, \ldots, \pi_k) \in \Re^k$ other than the origin. The dual of p, denoted $\mathcal{D}(p)$, is the hyperplane $\pi_1 x_1 + \pi_2 x_2 + \cdots + \pi_k x_k = 1$. Similarly, a hyperplane that does not contain the origin is mapped to a point such that $\mathcal{D}(\mathcal{D}(p)) = p$. Geometrically speaking, point p is mapped to a hyperplane whose normal is the vector determined by p and the origin and whose distance to the origin is the reciprocal of that between p and the origin. Let S denote the unit sphere $S : x_1^2 + x_2^2 + \cdots + x_k^2 = 1$. If point p is external to S, then it is mapped to a hyperplane $\mathcal{D}(p)$ that intersects S at those points q that admit supporting hyperplanes h such that $h \cap S = q$ and $p \in h$. In two dimensions a point p outside of the unit disk will be mapped to a line intersecting the disk at two points, q_1 and q_2, such that line segments $\overline{p, q_1}$ and $\overline{p, q_2}$ are tangent to the disk. Note that the distances from p to the origin and from the line $\mathcal{D}(p)$ to the origin are reciprocal to each other. Figure 11.3a shows the duality transformation in two dimensions. In

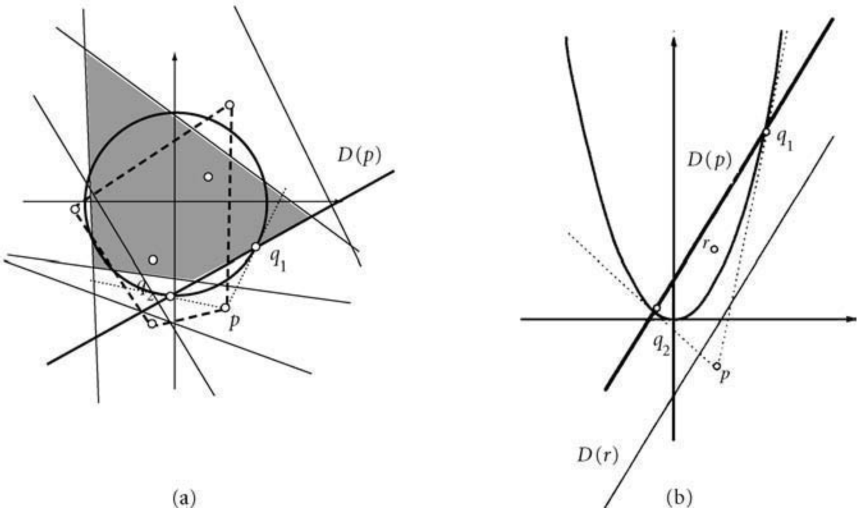


FIGURE 11.3 Geometric duality transformation in two dimensions.

particular, point p is mapped to the line shown in boldface. For each hyperplane $\mathcal{D}(p)$, let $\mathcal{D}(p)^+$ denote the half-space that contains the origin and let $\mathcal{D}(p)^-$ denote the other half-space.

The duality transformation not only leads to dual arrangements of hyperplanes and configurations of points and vice versa, but also preserves the following properties.

Incidence: Point p belongs to hyperplane h if and only if point $\mathcal{D}(\langle)$ belongs to hyperplane $\mathcal{D}(p)$.
Order: Point p lies in half-space h^+ (respectively, h^-) if and only if point $\mathcal{D}(\langle)$ lies in half-space $\mathcal{D}(p)^+$ (respectively, $\mathcal{D}(p)^-$).

Figure 11.3a shows the convex hull of a set of points that are mapped by the duality transformation to the shaded region, which is the common intersection of the half-planes $\mathcal{D}(p)^+$ for all points p.

Another transformation using the unit paraboloid U, represented as $U : x_k = x_1^2 + x_2^2 + \cdots + x_{k-1}^2$, can also be similarly defined. That is, point $p = (\pi_1, \pi_2, \ldots, \pi_k) \in R^k$ is mapped to a hyperplane $\mathcal{D}_{\sqcap}(\surd)$ represented by the equation $x_k = 2\pi_1 x_1 + 2\pi_2 x_2 + \cdots + 2\pi_{k-1} x_{k-1} - \pi_k$. And each nonvertical hyperplane is mapped to a point in a similar manner such that $\mathcal{D}_u(\mathcal{D}_u(p)) = p$. Figure 11.3b illustrates the two-dimensional case, in which point p is mapped to a line shown in boldface. For more details see, e.g., Edelsbrunner [1987] and Preparata and Shamos [1985].

11.2.4 Locus

This approach is often used as a preprocessing step for a geometric *searching* problem to achieve faster query-answering response time. For instance, given a *fixed* database consisting of geographical locations of post offices, each represented by a point in the plane, one would like to be able to efficiently answer queries of the form: "what is the nearest post office to location q?" for some query point q. The locus approach to this problem is to partition the plane into n regions, each of which consists of the locus of *query* points for which the *answer* is the same. The partition of the plane is the so-called *Voronoi* diagram discussed subsequently. In Figure 11.7, the post office closest to query point q is site s_i. Once the Voronoi diagram is available, the query problem reduces to that of locating the region that contains the query, an instance of the point-location problem discussed in Section 11.3.

11.2.5 Divide-and-Conquer

This is a classic problem-solving technique and has proven to be very powerful for geometric problems as well. This technique normally involves partitioning of the given problem into several subproblems,

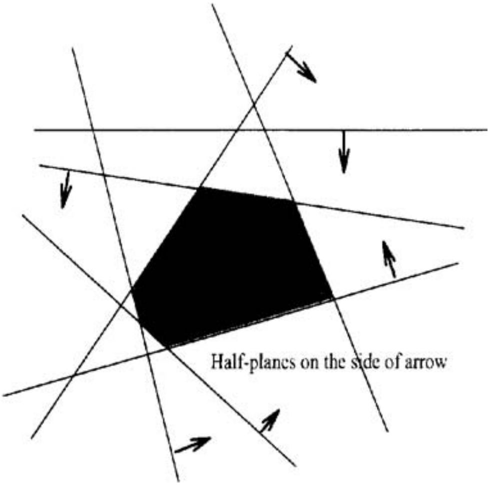


FIGURE 11.4 The common intersection of half-planes.

recursively solving each subproblem, and then combining the solutions to each of the subproblems to obtain the final solution to the original problem. We illustrate this paradigm by considering the problem of computing the common intersection of n half-planes in the plane. Given is a set S of n half-planes, h_i, represented by $a_i x + b_i y \leq c_i, i = 1, 2, \ldots, n$. It is well known that the common intersection of half-planes, denoted $CI(S) = \bigcap_{i=1}^{n} h_i$, is a convex set, which may or may not be bounded. If it is bounded, it is a convex polygon. See Figure 11.4, in which the shaded area is the common intersection.

The divide-and-conquer paradigm consists of the following steps.

Algorithm Common_Intersection_D&C (S)

```
1. If $|S| \leq 3$, compute the intersection $CI(S)$ explicitly. Return ($CI(S)$).
2. Divide S into two approximately equal subsets $S_1$ and $S_2$.
3. $CI(S_1)$ = Common_Intersection_D&C($S_1$).
4. $CI(S_2)$ = Common_Intersection_D&C($S_2$).
5. $CI(S)$ = Merge($CI(S_1)$, $CI(S_2)$).
6. Return ($CI(S)$).
```

The key step is the *merge* of two common intersections. Because $CI(S_1)$ and $CI(S_2)$ are convex, the merge step basically calls for the computation of the intersection of two convex polygons, which can be solved in time proportional to the size of the polygons (cf. subsequent section on intersection). The running time of the divide-and-conquer algorithm is easily shown to be $O(n \log n)$, as given by the following recurrence formula, where $n = |S|$:

$$T(3) = O(1)$$
$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) + M\left(\frac{n}{2}, \frac{n}{2}\right)$$

where $M(n/2, n/2) = O(n)$ denotes the merge time (step 5).

Theorem 11.3 *The common intersection of n half-planes can be solved in $O(n \log n)$ time by the divide-and-conquer method.*

The time complexity of the algorithm is asymptotically optimal, as the problem of sorting can be reduced to it [Preparata and Shamos 1985].

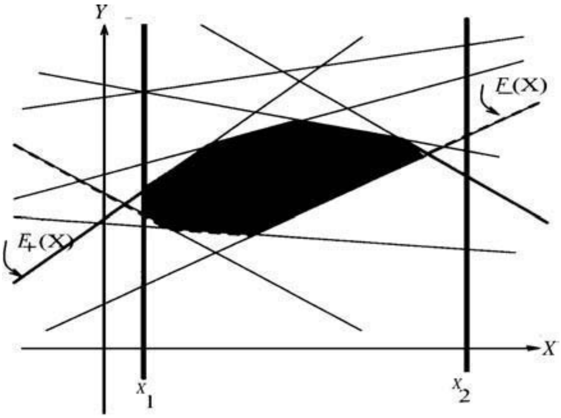


FIGURE 11.5 Feasible region defined by upward- and downward-convex piecewise linear functions.

11.2.6 Prune-and-Search

This approach, developed by Dyer [1986] and Megiddo [1983a, 1983b, 1984], is a very powerful method for solving a number of geometric optimization problems, one of which is the well-known linear programming problem. Using this approach, they obtained an algorithm whose running time is linear in the number of constraints. For more development of linear programming problems, see Megiddo [1983c, 1986]. The main idea is to prune away a fraction of *redundant* input constraints in each iteration while searching for the solution. We use a two-dimensional linear programming problem to illustrate this approach. Without loss of generality, we consider the following linear programming problem:

$$\begin{aligned} &\text{Minimize} && Y \\ &\text{subject to} && \alpha_i X + \beta_i Y + \gamma_i \leq 0, \quad i = 1, 2, \ldots, n \end{aligned}$$

These n constraints are partitioned into three classes, C_0, C_+, C_-, depending on whether β_i is zero, positive, or negative, respectively. The constraints in class C_0 define an X-interval $[x_1, x_2]$, which constrains the solution, if any. The constraints in classes C_+ and C_- define, however, upward- and downward-convex piecewise linear functions $F_+(X)$ and $F_-(X)$ delimiting the feasible region* (Figure 11.5). The problem now becomes

$$\begin{aligned} &\text{Minimize} && F_-(X) \\ &\text{subject to} && F_-(X) \leq F_+(X) \\ & && x_1 \leq X \leq x_2 \end{aligned}$$

Let λ^* denote the optimal solution, if it exists. The values of $F_-(\lambda)$ and $F_+(\lambda)$ for any λ can be computed in $O(n)$ time, based on the slopes $-\alpha_i/\beta_i$. Thus, in $O(n)$ time one can determine for any $\lambda' \in [x_1, x_2]$ if (1) λ' is infeasible, and there is no solution, (2) λ' is infeasible, and we know a feasible solution is less or greater than λ', (3) $\lambda' = \lambda^*$, or (4) λ' is feasible, and whether λ^* is less or greater than λ'.

To choose λ' we partition constraints in classes C_- and C_+ into pairs and find the abscissa $\lambda_{i,j}$ of their intersection. If $\lambda_{i,j} \notin [x_1, x_2]$ then one of the constraints can be eliminated as redundant. For those $\lambda_{i,j}$ that are in $[x_1, x_2]$ we find in $O(n)$ time [Dobkin and Munro 1981] the median $\lambda'_{i,j}$ and compute $F_-(\lambda'_{i,j})$ and $F_+(\lambda'_{i,j})$. By the preceding arguments that we can determine where λ^* should lie, we know one-half of the $\lambda_{i,j}$ do not lie in the region containing λ^*. Therefore, one constraint of the corresponding pair can

*These upward- and downward-convex functions are also known as the upper and lower *envelopes* of the line arrangements for lines belonging to classes C_- and C_+, respectively.

be eliminated. The process iterates. In other words, in each iteration at least a fixed fraction $\delta = 1/4$ of the current constraints can be eliminated. Because each iteration takes $O(n)$ time, the total time spent is $Cn + C\delta n + \cdots = O(n)$. In higher dimensions, we have the following result due to Dyer [1986] and Clarkson [1986].

Theorem 11.4 *A linear program in k-dimensions with n constraints can be solved in $O(3^{k^2} n)$ time.*

We note here some of the new recent developments for linear programming. There are several randomized algorithms for this problem, of which the best expected complexity, $O(k^2 n + k^{k/2+O(1)} \log n)$ is due to Clarkson [1988], which is later improved by Matoušek et al. [1992] to run in $O(k^2 n + e^{O(\sqrt{k\ell n k})} \log n)$. Clarkson's [1988] algorithm is applicable to work in a general framework, which includes various other geometric optimization problems, such as *smallest enclosing ellipsoid*. The best known deterministic algorithm for linear programming is due to Chazelle and Matoušek [1993], which runs in $O(k^{7k+o(k)} n)$ time.

11.2.7 Dynamization

Techniques have been developed for query-answering problems, classified as *geometric searching* problems, in which the underlying database is changing over (discrete) time. A typical geometric searching problem is the *membership* problem, i.e., given a set $\mathcal{D}$ of objects, determine if x is a member of $\mathcal{D}$, or the *nearest neighbor searching* problem, i.e., given a set $\mathcal{D}$ of objects, find an object that is closest to x according to some distance measure. In the database area, these two problems are referred to as the *exact match* and *best match* queries. The idea is to make use of good data structures for a static database and enhance them with dynamization mechanisms so that updates of the database can be accommodated on line and yet queries to the database can be answered efficiently.

A general query Q contains a variable of type $T1$ and is asked of a set of objects of type $T2$. The answer to the query is of type $T3$. More formally, Q can be considered as a mapping from $T1$ and subsets of $T2$ to $T3$, that is, $Q : T1 \times 2^{T2} \rightarrow T3$. The class of geometric searching problems to which the dynamization techniques are applicable is the class of *decomposable searching problems* [Bentley and Saxe 1980].

Definition 11.1 A searching problem with query Q is decomposable if there exists an efficiently computable associative, and communtantive binary operator @ satisfying the condition

$$Q(x, A \cup B) = @(Q(x, A), Q(x, B))$$

In other words, the answer to a query Q in $\mathcal{D}$ can be computed by the answers to two subsets $\mathcal{D}_\infty$ and $\mathcal{D}_\in$ of $\mathcal{D}$. The membership problem and the nearest-neighbor searching problem previously mentioned are decomposable.

To answer queries efficiently, we have a data structure to support various update operations. There are typically three measures to evaluate a static data structure $\mathcal{A}$. They are:

1. $P_{\mathcal{A}}(N)$, the preprocessing time required to build $\mathcal{A}$
2. $S_{\mathcal{A}}(N)$, the storage required to represent $\mathcal{A}$
3. $Q_{\mathcal{A}}(N)$, the query response time required to search in $\mathcal{A}$

where N denotes the number of elements represented in $\mathcal{A}$. One would add another measure $U_{\mathcal{A}}(N)$ to represent the *update* time.

Consider the nearest-neighbor searching problem in the Euclidean plane. Given a set of n points in the plane, we want to find the nearest neighbor of a query point x. One can use the Voronoi diagram data structure $\mathcal{A}$ (cf. subsequent section on Voronoi diagrams) and point location scheme (cf. subsequent section on point location) to achieve the following: $P_{\mathcal{A}}(n) = O(n \log n)$, $S_{\mathcal{A}}(n) = O(n)$, and $Q_{\mathcal{A}}(n) = O(\log n)$. We now convert the static data structure $\mathcal{A}$ to a dynamic one, denoted $\mathcal{D}$, to support insertions and deletions as well. There are a number of dynamization techniques, but we describe the technique developed by van Leeuwan and Wood [1980] that provides the general flavor of the approach.

The general principle is to decompose $\mathcal{A}$ into a collection of separate data structures so that each update can be confined to one or a small, fixed number of them; however, to avoid degrading the query response time we cannot afford to have excessive fragmentation because queries involve the entire collection.

Let $\{x_k\}_{k\geq 1}$ be a sequence of increasing integers, called *switch points*, where x_k is divisible by k and $x_{k+1}/(k+1) > x_k/k$. Let $x_0 = 0, y_k = x_k/k$, and n denote the current size of the point set. For a given *level* k, $\mathcal{D}$ consists of $(k+1)$ static structures of the same type, one of which, called *dump* is designated to allow for insertions. Each substructure $\mathcal{B}$ has size $y_k \leq s(\mathcal{B}) \leq \dagger_{\|+\infty}$, and the dump has size $0 \leq s(dump) < y_{k+1}$. A block $\mathcal{B}$ is called *low* or *full* depending on whether $s(\mathcal{B}) = \dagger_{\|}$ or $s(\mathcal{B}) = \dagger_{\|+\infty}$, respectively, and is called *partial* otherwise. When an insertion to the dump makes its size equal to y_{k+1}, it becomes a full block and any nonfull block can be used as the dump. If all blocks are full, we switch to the next level. Note that at this point the total size is $y_{k+1} * (k+1) = x_{k+1}$. That is, at the beginning of level $k+1$, we have $k+1$ low blocks and we create a new dump, which has size 0. When a deletion from a low block occurs, we need to borrow an element either from the dump, if it is not empty, or from a partial block. When all blocks are low and $s(dump) = 0$, we switch to level $k-1$, making the low block from which the latest deletion occurs the *dump*. The level switching can be performed in $O(1)$ time. We have the following:

Theorem 11.5 *Any static data structure $\mathcal{A}$ used for a decomposable searching problem can be transformed into a dynamic data structure $\mathcal{D}$ for the same problem with the following performance. For $x_k \leq n < x_{k+1}$, $Q_{\mathcal{D}}(n) = O(kQ_{\mathcal{A}}(y_{k+1}))$, $U_{\mathcal{D}}(n) = O(C(n)+U_{\mathcal{A}}(y_{k+1}))$, and $S_{\mathcal{D}}(n) = O(kS_{\mathcal{A}}(y_{k+1}))$, where $C(n)$ denotes the time needed to look up the block which contains the data when a deletion occurs.*

If we choose, for example, x_k to be the first multiple of k greater than or equal to 2^k, that is, $k = \log_2 n$, then y_k is about $n/\log_2 n$. Because we know there exists an $\mathcal{A}$ with $Q_{\mathcal{A}}(n) = O(\log n)$ and $U_{\mathcal{A}}(n) = P_{\mathcal{A}}(n) = O(n \log n)$, we have the following corollary.

Corollary 11.1 The nearest-neighbor searching problem in the plane can be solved in $O(\log^2 n)$ query time and $O(n)$ update time. [Note that $C(n)$ in this case is $O(\log n)$.]

There are other dynamization schemes that exhibit various query-time/space and query-time/update-time tradeoffs. The interested reader is referred to Chiang and Tamassia [1992], Edelsbrunner [1987], Mehlhorn [1984], Overmars [1983], and Preparata and Shamos [1985] for more information.

11.2.8 Random Sampling

Randomized algorithms have received a great deal of attention recently because of their potential applications. See Chapter 4 for more information. For a variety of geometric problems, randomization techniques help in building geometric subdivisions and data structures to quickly answer queries about such subdivisions. The resulting randomized algorithms are simpler to implement and/or asymptotically faster than those previously known. It is important to note that the focus of randomization is *not* on random input, such as a collection of points randomly chosen uniformly and independently from a region. We are concerned with algorithms that use a source of random numbers and analyze their performance for an arbitrary input. Unlike *Monte Carlo* algorithms, whose output may be incorrect (with very low probability), the randomized algorithms, known as *Las Vegas* algorithms, considered here are guaranteed to produce a correct output.

There are a good deal of newly developed randomized algorithms for geometric problems. See Du and Hwang [1992] for more details. Randomization gives a general way to divide and conquer geometric problems and can be used for both parallel and serial computation. We will use a familiar example to illustrate this approach.

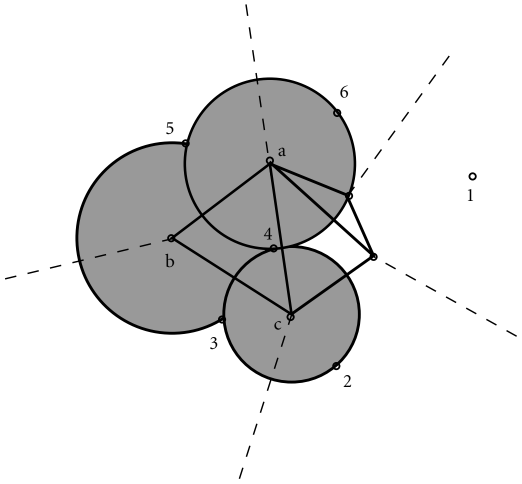


FIGURE 11.6 A triangulation of the Voronoi diagram of six sites and $K_{\mathcal{R}}(T)$, $T = \Delta(a, b, c)$.

Let us consider the problem of nearest-neighbor searching discussed in the preceding subsection. Let $\mathcal{D}$ be a set of n points in the plane and q be the query point. A simple approach to this problem is:

Algorithm S

- *Compute the distance to* q *for each point* $p \in \mathcal{D}$.
- *Return the point* p *whose distance is the smallest.*

It is clear that Algorithm S, requiring $O(n)$ time, is not suitable if we need to answer many queries of this type. To obtain faster query response time one can use the technique discussed in the preceding subsection. An alternative is to use the *random sampling* technique as follows. We pick a random sample, a subset $\mathcal{R} \subset \mathcal{D}$ of size r. Let point $p \in \mathcal{R}$ be the nearest neighbor of q in $\mathcal{R}$. The open disk $K_{\mathcal{R}}(q)$ centered at q and passing through p does not contain any other point in $\mathcal{R}$. The answer to the query is either p or some point of $\mathcal{D}$ that lies in $K_{\mathcal{R}}(q)$.

We now extend the above observation to a finite region G in the plane. Let $K_{\mathcal{R}}(G)$ be the union of disks $K_{\mathcal{R}}(r)$ for all $r \in G$. If a query q lies in G, the nearest neighbor of q must be in $K_{\mathcal{R}}(G)$ or in $\mathcal{R}$. Let us consider the Voronoi diagram, $\mathcal{V}(\mathcal{R})$ of $\mathcal{R}$ and a triangulation, $\Delta(\mathcal{V}(\mathcal{R}))$. For each triangle T with vertices a, b, c of $\Delta(\mathcal{V}(\mathcal{R}))$ we have $K_{\mathcal{R}}(T) = K_{\mathcal{R}}(a) \cup K_{\mathcal{R}}(b) \cup K_{\mathcal{R}}(c)$, shown as the shaded area in Figure 11.6. A probability lemma [Clarkson 1988] shows that with probability at least $1 - O(1/n^2)$ the candidate set $\mathcal{D} \cap K_{\mathcal{R}}(T)$ for all $T \in \Delta(\mathcal{V}(\mathcal{R}))$ contains $O(\log n)n/r$ points. More precisely, if $r > 5$ then with probability at least $1 - e^{-C/2+3\ell nr}$ each open disk $K_{\mathcal{R}}(r)$ for $r \in \mathcal{R}$ contains no more than Cn/r points of $\mathcal{D}$. If we choose r to be $\sqrt{n}$, the query time becomes $O(\sqrt{n} \log n)$, a speedup from Algorithm S. If we apply this scheme recursively to the candidate sets of $\Delta(\mathcal{V}(\mathcal{R}))$, we can get a query time $O(\log n)$ [Clarkson 1988].

There are many applications of these random sampling techniques. Derandomized algorithms were also developed. See, e.g., Chazelle and Friedman [1990] for a deterministic view of random sampling and its use in geometry.

11.3 Classes of Problems

In this section we aim to touch upon classes of problems that are fundamental in this field and describe solutions to them, some of which may be nontrivial. The reader who needs further information about these problems is strongly encouraged to refer to the original articles cited in the references.

11.3.1 Convex Hull

The convex hull of a set of points in $\Re^k$ is the most fundamental problem in computational geometry. Given is a set of points, and we are interested in computing its convex hull, which is defined to be the smallest convex body containing these points. Of course, the first question one has to answer is how to represent the convex hull. An implicit representation is just to list all of the extreme points,* whereas an explicit representation is to list all of the extreme d-faces of dimensions $d = 0, 1, \dots, k-1$. Thus, the complexity of any convex hull algorithm would have two parts, computation part and the output part. An algorithm is said to be *output sensitive* if its complexity depends on the size of output.

Definition 11.2 The convex hull of a set S of points in $\Re^k$ is the smallest convex set containing S. In two dimensions, the convex hull is a convex polygon containing S; in three dimensions it is a convex polyhedron.

11.3.1.1 Convex Hulls in Two and Three Dimensions

For an arbitrary set of n points in two and three dimensions, we can compute the convex hull using the *Graham scan*, *gift-wrapping*, or *divide-and-conquer* paradigm, which are briefly described next.

Recall that the convex hull of an arbitrary set of points in two dimensions is a convex polygon. The Graham scan computes the convex hull by (1) sorting the input set of points with respect to an interior point, say, O, which is the centroid of the first three noncollinear points, (2) connecting these points into a star-shaped polygon P centered at O, and (3) performing a linear scan to compute the convex hull of the polygon [Preparata and Shamos 1985]. Because step 1 is the dominating step, the Graham scan takes $O(n \log n)$ time.

One can also use the gift-wrapping technique to compute the convex polygon. Starting with a vertex that is known to be on the convex hull, say, the point O, with the smallest y-coordinate, we sweep a half-line emanating from O counterclockwise. The first point v_1 we hit will be the next point on the convex polygon. We then march to v_1, repeat the same process, and find the next vertex v_2. This process terminates when we reach O again. This is similar to wrapping an object with a *rope*. Finding the next vertex takes time proportional to the number of points remaining. Thus, the total time spent is $O(n\mathcal{H})$, where $\mathcal{H}$ denotes the number of points on the convex polygon. The gift-wrapping algorithm is output sensitive and is more efficient than Graham scan if the number of points on the convex polygon is small, that is, $o(\log n)$.

One can also use the divide-and-conquer paradigm. As mentioned previously, the key step is the merge of two convex hulls, each of which is the solution to a subproblem derived from the recursive step. In the division step, we can recursively separate the set into two subsets by a vertical line L. Then the merge step basically calls for computation of two common tangents of these two convex polygons. The computation of the common tangents, also known as *bridges* over line L, begins with a segment connecting the rightmost point l of the left convex polygon to the leftmost point r of the right convex polygon. Advancing the endpoints of this segment in a *zigzag* manner we can reach the top (or the bottom) common tangent such that the entire set of points lies on one side of the line containing the tangent. The running time of the divide-and-conquer algorithm is easily shown to be $O(n \log n)$.

A more sophisticated output-sensitive and optimal algorithm, which runs in $O(n \log \mathcal{H})$ time, has been developed by Kirkpatrick and Seidel [1986]. It is based on a variation of the divide-and-conquer paradigm. The main idea in achieving the optimal result is that of eliminating *redundant* computations. Observe that in the divide-and-conquer approach after the common tangents are obtained, some vertices that used to belong to the left and right convex polygons must be deleted. Had we known these vertices were not on the final convex hull, we could have saved time by not computing them. Kirkpatrick and Seidel capitalized on this concept and introduced the *marriage-before-conquest* principle. They construct the convex hull by

*A point in S is an extreme point if it cannot be expressed as a convex combination of other points in S. In other words, the convex hull of S would change when an extreme point is removed from S.

computing the upper and lower hulls of the set; the computations of these two hulls are symmetric. It performs the *divide* step as usual that decomposes the problem into two subproblems of approximately equal size. Instead of computing the upper hulls recursively for each subproblem, it finds the common tangent segment of the two yet-to-be-computed upper hulls and proceeds recursively. One thing that is worth noting is that the points known not to be on the (convex) upper hull are discarded before the algorithm is invoked recursively. This is the key to obtaining a time bound that is both output sensitive and asymptotically optimal.

The divide-and-conquer scheme can be easily generalized to three dimensions. The merge step in this case calls for computing common supporting faces that *wrap* two recursively computed convex polyhedra. It is observed by Preparata and Hong that the common supporting faces are computed from connecting two *cyclic* sequences of edges, one on each polyhedron [Preparata and Shamos 1985]. The computation of these supporting faces can be accomplished in linear time, giving rise to an $O(n \log n)$ time algorithm. By applying the marriage-before-conquest principle Edelsbrunner and Shi [1991] obtained an $O(n \log^2 \mathcal{H})$ algorithm.

The gift-wrapping approach for computing the convex hull in three dimensions would mimic the process of wrapping a gift with a piece of paper and has a running time of $O(n\mathcal{H})$.

11.3.1.2 Convex Hulls in *k*-Dimensions, $k > 3$

For convex hulls of higher dimensions, a recent result by Chazelle [1993] showed that the convex hull can be computed in time $O(n \log n + n^{\lfloor k/2 \rfloor})$, which is optimal in all dimensions $k \geq 2$ in the worst case. But this result is insensitive to the output size. The gift-wrapping approach generalizes to higher dimensions and yields an output-sensitive solution with running time $O(n\mathcal{H})$, where $\mathcal{H}$ is the total number of i-faces, $i = 0, 1, \ldots, k-1$, and $\mathcal{H} = O(n^{\lfloor k/2 \rfloor})$ [Edelsbrunner 1987]. One can also use the *beneath-beyond* method of adding points one at a time in ascending order along one of the coordinate axes.* We compute the convex hull $CH(S_{i-1})$ for points $S_{i-1} = \{p_1, p_2, \ldots, p_{i-1}\}$. For each added point p_i, we update $CH(S_{i-1})$ to get $CH(S_i)$, for $i = 2, 3, \ldots, n$, by deleting those t-faces, $t = 0, 1, \ldots, k-1$, that are internal to $CH(S_{i-1} \cup \{p_i\})$. It is shown by Seidel (see Edelsbrunner [1987])that $O(n^2 + \mathcal{H} \log n)$ time is sufficient. Most recently Chan [1995] obtained an algorithm based on the gift-wrapping method that runs in $O(n \log \mathcal{H} + (n\mathcal{H})^{1-1/(\lfloor k/2 \rfloor + 1)} \log^{O(1)} n)$ time. Note that the algorithm is optimal when $k = 2, 3$. In particular, it is optimal when $\mathcal{H} = o(n^{1-\epsilon})$ for some $0 < \epsilon < 1$.

We conclude this subsection with the following theorem [Chan 1995].

Theorem 11.6 *The convex hull of a set S of n points in $\Re^k$ can be computed in $O(n \log \mathcal{H})$ time for $k = 2$ or $k = 3$, and in $O(n \log \mathcal{H} + (n\mathcal{H})^{1-1/(\lfloor k/2 \rfloor + 1)} \log^{O(1)} n)$ time for $k > 3$, where $\mathcal{H}$ is the number of i-faces, $i = 0, 1, \ldots, k-1$.*

11.3.2 Proximity

In this subsection we address proximity related problems.

11.3.2.1 Closest Pair

Consider a set S of n points in $\Re^k$. The closest pair problem is to find in S a pair of points whose distance is the minimum, i.e., find p_i and p_j, such that $d(p_i, p_j) = \min_{k \neq l}\{d(p_k, p_l)$, for all points $p_k, p_l \in S\}$, where $d(a, b)$ denotes the Euclidean distance between a and b. (The subsequent result holds for any distance metric in Minkowski's norm.) The brute force method takes $O(d \cdot n^2)$ time by computing all $O(n^2)$ interpoint distances and taking the minimum; the pair that gives the minimum distance is the closest pair.

*If the points of S are not given a priori, the algorithm can be made *on line* by adding an extra step of checking if the newly added point is internal or external to the current convex hull. If internal, just discard it.

In one dimension, the problem can be solved by sorting these points and then scanning them in order, as the two closest points must occur consecutively. And this problem has a lower bound of $\Omega(n \log n)$ even in one dimension following from a linear time transformation from the *element uniqueness problem*. See Preparata and Shamos [1985].

But sorting is not applicable for dimension $k > 1$. Indeed this problem can be solved in optimal time $O(n \log n)$ by using the divide-and-conquer approach as follows. Let us first consider the case when $k = 2$. Consider a vertical cutting line λ that divides S into S_1 and S_2 such that $|S_1| = |S_2| = n/2$. Let δ_i be the minimum distance defined by points in $S_i, i = 1, 2$. Observe that the minimum distance defined by points in S can be either δ_1, δ_2, or defined by two points, one in each set. In the former case, we are done. In the latter, these two points must lie in the vertical strip of width $\delta = \min\{\delta_1, \delta_2\}$ on each side of the cutting line λ. The problem now reduces to that of finding the closest pair between points in S_1 and S_2 that lie inside the strip of width 2δ. This subproblem has a special property, known as the *sparsity* condition, i.e., the number of points in a box* of length 2δ is bounded by a constant $c = 4 \cdot 3^{k-1}$, because in each set S_i, there exists no point that lies in the interior of the δ-ball centered at each point in $S_i, i = 1, 2$ [Preparata and Shamos 1985]. It is this sparsity condition that enables us to solve the bichromatic closest pair problem (cf. the following subsection for more information) in $O(n)$ time. Let $\overline{S}_i \subseteq S_i$ denote the set of points that lies in the vertical strip. In two dimensions, the sparsity condition ensures that for each point in $\overline{S}_1$ the number of candidate points in $\overline{S}_2$ for the closest pair is at most 6. We therefore can scan these points $\overline{S}_1 \cup \overline{S}_2$ in order along the cutting line λ and compute the distance between each point scanned and its six candidate points. The pair that gives the minimum distance δ_3 is the bichromatic closest pair. The minimum distance of all pairs of points in S is then equal to $\delta_S = \min\{\delta_1, \delta_2, \delta_3\}$.

Since the merge step takes linear time, the entire algorithm takes $O(n \log n)$ time. This idea generalizes to higher dimensions, except that to ensure the sparsity condition the cutting hyperplane should be appropriately chosen to obtain an $O(n \log n)$ algorithm [Preparata and Shamos 1985].

11.3.2.2 Bichromatic Closest Pair

Given two sets of *red* and *blue* points, denoted R and B, respectively, find two points, one in R and the other in B, that are closest among all such mutual pairs.

The special case when the two sets satisfy the sparsity condition defined previously can be solved in $O(n \log n)$ time, where $n = |R| + |B|$. In fact a more general problem, known as *fixed radius all nearest-neighbor problem in a sparse set* [Bentley 1980, Preparata and Shamos 1985], i.e., given a set M of points in $\Re^k$ that satisfies the sparsity condition, find all pairs of points whose distance is less than a given parameter δ, can be solved in $O(|M| \log |M|)$ time [Preparata and Shamos 1985]. The bichromatic closest pair problem in general, however, seems quite difficult. Agarwal et al. [1991] gave an $O(n^{2(1-1/(\lceil k/2 \rceil+1))+\epsilon})$ time algorithm and a randomized algorithm with an expected running time of $O(n^{4/3} \log^c n)$ for some constant c. Chazelle et al. [1993] gave an $O(n^{2(1-1/(\lfloor k/2 \rfloor+1))+\epsilon})$ time algorithm for the bichromatic farthest pair problem, which can be used to find the diameter of a set S of points by setting $R = B = S$.

A lower bound of $\Omega(n \log n)$ for the bichromatic closest pair problem can be established. (See e.g., Preparata and Shamos [1985].) However, when the two sets are given as two simple polygons, the bichromatic closest pair problem can be solved relatively easily. Two problems can be defined. One is the *closest visible vertex pair* problem, and the other is the *separation problem*. In the former, one looks for a red–blue pair of vertices that are visible to each other and are the closest; in the latter, one looks for two boundary points that have the shortest distance. Both the closest visible vertex pair problem and the separation problem can be solved in linear time [Amato 1994, 1995]. But if both polygons are convex, the separation problem can be solved in $O(\log n)$ time [Chazelle and Dobkin 1987, Edelsbrunner 1985].

Additional references about different variations of closest pair problems can be found in Bespamyatnikh [1995], Callahan and Kosaraju [1995], Kapoor and Smid [1996], Schwartz et al. [1994], and Smid [1992].

*A box is also known as a hypercube.

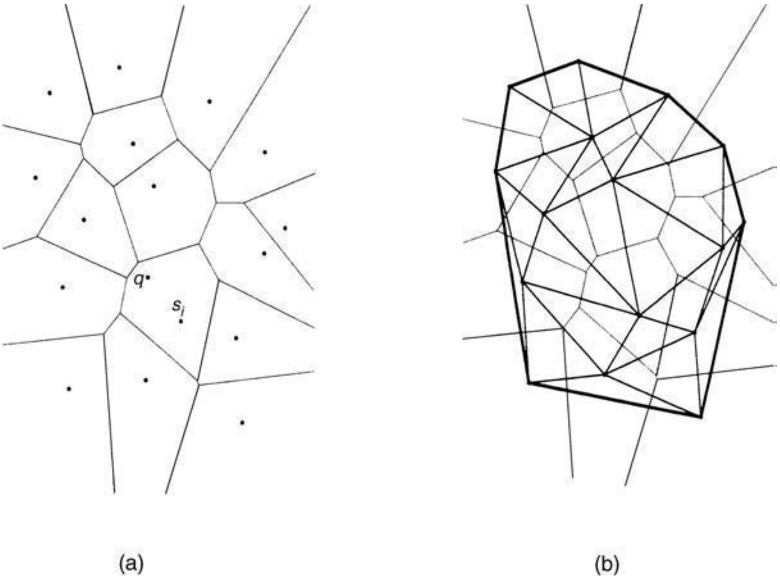


FIGURE 11.7 The Voronoi diagram of a set of 16 points in the plane.

11.3.2.3 Voronoi Diagrams

The Voronoi diagram $\mathcal{V}(S)$ of a set S of points, called *sites*, $S = \{s_1, s_2, \ldots, s_n\}$ in $\Re^k$ is a partition of $\Re^k$ into Voronoi cells $V(s_i)$, $i = 1, 2, \ldots, n$, such that each cell contains points that are closer to site s_i than to any other site s_j, $j \neq i$, i.e.,

$$V(s_i) = \{x \in \Re^k \mid d(x, s_i) \leq d(x, s_j) \forall s_j \in \Re^k, j \neq i\}$$

Figure 11.7a shows the Voronoi diagram of 16 point sites in two dimensions. Figure 11.7b shows the straight-line dual graph of the Voronoi diagram, which is called the Delaunay triangulation.

In two dimensions, $\mathcal{V}(S)$ is a planar graph and is of size linear in $|S|$. In dimensions $k \geq 2$, the total number of d-faces of dimensions $d = 0, 1, \ldots, k-1$, in $\mathcal{V}(S)$ is $O(n^{\lceil d/2 \rceil})$.

11.3.2.3.1 Construction of Voronoi Diagram in Two Dimensions

The Voronoi diagram possesses many properties that are proximity related. For instance, the closest pair problem for S can be solved in linear time after the Voronoi diagram has been computed. Because this pair of points must be adjacent in the Delaunay triangulation, all one has to do is examine all adjacent pairs of points and report the pair with the smallest distance. A divide-and-conquer algorithm to compute the Voronoi diagram of a set of points in the Euclidean plane was first given by Shamos and Hoey and generalized by Lee to L_p-metric for all $1 \leq p \leq \infty$ [Preparata and Shamos 1985]. A *plane-sweep* technique for constructing the diagram is proposed by Fortune [1987] that runs in $O(n \log n)$ time. There is a rich body of literature concerning the Voronoi diagram. The interested reader is referred to a recent survey by Fortune in Du and Hwang [1992, pp. 192–234].

Although $\Omega(n \log n)$ is the lower bound for computing the Voronoi diagram for an arbitrary set of n sites, this lower bound does not apply to special cases, e.g., when the sites are on the vertices of a convex polygon. In fact the Voronoi diagram of a convex polygon can be computed in linear time [Aggarwal et al. 1989]. This demonstrates further that an additional property of the input is to help reduce the complexity of the problem.

11.3.2.3.2 Construction of Voronoi Diagrams in Higher Dimensions

The Voronoi diagrams in $\Re^k$ are related to the convex hulls $\Re^{k+1}$ via a geometric transformation similar to duality discussed earlier in the subsection on geometric duality. Consider a set of n sites in $\Re^k$, which is the hyperplane $\mathcal{H}^0$ in $\Re^{k+1}$ such that $x_{k+1} = 0$, and a paraboloid $\mathcal{P}$ in $\Re^{k+1}$ represented as $x_{k+1} = x_1^2 + x_2^2 + \cdots + x_k^2$. Each site $s_i = (\mu_1, \mu_2, \ldots, \mu_k)$ is transformed into a hyperplane $\mathcal{H}(s_i)$ in $\Re^{k+1}$ denoted as

$$x_{k+1} = 2\sum_{j=1}^{k} \mu_j x_j - \left(\sum_{j=1}^{k} \mu_j^2\right)$$

That is, $\mathcal{H}(s_i)$ is tangent to the paraboloid $\mathcal{P}$ at a point $\mathcal{P}(s_i) = (\mu_1, \mu_2, \ldots, \mu_k, \mu_1^2 + \mu_2^2 + \cdots + \mu_k^2)$, which is just the vertical projection of site s_i onto the paraboloid $\mathcal{P}$. The half-space defined by $\mathcal{H}(s_i)$ and containing the paraboloid $\mathcal{P}$ is denoted as $\mathcal{H}^+(s_i)$. The intersection of all half-spaces, $\bigcap_{i=1}^{n} \mathcal{H}^+(s_i)$ is a convex body, and the boundary of the convex body is denoted $CH(\mathcal{H}(S))$. Any point $p \in \Re^k$ lies in the Voronoi cell $V(s_i)$ if the vertical projection of p onto $CH(\mathcal{H}(S))$ is contained in $\mathcal{H}(s_i)$. In other words, every κ-face of $CH(\mathcal{H}(S))$ has a vertical projection on the hyperplane $\mathcal{H}^0$ equal to the κ-face of the Voronoi diagram of S in $\mathcal{H}^0$.

We thus obtain the result which follows from Theorem 11.6 [Edelsbrunner 1987].

Theorem 11.7 *The Voronoi diagram of a set S of n points in $\Re^k$, $k \geq 3$, can be computed in $O(CH_{RH}(n))$ time and $O(n^{\lceil k/2 \rceil})$ space, where $CH_\ell(n)$ denotes the time for constructing the convex hull of n points in $\Re^\ell$.*

For more results concerning the Voronoi diagrams in higher dimensions and duality transformation see Aurenhammer [1990].

11.3.2.4 Farthest-Neighbor Voronoi Diagram

The Voronoi diagram defined in the preceding subsection is also known as the nearest-neighbor Voronoi diagram. A variation of this partitioning concept is a partition of the space into cells, each of which is associated with a site, which contains all points that are farther from the site than from any other site. This diagram is called the *farthest-neighbor* Voronoi diagram. Unlike the nearest-neighbor Voronoi diagram, only a subset of sites have a Voronoi cell associated with them. Those sites that have a nonempty Voronoi cell are those that lie on the convex hull of S. A similar partitioning of the space is known as the order κ-nearest-neighbor Voronoi diagram, in which each Voronoi cell is associated with a subset of κ sites in S for some fixed integer κ such that these κ sites are the closest among all other sites. For $\kappa = 1$ we have the nearest-neighbor Voronoi diagram, and for $\kappa = n - 1$ we have the farthest-neighbor Voronoi diagram. The higher order Voronoi diagrams in k-dimensions are related to the levels of hyperplane arrangements in $k + 1$ dimensions using the paraboloid transformation [Edelsbrunner 1987].

Because the farthest-neighbor Voronoi diagram is related to the convex hull of the set of sites, one can use the marriage-before-conquest paradigm of Kirkpatrick and Seidel [1986] to compute the farthest-neighbor Voronoi diagram of S in two dimensions in time $O(n \log \mathcal{H})$, where $\mathcal{H}$ is the number of sites on the convex hull.

11.3.2.5 Weighted Voronoi Diagrams

When the sites are associated with weights such that the distance function from a point to the sites is weighted, the structure of the Voronoi diagram can be drastically different than the unweighted case.

11.3.2.5.1 Power Diagrams

Suppose each site s in $\Re^k$ is associated with a nonnegative weight, w_s. For an arbitrary point p in $\Re^k$ the weighted distance from p to s is defined as

$$\delta(s, p) = d(s, p)^2 - w_s^2$$

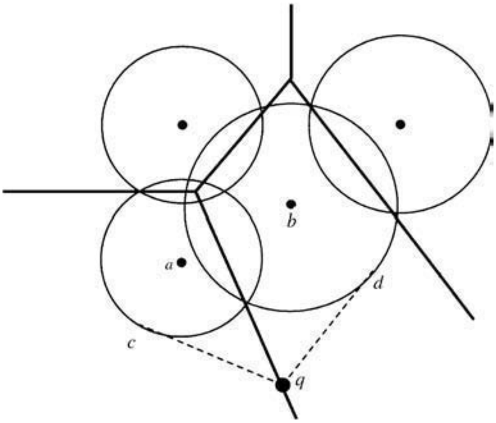


FIGURE 11.8 The power diagram in two dimensions; solid lines are equidistant to two sites.

If w_s is positive, and if $d(s, p) \geq w_s$, then $\sqrt{\delta(s, p)}$ is the length of the tangent of p to the ball $b(s)$ of radius w_s and centered at s. Here $\delta(s, p)$ is also called the *power of p* with respect to the ball $b(s)$. The locus of points p equidistant from two sites $s \neq t$ of equal weight will be a hyperplane called the *chordale* of s and t. See Figure 11.8. Point q is equidistant to sites a and b, and the distance is the length of the tangent line $\overline{q, c} = \overline{q, d}$.

The power diagram of two dimensions can be used to compute the contour of the union of n disks and the connected components of n disks in $O(n \log n)$ time, and in higher dimensions it can be used to compute the union or intersection of n axis-parallel cones in $\Re^k$ with apices in a common hyperplane in time $O(CH_{k+1}(n))$, the multiplicative weighted nearest-neighbor Voronoi diagram (defined subsequently) for n points in $\Re^k$ in time $O(CH_{k+2}(n))$, and the Voronoi diagrams for n spheres in $\Re^k$ in time $O(CH_{k+2}(n))$, where $CH_\ell(n)$ denotes the time for constructing the convex hull of n points in $\Re^\ell$ [Aurenhammer 1987]. For the best time bound for $CH_\ell(n)$ consult the subsection on convex hulls.

11.3.2.5.2 Multiplicative-Weighted Voronoi Diagrams

Each site $s \in \Re^k$ has a positive weight w_s, and the distance from a point p to s is defined as

$$\delta_{\text{multi}-w}(s, p) = d(p, s)/w_s$$

In two dimensions, the locus of points equidistant to two sites $s \neq t$ is a circle, if $w_s \neq w_t$, and a perpendicular bisector of line segment $\overline{s, t}$, if $w_s = w_t$. Each cell associated with a site s consists of all points closer to s than to any other site and may be disconnected. In the worst case the nearest-neighbor Voronoi diagram of a set S of n points in two dimensions can have an $O(n^2)$ regions and can be found in $O(n^2)$ time. In one dimension, the diagram can be computed optimally in $O(n \log n)$ time. However, the farthest-neighbor multiplicative-weighted Voronoi diagram has a very different characteristic. Each Voronoi cell associated with a site remains connected, and the size of the diagram is still linear in the number of sites. An $O(n \log^2 n)$ time algorithm for constructing such a diagram is given in Lee and Wu [1993]. See Schaudt and Drysdale [1991] for more applications of the diagram.

11.3.2.5.3 **Additive-Weighted Voronoi Diagrams**

The distance of a point p to a site s of a weight w_s is defined as

$$\delta_{\text{add}-w}(s, p) = d(p, s) - w_s$$

In two dimensions, the locus of points equidistant to two sites $s \neq t$ is a branch of a hyperbola, if $w_s \neq w_t$, and a perpendicular bisector of line segment $\overline{s, t}$ if $w_s = w_t$. The Voronoi diagram has properties similar to the ordinary unweighted diagram. For example, each cell is still connected and the size of the diagram is linear. If the weights are positive, the diagram is the same as the Voronoi diagram of a set of spheres centered at site s and of radius w_s, in two dimensions this diagram for n disks can be computed in $O(n \log^2 n)$ time [Lee and Drysdale 1981, Sharir 1985], and in $k \geq 3$ one can use the notion of power diagram to compute the diagram [Aurenhammer 1987].

11.3.2.6 Other Generalizations

The sites mentioned so far are point sites. They can be of different shapes. For instance, they can be line segments, disks, or polygonal objects. The metric used can also be a convex distance function or other norms. See Alt and Schwarzkopf [1995], Boissonnat et al. [1995], Klein [1989], and Yap [1987a] for more information.

11.3.3 Point Location

Point location is yet another fundamental problem in computational geometry. Given a planar subdivision and a query point, one would like to find which region contains the point in question.

In this context, we are mostly interested in fast response time to answer repeated queries to a fixed database. An earlier approach is based on the *slab method* [Preparata and Shamos 1985], in which parallel lines are drawn through each vertex, thus partitioning the plane into parallel slabs. Each parallel slab is further divided into subregions by the edges of the subdivision that can be ordered. Any given point can thus be located by two binary searches: one to locate the slab containing the point among the $n + 1$ horizontal slabs, followed by another to locate the region defined by a pair of consecutive edges that are ordered from left to right. This requires preprocessing of the planar subdivision, and setting up suitable search tree structures for the slabs and the edges crossing each slab. We use a three-tuple, $(P(n), S(n), Q(n)) =$ (preprocessing time, space requirement, query time) to denote the performance of the search strategy (cf. section on dynamization). The slab method gives an $(O(n^2), O(n^2), O(\log n))$ algorithm. Because preprocessing time is only performed once, the time requirement is not as critical as the space requirement. The primary goal of the query processing problems is to minimize the query time and the space required.

Lee and Preparata first proposed a *chain decomposition* method to decompose a monotone planar subdivision with n points into a collection of $m \leq n$ monotone chains organized in a complete binary tree [Preparata and Shamos 1985]. Each node in the binary tree is associated with a monotone chain of at most n edges, ordered in the y-coordinate. Between two adjacent chains, there are a number of disjoint regions. Each query point is compared with the node, hence the associated chain, to decide on which side of the chain the query point lies. Each chain comparison takes $O(\log n)$ time, and the total number of nodes visited is $O(\log m)$. The search on the binary tree will lead to two adjacent chains and hence identify a region that contains the point. Thus, the query time is $O(\log m \log n) = O(\log^2 n)$. Unlike the slab method in which each edge may be stored as many as $O(n)$ times, resulting in $O(n^2)$ space, it can be shown that each edge in the planar subdivision, with an appropriate chain assignment scheme, is stored only once. Thus, the space requirement is $O(n)$. The chain decomposition scheme gives rise to an $(O(n \log n), O(n), O(\log^2 n))$ algorithm. The binary search on the chains is not efficient enough. Recall that after each *chain comparison*, we will move down the binary search tree to perform the next chain comparison and start over another binary search on the y-coordinate to find an edge of the chain, against which a comparison is made to decide if the point lies to the left or right of the chain. A more efficient scheme is to perform a binary search of the y-coordinate at the root node and to spend only $O(1)$ time per node as we go down the chain tree, shaving off an $O(\log n)$ factor from the query time [Edelsbrunner et al. 1986]. This scheme is similar to the ones adopted by Chazelle and Guibas [1986] in a fractional cascading search paradigm and by Willard [1985] in his range tree search method. With the linear time algorithm for triangulating a simple polygon due to Chazelle [1991] (cf. subsequent subsection on triangulation) we conclude with the following optimal search structure for planar point location.

Theorem 11.8 *Given a planar subdivision of n vertices, one can preprocess the subdivision in linear time and space such that each point location query can be answered in* $O(\log n)$ *time.*

The point location problem in arrangements of hyperplanes is also of significant interest. See, e.g., Chazelle and Friedman [1990]. Dynamic versions of the point location problem have also been investigated. See Chiang and Tamassia [1992] for a survey of dynamic computational geometry.

11.3.4 Motion Planning: Path Finding Problems

The problem is mostly cast in the following setting. Given are a set of obstacles O, an object, called *robot*, and an initial and final position, called source and destination, respectively. We wish to find a path for the robot to move from the source to the destination, avoiding all of the obstacles. This problem arises in several contexts. For instance, in robotics this is referred to as the *piano movers' problem* [Yap 1987b] or *collision avoidance* problem, and in VLSI routing this is the *wiring* problem for 2-terminal nets. In most applications we are searching for a collision avoidance path that has a shortest length, where the distance measure is based on the Euclidean or L_1-metric. For more information regarding motion planning see, e.g., Alt and Yap [1990] and Yap [1987b].

11.3.4.1 Path Finding in Two Dimensions

In two dimensions, the Euclidean shortest path problem in which the robot is a point and the obstacles are simple polygons, is well studied. A most fundamental approach is by using the notion of *visibility graph*. Because the shortest path must make turns at polygonal vertices, it is sufficient to construct a graph whose vertices are the vertices of the polygonal obstacles and the source and destination and whose edges are determined by vertices that are mutually *visible*, i.e., the segment connecting the two vertices does not intersect the interior of any obstacle. Once the visibility graph is constructed with edge weight equal to the Euclidean distance between the two vertices, one can then apply Dijkstra's shortest path algorithms [Preparata and Shamos 1985] to find a shortest path between the source and destination. The Euclidean shortest path between two points is referred to as the *geodesic path* and the distance as the *geodesic distance*. The computation of the visibility graph is the dominating factor for the complexity of any visibility graph-based shortest path algorithm. Research results aiming at more efficient algorithms for computing the visibility graph and for computing the geodesic path in time proportional to the size of the graph have been obtained. Ghosh and Mount [1991] gave an output-sensitive algorithm that runs in $O(E + n \log n)$ time for computing the visibility graph, where E denotes the number of edges in the graph.

Mitchell [1993] used the so-called *continuous Dijkstra* wave front approach to the problem for the general polygonal domain of n obstacle vertices and obtained an $O(n^{5/3+\epsilon})$ time algorithm. He constructed a *shortest path map* that partitions the plane into regions such that all points q that lie in the same region have the same vertex sequence in the shortest path from the given source to q. The shortest path map takes $O(n)$ space and enables us to perform shortest path queries, i.e., find a shortest path from the given source to any query points, in $O(\log n)$ time. Hershberger and Suri [1993] on the other hand, used a plane subdivision approach and presented an $O(n \log^2 n)$-time and $O(n \log n)$-space algorithm to compute the shortest path map of a given source point. They later improved the time bound to $O(n \log n)$. If the source-destination path is confined in a simple polygon with n vertices, the shortest path can be found in $O(n)$ time [Preparata and Shamos 1985].

In the context of VLSI routing one is mostly interested in rectilinear paths (L_1-metric) whose edges are either horizontal or vertical. As the paths are restricted to be rectilinear, the shortest path problem can be solved more easily. Lee et al. [1996] gave a survey on this topic.

In a two-layer VLSI routing model, the number of segments in a rectilinear path reflects the number of *vias*, where the wire segments change layers, which is a factor that governs the fabrication cost. In robotics, a straight-line motion is not as costly as making turns. Thus, the number of segments (or *turns*) has also become an objective function. This motivates the study of the problem of finding a path with the smallest number of segments, called the *minimum link path problem* [Mitchell et al. 1992, Suri 1990].

These two cost measures, length and number of links, are in conflict with each other. That is, a shortest path may have far too many links, whereas a minimum link path may be arbitrarily long compared with a shortest path. Instead of optimizing both measures *simultaneously*, one can seek a path that either optimizes a linear function of both length and the number of links or optimizes them in a lexicographical order. For example, we optimize the length first, and then the number of links, i.e., among those paths that have the same shortest length, find one whose number of links is the smallest, and vice versa.

A generalization of the collision-avoidance problem is to allow collision with a cost. Suppose each obstacle has a weight, which represents the cost if the obstacle is *penetrated*. Mitchell and Papadimitriou [1991] first studied the weighted region shortest path problem. Lee et al. [1991] studied a similar problem in the rectilinear case. Another generalization is to include in the set of obstacles some subset $F \subset O$ of obstacles, whose vertices are *forbidden* for the solution path to make turns. Of course, when the weight of obstacles is set to be ∞, or the forbidden set $F = \emptyset$, these generalizations reduce to the ordinary collision-avoidance problem.

11.3.4.2 Path Finding in Three Dimensions

The Euclidean shortest path problem between two points in a three-dimensional polyhedral environment turns out to be much harder than its two-dimensional counterpart. Consider a convex polyhedron P with n vertices in three dimensions and two points s, d on the surface of P. A shortest path from s to d on the surface will cross a sequence of edges, denoted $\xi(s, d)$. Here $\xi(s, d)$ is called the *shortest path edge sequence* induced by s and d and consists of distinct edges. If the edge sequence is known, the shortest path between s and d can be computed by a planar unfolding procedure so that these faces crossed by the path lie in a common plane and the path becomes a straight-line segment.

Mitchell et al. [1987] gave an $O(n^2 \log n)$ algorithm for finding a shortest path between s and d even if the polyhedron may not be convex. If s and d lie on the surface of two different polyhedra, Sharir [1987] gave an $O(N^{O(k)})$ algorithm, where N denotes the total number of vertices of k obstacles. In general, the problem of determining the shortest path edge sequence of a path between two points among k polyhedra is NP-hard [Canny and Reif 1987].

11.3.4.3 Motion Planning of Objects

In the previous sections, we discussed path planning for moving a point from the source to a destination in the presence of polygonal or polyhedral obstacles. We now briefly describe the problem of moving a polygonal or polyhedral object from an initial position to a final position subject to translational and/or rotational motions.

Consider a set of k convex polyhedral obstacles, O_1, O_2, ..., O_k, and a convex polyhedral robot, R in three dimensions. The motion planning problem is often solved by using the so-called *configuration space*, denoted $\mathcal{C}$, which is the space of parametric representations of possible robot placements [Lozano-Pérez 1983]. The free placement (FP) is the subspace of $\mathcal{C}$ of points at which the robot does not intersect the interior of any obstacle. For instance, if only translations of R are allowed, the free configuration space will be the union of the Minkowski sums $M_i = O_i \oplus (-R) = \{a - b \mid a \in O_i, b \in R\}$ for $i = 1, 2, \ldots, k$. A *feasible* path exists if the initial placement of R and final placement belong to the same connected component of FP. The problem is to find a continuous curve connecting the initial and final positions in FP. The combinatorial complexity, i.e., the number of vertices, edges, and faces on the boundary of FP, largely influences the efficiency of any $\mathcal{C}$-based algorithm. For translational motion planning, Aronov and Sharir [1994] showed that the combinatorial complexity of FP is $O(nk \log^2 k)$, where k is the number of obstacles defined above and n is the total complexity of the Minkowski sums $M_i, 1 \leq i \leq k$.

Moving a ladder (represented as a line segment) among a set of polygonal obstacles of size n can be done in $O(K \log n)$ time, where K denotes the number of pairs of obstacle vertices whose distance is less than the length of the ladder and is $O(n^2)$ in general [Sifrony and Sharir 1987]. If the moving robot is also a polygonal object, Avnaim et al. [1988] showed that $O(n^3 \log n)$ time suffices. When the obstacles are *fat** Van der Stappen and Overmars [1994] showed that the two preceding two-dimensional motion planning problems can be solved in $O(n \log n)$ time, and in three dimensions the problem can be solved in $O(n^2 \log n)$ time, if the obstacles are ℓ-fat for some positive constant ℓ.

11.3.5 Geometric Optimization

The geometric optimization problems arise in operations research, pattern recognition, and other engineering disciplines. We list some representative problems.

11.3.5.1 Minimum Cost Spanning Trees

The minimum (cost) spanning tree MST of an undirected, weighted graph $G(V, E)$, in which each edge has a nonnegative weight, is a well-studied problem in graph theory and can be solved in $O(|E| \log |V|)$ time [Preparata and Shamos 1985]. When cast in the Euclidean or other L_p-metric plane in which the input consists of a set S of n points, the complexity of this problem becomes different. Instead of constructing a *complete* graph whose edge weight is defined by the distance between its two endpoints, from which to extract an MST, a sparse graph, known as the *Delaunay triangulation* of the point set, is computed. It can be shown that the MST of S is a subgraph of the Delaunay triangulation. Because the MST of a planar graph can be found in linear time [Preparata and Shamos 1985], the problem can be solved in $O(n \log n)$ time. In fact, this is asymptotically optimal, as the closest pair of the set of points must define an edge in the MST, and the closest pair problem is known to have an $\Omega(n \log n)$ lower bound, as mentioned previously.

This problem in three or more dimensions can be solved in subquadratic time. For instance, in three dimensions $O((n \log n)^{1.5})$ time is sufficient [Chazelle 1985] and in $k \geq 3$ dimensions $O(n^{2(1-1/(\lceil k/2 \rceil+1))+\epsilon})$ time suffices [Agarwal et al. 1991].

11.3.5.2 Minimum Diameter Spanning Tree

The minimum *diameter* spanning tree (MDST) of an undirected, weighted graph $G(V, E)$ is a spanning tree such that the total weight of the longest path in the tree is minimum. This arises in applications to communication networks where a tree is sought such that the maximum delay, instead of the total cost, is to be minimized. A graph-theoretic approach yields a solution in $O(|E||V| \log |V|)$ time [Handler and Mirchandani 1979]. Ho et al. [1991] showed that by the triangle inequality there exists an MDST such that the longest path in the tree consists of no more than *three* segments. Based on this an $O(n^3)$ time algorithm was obtained.

Theorem 11.9 *Given a set S of n points, the minimum diameter spanning tree for S can be found in $\theta(n^3)$ time and $O(n)$ space.*

We remark that the problem of finding a spanning tree whose total cost and the diameter are both bounded is NP-complete [Ho et al. 1991]. A similar problem that arises in VLSI clock tree routing is to find a tree from a source to multiple sinks such that every source-to-sink path is the shortest and the total wire length is to be minimized. This problem still is not known to be solvable in polynomial time or NP-hard. Recently, we have shown that the problem of finding a minimum spanning tree such that the longest source-to-sink path is bounded by a given parameter is NP-complete [Seo and Lee 1995].

11.3.5.3 Minimum Enclosing Circle Problem

Given a set S of points, the problem is to find the smallest disk enclosing the set. This problem is also known as the (unweighted) one-center problem. That is, find a center such that the maximum distance

*An object $O \subseteq R^k$ is said to be *ℓ-fat* if for all hyperspheres S centered inside O and not fully containing O we have $\ell \cdot$ *volume* $(O \cap S) \geq$ *volume*(S).

from the center to the points in S is minimized. More formally, we need to find the center $c \in \Re^2$ such that $\max_{p_j \in S} d(c, p_j)$ is minimized. The weighted one-center problem, in which the distance function $d(c, p_j)$ is multiplied by the weight w_j, is a well-known minimax problem, also known as the *emergency center problem* in operations research. In two dimensions, the one-center problem can be solved in $O(n)$ time [Dyer 1986, Megiddo 1983b]. The minimum enclosing ball problem in higher dimensions is also solved by using a linear programming technique [Megiddo 1983b, 1984].

11.3.5.4 Largest Empty Circle Problem

This problem, in contrast to the minimum enclosing circle problem, is to find a circle centered in the interior of the convex hull of the set S of points that does not contain any given point and the radius of the circle is to be maximized. This is mathematically formalized as a maximin problem; the minimum distance from the center to the set is maximized. The weighted version is also known as the *obnoxious center* problem in facility location. An $O(n \log n)$ time solution for the unweighted version can be found in [Preparata and Shamos 1985].

11.3.5.5 Minimum Annulus Covering Problem

The *minimum annulus covering problem* is defined as follows. Given a set of S of n points find an annulus (defined by two concentric circles) whose center lies internal to the convex hull of S such that the *width* of the annulus is minimized. The problem arises in mechanical part design. To measure whether a circular part is *round*, an American National Standards Institute (ANSI) standard is to use the width of an annulus covering the set of points obtained from a number of measurements. This is known as the *roundness* problem [Le and Lee 1991]. It can be shown that the center of the annulus is either at a vertex of the nearest-neighbor Voronoi diagram, a vertex of the farthest-neighbor Voronoi diagram, or at the intersection of these two diagrams [Le and Lee 1991]. If the input is defined by a simple polygon P with n vertices, and the problem is to find a minimum-width annulus that contains the boundary of P, the problem can be solved in $O(n \log n + k)$, where k denotes the number of intersection points of the *medial axis* of the simple polygon and the boundary of P [Le and Lee 1991]. When the polygon is known to be convex, a linear time is sufficient [Swanson et al. 1995]. If the center of the smallest annulus of a point set can be arbitrarily placed, the center may lie at infinity and the annulus degenerates to a pair of parallel lines enclosing the set of points. This problem is different from the problem of finding the width of a set, which is to find a pair of parallel lines enclosing the set such that the distance between them is minimized. The width of a set of n points can be found in $O(n \log n)$ time, which is optimal [Lee and Wu 1986]. In three dimensions the *width* of a set is also used as a measure for flatness of a *plate—flatness* problem. Houle and Toussaint [1988] gave an $O(n^2)$ time algorithm, and Chazelle et al. [1993] improved it to $O(n^{8/5+\epsilon})$.

11.3.6 Decomposition

Polygon decomposition arises in pattern recognition in which recognition of a shape is facilitated by first decomposing it into simpler parts, called *primitives*, and comparing them to templates previously stored in a library via some similarity measure. The primitives are often convex, with the simplest being the shape of a triangle.

We consider two types of decomposition, *partition* and *covering*. In the former type, the components are pairwise disjoint except they may have some boundary edges in common. In the latter type, the components may overlap. A minimum decomposition is one such that the number of components is minimized. Sometimes additional points, called *Steiner points*, may be introduced to obtain a minimum decomposition. Unless otherwise specified, we assume that no Steiner points are used.

11.3.6.1 Triangulation

Triangulating a simple polygon or, in general, triangulating a planar straight-line graph, is a process of introducing noncrossing edges so that each face is a triangle. It is also a fundamental problem in computer graphics, geographical information systems, and finite-element methods.

Let us begin with the problem of triangulating a simple polygon with n vertices. It is obvious that for a simple polygon with n edges, one needs to introduce at most $n - 3$ diagonals to triangulate the interior into $n - 2$ triangles. This problem has been studied very extensively. A pioneering work is due to Garey et al., which gave an $O(n \log n)$ algorithm and a linear algorithm if the polygon is monotone [O'Rourke 1994, Preparata and Shamos 1985]. A breakthrough linear time triangulation result of Chazelle [1991] settled the long-standing open problem. As a result of this linear triangulation algorithm, a number of problems can be solved in linear time, for example, the simplicity test, defined subsequently, and many other shortest path problems inside a simple polygon [Guibas and Hershberger 1989]. Note that if the polygons have holes, the problem of triangulating the interior requires $\Omega(n \log n)$ time [Asano et al. 1986].

Sometimes we want to look for *quality* triangulation instead of just an arbitrary one. For instance, triangles with large or small angles are not desirable. It is well known that the Delaunay triangulation of points in general position is unique, and it will maximize the minimum angle. In fact, the characteristic angle vector* of the Delaunay triangulation of a set of points is *lexicographically maximum* [Lee 1978]. The notion of Delaunay triangulation of a set of points can be generalized to a planar straight-line graph $G(V, E)$. That is, we would like to have G as a subgraph of a triangulation $G'(V, E')$, $E \subseteq E'$, such that each triangle satisfies the *empty circumcircle* property; no vertex visible from the vertices of a triangle is contained in the interior of the circle. This *generalized* Delaunay triangulation was first introduced by Lee [1978] and an $O(n^2)$ (respectively, $O(n \log n)$) algorithm for constructing the generalized triangulation of a planar graph (respectively, a simple polygon) with n vertices was given in Lee and Lin [1986b]. Chew [1989] later improved the result and gave an $O(n \log n)$ time algorithm using divide-and-conquer. Triangulations that minimize the maximum angle or maximum edge length were also studied. But if constraints on the measure of the triangles, for instance, each triangle in the triangulation must be nonobtuse, then Steiner points must be introduced. See Bern and Eppstein (in Du and Hwang [1992, pp. 23–90]) for a survey of different criteria of triangulations and discussions of triangulations in two and three dimensions.

The problem of triangulating a set P of points in $\Re^k$, $k \geq 3$, is less studied. In this case, the convex hull of P is to be partitioned into $\mathcal{F}$ nonoverlapping simplices, the vertices of which are points in P. A simplex in k-dimensions consists of exactly $k + 1$ points, all of which are extreme points. Avis and ElGindy [1987] gave an $O(k^4 n \log_{1+1/k} n)$ time algorithm for triangulating a simplicial set of n points in $\Re^k$. In $\Re^3$ an $O(n \log n + \mathcal{F})$ time algorithm was presented and $\mathcal{F}$ is shown to be linear if no three points are collinear and at most $O(n^2)$ otherwise. See Du and Hwang [1992] for more references on three-dimensional triangulations and Delaunay triangulations in higher dimensions.

11.3.6.2 Other Decompositions

Partitioning a simple polygon into shapes such as convex polygons, star-shaped polygons, spiral polygons, monotone polygons, etc., has also been investigated [Toussaint 1985]. A linear time algorithm for partitioning a polygon into star-shaped polygons was given by Avis and Toussaint [1981] after the polygon has been triangulated. This algorithm provided a very simple proof of the traditional art gallery problem originally posed by Klee, i.e., $\lfloor n/3 \rfloor$ vertex guards are always sufficient to see the entire region of a simple polygon with n vertices. But if a minimum partition is desired, Keil [1985] gave an $O(n^5 N^2 \log n)$ time, where N denotes the number of reflex vertices. However, the problem of *covering* a simple polygon with a minimum number of star-shaped parts is NP-hard [Lee and Lin 1986a]. The problem of partitioning a polygon into a minimum number of convex parts can be solved in $O(N^2 n \log n)$ time [Keil 1985]. The minimum covering problem by star-shaped polygons for rectilinear polygons is still open. For variations and results of art gallery problems the reader is referred to O'Rourke [1987] and Shermer [1992]. Polynomial time algorithms for computing the minimum partition of a simple polygon into simpler parts while allowing Steiner points can be found in Asano et al. [1986] and Toussaint [1985].

*The characteristic angle vector of a triangulation is a vector of minimum angles of each triangle arranged in nondescending order. For a given point set, the number of triangles is the same for all triangulations, and therefore each of them is associated with a characteristic angle vector.

The minimum partition or covering problem for simple polygons becomes NP-hard when the polygons are allowed to have *holes* [Keil 1985, O'Rourke and Supowit 1983]. Asano et al. [1986] showed that the problem of partitioning a simple polygon with h holes into a minimum number of trapezoids with two horizontal sides can be solved in $O(n^{h+2})$ time and that the problem is NP-complete if h is part of the input. An $O(n \log n)$ time 3-approximation algorithm was presented. Imai and Asano [1986] gave an $O(n^{3/2} \log n)$ time and $O(n \log n)$ space algorithm for partitioning a rectilinear polygon with holes into a minimum number of rectangles (allowing Steiner points). The problem of covering a rectilinear polygon (without holes) with a minimum number of rectangles, however, is also NP-hard [Culberson and Reckhow 1988].

The problem of minimum partition into convex parts and the problem of determining if a nonconvex polyhedron can be partitioned into tetrahedra without introducing Steiner points are NP-hard [O'Rourke and Supowit 1983, Ruppert and Seidel 1992].

11.3.7 Intersection

This class of problems arises in architectural design, computer graphics [Dorward 1994], etc., and encompasses two types of problems, *intersection detection* and *intersection computation*.

11.3.7.1 Intersection Detection Problems

The intersection detection problem is of the form: Given a set of objects, do any two intersect? The intersection detection problem has a lower bound of $\Omega(n \log n)$ [Preparata and Shamos 1985]. The pairwise intersection detection problem is a precursor to the general intersection detection problem.

In two dimensions the problem of detecting if two polygons of r and b vertices intersect was easily solved in $O(n \log n)$ time, where $n = r + b$ using the red–blue segment intersection algorithm [Mairson and Stolfi 1988]. However, this problem can be reduced in linear time to the problem of detecting the self-intersection of a polygonal curve. The latter problem is known as the *simplicity* test and can be solved optimally in linear time by Chazelle's [1991] linear time triangulation algorithm. If the two polygons are convex, then $O(\log n)$ suffices [Chazelle and Dobkin 1987, Edelsbrunner 1985]. We remark here that, although detecting whether two convex polygons intersect can be done in logarithmic time, detecting whether the boundary of the two convex polygons intersects requires $\Omega(n)$ time [Chazelle and Dobkin 1987].

In three dimensions, detecting if two convex polyhedra intersect can be solved in linear time by using a hierarchical representation of the convex polyhedron, or by formulating it as a linear programming problem in three variables [Chazelle and Dobkin 1987, Dobkin and Kirkpatrick 1985, Dyer 1984, Megiddo 1983b].

For some applications, we would not only detect intersection but also *report* all such intersecting pairs of objects or *count* the number of intersections, which is discussed next.

11.3.7.2 Intersection Reporting/Counting Problems

One of the simplest of such intersecting reporting problems is that of *reporting* all intersecting pairs of line segments in the plane. Using the plane sweep technique, one can obtain an $O((n + \mathcal{F}) \log n)$ time, where $\mathcal{F}$ is the output size. It is not difficult to see that the lower bound for this problem is $\Omega(n \log n + \mathcal{F})$; thus the preceding algorithm is $O(\log n)$ factor from the optimal. Recently, this segment intersection reporting problem was solved optimally by Chazelle and Edelsbrunner [1992], who used several important algorithm design and data structuring techniques as well as some crucial combinatorial analysis. In contrast to this asymptotically optimal *deterministic* algorithm, a simpler randomized algorithm for this problem that takes $O(n \log n + \mathcal{F})$ time but requires only $O(n)$ space (instead of $O(n + \mathcal{F})$) was obtained [Du and Hwang 1992]. Balaban [1995] recently reported a deterministic algorithm that solves this problem optimally both in time and space.

On a separate front, the problem of finding intersecting pairs of segments from different sets was considered. This is called the *bichromatic line segment* intersection problem. Nievergelt and Preparata [1982] considered the problem of merging two planar convex subdivisions of total size n and showed that the resulting subdivision can be computed in $O(n \log n + \mathcal{F})$ time. This result [Nievergelt and Preparata 1982] was extended in two ways. Mairson and Stolfi [1988] showed that the bichromatic line segment intersection reporting problem can be solved in $O(n \log n + \mathcal{F})$ time. Guibas and Seidel [1987] showed that merging two convex subdivisions can actually be solved in $O(n + \mathcal{F})$ time using topological plane sweep.

Most recently, Chazelle et al. [1994] used *hereditary segment trees* structure and *fractional cascading* [Chazelle and Guibas 1986] and solved both segment intersection reporting and counting problems optimally in $O(n \log n)$ time and $O(n)$ space. (The term $\mathcal{F}$ should be included for reporting.)

The *rectangle intersection reporting* problem arises in the design of VLSI circuitry, in which each rectangle is used to model a certain circuitry component. This is a well-studied classic problem and optimal algorithms ($O(n \log n + \mathcal{F})$ time) have been reported (see Lee and Preparata [1984] for references). The k-dimensional hyperrectangle intersection reporting (respectively, counting) problem can be solved in $O(n^{k-2} \log n + \mathcal{F})$ time and $O(n)$ space [respectively, in time $O(n^{k-1} \log n)$ and space $O(n^{k-2} \log n)$].

11.3.7.3 Intersection Computation

Computing the actual intersection is a basic problem, whose efficient solutions often lead to better algorithms for many other problems.

Consider the problem of computing the common intersection of half-planes discussed previously. Efficient computation of the intersection of two convex polygons is required. The intersection of two convex polygons can be solved very efficiently by plane sweep in linear time, taking advantage of the fact that the edges of the input polygons are ordered. Observe that in each vertical strip defined by two consecutive sweep lines, we only need to compute the intersection of two trapezoids, one derived from each polygon [Preparata and Shamos 1985].

The problem of intersecting two convex polyhedra was first studied by Muller and Preparata [Preparata and Shamos 1985], who gave an $O(n \log n)$ algorithm by reducing the problem to the problems of intersection detection and convex hull computation. From this one can easily derive an $O(n \log^2 n)$ algorithm for computing the common intersection of n half-spaces in three dimensions by the divide-and-conquer method. However, using geometric duality and the concept of separating plane, Preparata and Muller [Preparata and Shamos 1985] obtained an $O(n \log n)$ algorithm for this problem, which is asymptotically optimal. There appears to be a difference in the approach to solving the common intersection problem of half-spaces in two and three dimensions. In the latter, we resorted to geometric duality instead of divide-and-conquer. This *inconsistency* was later resolved. Chazelle [1992] combined the hierarchical representation of convex polyhedra, geometric duality, and other ingenious techniques to obtain a linear time algorithm for computing the intersection of two convex polyhedra. From this result several problems can be solved optimally: (1) the common intersection of half-spaces in three dimensions can now be solved by divide-and-conquer optimally, (2) the merging of two Voronoi diagrams in the plane can be done in linear time by observing the relationship between the Voronoi diagram in two dimensions and the convex hull in three dimensions (cf. subsection on Voronoi diagrams), and (3) the medial axis of a simple polygon or the Voronoi diagram of vertices of a convex polygon can be solved in linear time.

11.3.8 Geometric Searching

This class of problems is cast in the form of query answering as discussed in the subsection on dynamization. Given a collection of objects, with preprocessing allowed, one is to find objects that satisfy the queries. The problem can be static or dynamic, depending on whether the database is allowed to change over the course of query-answering sessions, and it is studied mostly in modes, *count-mode* and *report-mode*. In the former case only the number of objects satisfying the query is to be answered, whereas in the latter the actual identity of the objects is to be reported. In the report mode the query time of the algorithm consists of two components, *search time* and *output*, and expressed as $Q_A(n) = O(f(n) + \mathcal{F})$, where n denotes the size of the database, $f(n)$ a function of n, and $\mathcal{F}$ the size of output. It is obvious that algorithms that handle the report-mode queries can also handle the count-mode queries ($\mathcal{F}$ is the answer). It seems natural to expect that the algorithms for count-mode queries would be more efficient (in terms of the order of magnitude of the space required and query time), as they need not search for the objects. However, it was argued that in the report-mode range searching, one could take advantage of the fact that since reporting takes time, the more there is to report, the *sloppier* the search can be. For example, if we were to know that the ratio $n/\mathcal{F}$ is $O(1)$, we could use a sequential search on a linear list. Chazelle in his seminal paper on filtering search capitalizes on this observation and improves the time complexity for searching for several problems [Chazelle 1986]. As indicated subsequently, the count-mode range searching problem is harder than the report-mode counterpart.

11.3.8.1 Range Searching Problems

This is a fundamental problem in database applications. We will discuss this problem and the algorithm in two dimensions. The generalization to higher dimensions is straightforward using a known technique [Bentley 1980]. Given is a set of n points in the plane, and the ranges are specified by a product $(l_1, u_1) \times (l_2, u_2)$. We would like to find points $p = (x, y)$ such that $l_1 \leq x \leq u_1$ and $l_2 \leq y \leq u_2$. Intuitively we want to find those points that lie inside a query rectangle specified by the range. This is called *orthogonal range searching*, as opposed to other kinds of range searching problems discussed subsequently. Unless otherwise specified, a range refers to an orthogonal range. We discuss the static case; as this belongs to the class of decomposable searching problems, the dynamization transformation techniques can be applied. We note that the range tree structure mentioned later can be made dynamic by using a weight-balanced tree, called a $BB(\alpha)$ tree [Mehlhorn 1984, Willard and Luecker 1985].

For count-mode queries this problem can be solved by using the locus method as follows. Divide the plane into $O(n^2)$ cells by drawing horizontal and vertical lines through each point. The answer to the query q, i.e., find the number of points dominated by q (those points whose x- and y-coordinates are both no greater than those of q) can be found by locating the cell containing q. Let it be denoted by $Dom(q)$. Thus, the answer to the count-mode range queries can be obtained by some simple arithmetic operations of $Dom(q_i)$ for the four corners of the query rectangle. We have $Q(k, n) = O(k \log n)$, $S(k, n) = P(k, n) = O(n^k)$. To reduce the space requirement at the expense of query time has been a goal of further research on this topic. Bentley [1980] introduced a data structure, called *range trees*. Using this structure the following results were obtained: for $k \geq 2$, $Q(k, n) = O(\log^{k-1} n)$, $S(k, n) = P(k, n) = O(n \log^{k-1} n)$. (See Lee and Preparata [1984] and Willard [1985] for more references.)

For report-mode queries, Chazelle [1986] showed that by using a filtering search technique the space requirement can be further reduced by a $\log \log n$ factor. In essence we use less space to allow for more objects than necessary to be found by the search mechanism, followed by a filtering process leaving out unwanted objects for output. If the range satisfies additional conditions, e.g., *grounded* in one of the coordinates, say, $l_1 = 0$, or the aspect ratio of the intervals specifying the range is fixed, then less space is needed. For instance, in two dimensions, the space required is linear (a saving of $\log n / \log \log n$ factor) for these two cases. By using the so-called functional approach to data structures Chazelle [1988] developed a *compression* scheme to encode the *downpointers* used by Willard [1985] to reduce further the space requirement. Thus in k-dimensions, $k \geq 2$, for the count-mode range queries we have $Q(k, n) = O(\log^{k-1} n)$ and $S(k, n) = O(n \log^{k-2} n)$ and for report-mode range queries $Q(k, n) = O(\log^{k-1} n + \mathcal{F})$, and $S(k, n) = O(n \log^{k-2+\epsilon} n)$ for some $0 < \epsilon < 1$.

11.3.8.2 Other Range Searching Problems

There are other range searching problems, called the simplex range searching problem and the half-space range searching problem that have been well studied. A simplex range in $\Re^k$ is a range whose boundary is specifed by $k + 1$ hyperplanes. In two dimensions it is a triangle.

The report-mode half-space range searching problem in the plane is optimally solved by Chazelle et al. [1985] in $Q(n) = O(\log n + \mathcal{F})$ time and $S(n) = O(n)$ space, using geometric duality transform. But this method does not generalize to higher dimensions. For $k = 3$, Chazelle and Preparata [1986] obtained an optimal $O(\log n + \mathcal{F})$ time algorithm using $O(n \log n)$ space. Agarwal and Matoušek [1995] obtained a more general result for this problem: for $n \leq m \leq n^{\lfloor k/2 \rfloor}$, with $O(m^{1+\epsilon})$ space and preprocessing,

$Q(k, n) = O((n/m^{1/\lfloor k/2 \rfloor}) \log n + \mathcal{F})$. As the half-space range searching problem is also decomposable (cf. earlier subsection on dynamization) standard dynamization techniques can be applied.

A general method for simplex range searching is to use the notion of the *partition tree*. The search space is partitioned in a hierarchical manner using cutting hyperplanes, and a search structure is built in a tree structure. Willard [1982] gave a sublinear time algorithm for count-mode half-space query in $O(n^{\alpha})$ time using linear space, where $\alpha \approx 0.774$, for $k = 2$. Using Chazelle's cutting theorem Matoušek showed that for k-dimensions there is a linear space search structure for the simplex range searching problem with query time $O(n^{1-1/k})$, which is optimal in two dimensions and within $O(\log n)$ factor of being optimal for $k > 2$. For more detailed information regarding geometric range searching see Matoušek [1994].

The preceding discussion is restricted to the case in which the database is a collection of points. One may consider other kinds of objects, such as line segments, rectangles, triangles, etc., depending on the needs of the application. The inverse of the orthogonal range searching problem is that of the *point enclosure searching problem*. Consider a collection of isothetic rectangles. The point enclosure searching problem is to find all rectangles that contain the given query point q. We can cast these problems as the *intersection searching* problems, i.e., given a set S of objects and a query object q, find a subset $\mathcal{F}$ of S such that for any $f \in \mathcal{F}$, $f \cap q \neq \emptyset$. We then have the rectangle enclosure searching problem, rectangle containment problem, segment intersection searching problem, etc. We list only a few references about these problems [Bistiolas et al. 1993, Imai and Asano 1987, Lee and Preparata 1982]. Janardan and Lopez [1993] generalized intersection searching in the following manner. The database is a collection of *groups* of objects, and the problem is to find all groups of objects intersecting a query object. A group is considered to be intersecting the query object if any object in the group intersects the query object. When each group has only one object, this reduces to the ordinary searching problems.

11.4 Conclusion

We have covered in this chapter a wide spectrum of topics in computational geometry, including several major problem solving paradigms developed to date and a variety of geometric problems. These paradigms include incremental construction, plane sweep, geometric duality, locus, divide-and-conquer, prune-and-search, dynamization, and random sampling. The topics included here, i.e., convex hull, proximity, point location, motion planning, optimization, decomposition, intersection, and searching, are not meant to be exhaustive. Some of the results presented are classic, and some of them represent the state of the art of this field. But they may also become classic in months to come. The reader is encouraged to look up the literature in major computational geometry journals and conference proceedings given in the references. We have not discussed parallel computational geometry, which has an enormous amount of research findings. Atallah [1992] gave a survey on this topic.

We hope that this treatment will provide sufficient background information about this field and that researchers in other science and engineering disciplines may find it helpful and apply some of the results to their own problem domains.

Acknowledgment

This material is based on work supported in part by the National Science Foundation under Grant CCR-9309743 and by the Office of Naval Research under Grants N00014-93-1-0272 and N00014-95-1-1007.

References

Agarwal, P., Edelsbrunner, H., Schwarzkopf, O., and Welzl, E. 1991. Euclidean minimum spanning trees and bichromatic closest pairs. *Discrete Comput. Geom.* 6(5):407–422.

Agarwal, P. and Matoušek, J. 1995. Dynamic half-space range reporting and its applications. *Algorithmica* 13(4):325–345.

Aggarwal, A., Guibas, L. J., Saxe, J., and Shor, P. W. 1989. A linear-time algorithm for computing the Voronoi diagram of a convex polygon. *Discrete Comput. Geom.* 4(6):591–604.

Alt, H. and Schwarzkopf, O. 1995. The Voronoi diagram of curved objects, pp. 89–97. In *Proc. 11th Ann. ACM Symp. Comput. Geom.*, June.

Alt, H. and Yap, C. K. 1990. Algorithmic aspect of motion planning: a tutorial, part 1 & 2. *Algorithms Rev.* 1(1, 2):43–77.

Amato, N. 1994. Determining the separation of simple polygons. *Int. J. Comput. Geom. Appl.* 4(4):457–474.

Amato, N. 1995. Finding a closest visible vertex pair between two polygons. *Algorithmica* 14(2):183–201.

Aronov, B. and Sharir, M. 1994. On translational motion planning in 3-space, pp. 21–30. In *Proc. 10th Ann. ACM Comput. Geom.*, June.

Asano, T., Asano, T., and Imai, H. 1986. Partitioning a polygonal region into trapezoids. *J. ACM* 33(2):290–312.

Asano, T., Asano, T., and Pinter, R. Y. 1986. Polygon triangulation: efficiency and minimality. *J. Algorithms* 7:221–231.

Asano, T., Guibas, L. J., and Tokuyama, T. 1994. Walking on an arrangement topologically. *Int. J. Comput. Geom. Appl.* 4(2):123–151.

Atallah, M. J. 1992. Parallel techniques for computational geometry. *Proc. of IEEE* 80(9):1435–1448.

Aurenhammer, F. 1987. Power diagrams: properties, algorithms and applications. *SIAM J. Comput.* 16(1):78–96.

Aurenhammer, F. 1990. A new duality result concerning Voronoi diagrams. *Discrete Comput. Geom.* 5(3):243–254.

Avis, D. and ElGindy, H. 1987. Triangulating point sets in space. *Discrete Comput. Geom.* 2(2):99–111.

Avis, D. and Toussaint, G. T. 1981. An efficient algorithm for decomposing a polygon into star-shaped polygons. *Pattern Recog.* 13:395–398.

Avnaim, F., Boissonnat, J. D., and Faverjon, B. 1988. A practical exact motion planning algorithm for polygonal objects amidst polygonal obstacles, pp. 67–86. In *Proc. Geom. Robotics Workshop*. J. D. Boissonnat and J. P. Laumond, eds. LNCS Vol. 391.

Balaban, I. J. 1995. An optimal algorithm for finding segments intersections, pp. 211–219. In *Proc. 11th Ann. Symp. Comput. Geom.*, June.

Bentley, J. L. 1980. Multidimensional divide-and-conquer. *Comm. ACM* 23(4):214–229.

Bentley, J. L. and Saxe, J. B. 1980. Decomposable searching problems I: static-to-dynamic transformation. *J. Algorithms* 1:301–358.

Bespamyatnikh, S. N. 1995. An optimal algorithm for closest pair maintenance, pp. 152–166. In *Proc. 11th Ann. Symp. Comput. Geom.*, June.

Bieri, H. and Nef, W. 1982. A recursive plane-sweep algorithm, determining all cells of a finite division of R^d. *Computing* 28:189–198.

Bistiolas, V., Sofotassios, D., and Tsakalidis, A. 1993. Computing rectangle enclosures. *Comput. Geom. Theory Appl.* 2(6):303–308.

Boissonnat, J.-D., Sharir, M., Tagansky, B., and Yvinec, M. 1995. Voronoi diagrams in higher dimensions under certain polyhedra distance functions, pp. 79–88. In *Proc. 11th Ann. ACM Symp. Comput. Geom.*, June.

Callahan, P. and Kosaraju, S. R. 1995. Algorithms for dynamic closests pair and n-body potential fields, pp. 263–272. In *Proc. 6th ACM–SIAM Symp. Discrete Algorithms.*

Canny, J. and Reif, J. R. 1987. New lower bound techniques for robot motion planning problems, pp. 49–60. In *Proc. 28th Annual Symp. Found. Comput. Sci.*, Oct.

Chan, T. M. 1995. Output-sensitive results on convex hulls, extreme points, and related problems, pp. 10–19. In *Proc. 11th ACM Ann. Symp. Comput. Geom.*, June.

Chazelle, B. 1985. How to search in history. *Inf. Control* 64:77–99.

Chazelle, B. 1986. Filtering search: a new approach to query-answering, *SIAM J. Comput.* 15(3):703–724.

Chazelle, B. 1988. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.* 17(3):427–462.

Chazelle, B. 1991. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.* 6:485–524.

Chazelle, B. 1992. An optimal algorithm for intersecting three-dimensional convex polyhedra. *SIAM J. Comput.* 21(4):671–696.

Chazelle, B. 1993. An optimal convex hull algorithm for point sets in any fixed dimension. *Discrete Comput. Geom.* 8(2):145–158.

Chazelle, B. and Dobkin, D. P. 1987. Intersection of convex objects in two and three dimensions. *J. ACM* 34(1):1–27.

Chazelle, B. and Edelsbrunner, H. 1992. An optimal algorithm for intersecting line segments in the plane. *J. ACM* 39(1):1–54.

Chazelle, B., Edelsbrunner, H., Guibas, L. J., and Sharir, M. 1993. Diameter, width, closest line pair, and parametric searching. *Discrete Comput. Geom.* 8(2):183–196.

Chazelle, B., Edelsbrunner, H., Guibas, L. J., and Sharir, M. 1994. Algorithms for bichromatic line-segment problems and polyhedral terrains. *Algorithmica* 11(2):116–132.

Chazelle, B. and Friedman, J. 1990. A deterministic view of random sampling and its use in geometry. *Combinatorica* 10(3):229–249.

Chazelle, B. and Friedman, J. 1994. Point location among hyperplanes and unidirectional ray-shooting. *Comput. Geom. Theory Appl.* 4(2):53–62.

Chazelle, B. and Guibas, L. J. 1986. Fractional cascading: I. a data structuring technique. *Algorithmica* 1(2):133–186.

Chazelle, B., Guibas, L. J., and Lee, D. T. 1985. The power of geometric duality. *BIT* 25:76–90.

Chazelle, B. and Matoušek, J. 1993. On linear-time deterministic algorithms for optimization problems in fixed dimension, pp. 281–290. In *Proc. 4th ACM–SIAM Symp. Discrete Algorithms*.

Chazelle, B. and Preparata, F. P. 1986. Halfspace range search: an algorithmic application of k-sets. *Discrete Comput. Geom.* 1(1):83–93.

Chew, L. P. 1989. Constrained Delaunay triangulations. *Algorithmica* 4(1):97–108.

Chiang, Y.-J. and Tamassia, R. 1992. Dynamic algorithms in computational geometry. *Proc. IEEE* 80(9):1412–1434.

Clarkson, K. L. 1986. Linear programming in $O(n3^{d^2})$ time. *Inf. Proc. Lett.* 22:21–24.

Clarkson, K. L. 1988. A randomized algorithm for closest-point queries. *SIAM J. Comput.* 17(4): 830–847.

Culberson, J. C. and Reckhow, R. A. 1988. Covering polygons is hard, pp. 601–611. In *Proc. 29th Ann. IEEE Symp. Found. Comput. Sci.*

Dobkin, D. P. and Kirkpatrick, D. G. 1985. A linear algorithm for determining the separation of convex polyhedra. *J. Algorithms* 6:381–392.

Dobkin, D. P. and Munro, J. I. 1981. Optimal time minimal space selection algorithms. *J. ACM* 28(3):454–461.

Dorward, S. E. 1994. A survey of object-space hidden surface removal. *Int. J. Comput. Geom. Appl.* 4(3):325–362.

Du, D. Z. and Hwang, F. K., eds. 1992. *Computing in Euclidean Geometry*. World Scientific, Singapore.

Dyer, M. E. 1984. Linear programs for two and three variables. *SIAM J. Comput.* 13(1):31–45.

Dyer, M. E. 1986. On a multidimensional search technique and its applications to the Euclidean one-center problem. *SIAM J. Comput.* 15(3):725–738.

Edelsbrunner, H. 1985. Computing the extreme distances between two convex polygons. *J. Algorithms* 6:213–224.

Edelsbrunner, H. 1987. *Algorithms in Combinatorial Geometry*. Springer–Verlag.

Edelsbrunner, H. and Guibas, L. J. 1989. Topologically sweeping an arrangement. *J. Comput. Syst. Sci.* 38:165–194; (1991) *Corrigendum* 42:249–251.

Edelsbrunner, H., Guibas, L. J., and Stolfi, J. 1986. Optimal point location in a monotone subdivision. *SIAM J. Comput.* 15(2):317–340.

Edelsbrunner, H., O'Rourke, J., and Seidel, R. 1986. Constructing arrangements of lines and hyperplanes with applications. *SIAM J. Comput.* 15(2):341–363.